ADOL-C: [1]

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

Version 2.7.3-trunk, February 2026

Andrea Walther[2] and Andreas Griewank[3]

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 random access memory required by the given function evaluation program. Derivative matrices are obtained by columns, by rows or in sparse format. 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. Since the data is accessed strictly sequentially it can be 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++

---

[2]Institute of Mathematics, University of Paderborn, 33098 Paderborn, Germany

[3]Department of Mathematics, Humboldt-Universität zu Berlin, 10099 Berlin, Germany

Contents

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

1.1 Introduction

The 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**++. In contrast to source transformation approaches, overloading does not generate intermediate source code. As starting points to retrieve further information on techniques and application of automatic differentiation, as well as on other AD tools, we refer to the book [9]. Furthermore, the web page `http://www.autodiff.org` of the AD community forms a rich source of further information and pointers.

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.

This manual is organized as follows. This section explains the modifications required to convert undifferentiated code to code that compiles with ADOL-C. Section 2 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. Descriptions of easy-to-use drivers for a convenient derivative evaluation are contained in Section 3. Section 4 offers a more mathematical characterization of the different modes of AD to compute derivatives. At the same time, the corresponding drivers of ADOL-C are explained. The overloaded derivative evaluation routines using the forward and the reverse mode of AD are explained in Section 5. Advanced differentiation techniques as the optimal checkpointing for time integrations, the exploitation of fixed point iterations, the usages of external differentiated functions and the differentiation of OpenMP parallel programs are described in Section 6. The traceless forward mode is presented in Section 7. Section 9 details the installation and use of the ADOL-C package. Finally, Section 10 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 code. The user should simply refer to them if the more abstract and general descriptions of ADOL-C provided in this document do not suffice.

1.2 Declaring Active Variables

The key ingredient of automatic differentiation by overloading is the concept of an *active variable*. All variables that may be considered as differentiable quantities at some time during the program execution must be of an active type. ADOL-C uses one active scalar type, called `adouble`, whose real part is of the standard type `double`. Typically, one will declare the independent variables and all quantities that directly or indirectly depend on

them as *active*. Other variables that do not depend on the independent variables but enter, for example, as parameters, may remain 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 x.value(). In particular, such explicit conversions are needed for the standard output procedure printf. The output stream operator ≪ is overloaded such that first the real part of an adouble and then the string "(a)" is added to the stream. The input stream operator ≫ can be used to assign a constant value to an adouble. Naturally, adoubles may be components of vectors, matrices, and other arrays, as well as members of structures or classes.

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.cpp` and its header `<adolc/adtb_types.h>`. The latter must be included for compilation of all program files containing adoubles and corresponding operations.

1.3   Marking Active Sections

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

$$\text{trace\_on(tag,keep)} \qquad \text{and} \qquad \text{trace\_off(file)}$$

are recorded on a sequential data set called *tape*. Pairs of these function calls can appear anywhere in a C++ program, but they must 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 2. 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 `<adolc/taputil.h>`, which is included by the header `<adolc/adtb_types.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 adoubles and with the header file `<adolc/adtb_types.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 adoubles allocated within an active section be deleted before its completion. The values of all adoubles that

exist at the beginning and end of an active section are automatically recorded by trace_on and trace_off, respectively.

1.4 Selecting Independent and Dependent Variables

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

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

This special initialization ensures that x.value() = 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

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

which ensures that py = y.value() 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.

1.5 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 adoubles 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 `<adolc/adtb_types.h>` included as described in Section 1.2. 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.

```
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 of computation
      t += z[i]*x[i];              // continue
    . . . . . . . . . . . .        // continue
    y[m-1] = t/m;                  // end of computation
}                                  // end of function
```

Figure 1: Generic example of a subprogram to be activated

To automatically compute derivatives of the dependent variables y with respect to the independent variables x, we can make the body of the function into an active section. For example, we may modify the previous program segment as in Figure 2. The renaming and doubling up of the original independent and dependent variable vectors by active counterparts may seem at first a bit clumsy. However, this transformation has the advantage that the calling sequence and the computational part, i.e., where the function is really evaluated, 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 10.

1.6   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 x.value(). Most functions contained in the ANSI C standard for the math library are overloaded for active arguments. The only exceptions are the non-differentiable 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. For-

```
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

tunately, one can easily see that fabs(x) and all its compositions with smooth functions are still directionally differentiable. These directional derivatives of arbitrary order can be propagated in the forward mode without any ambiguity. In other words, the forward mode as implemented in ADOL-C 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, ADOL-C sets the derivative of fabs(x) at the origin 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

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

yields for $a = b$ and $a' < b'$ the incorrect directional derivative value $c' = b'$ rather than the correct $c' = a'$. Therefore this form of conditional assignment should be avoided by use of the function $\text{fmin}(a, b)$. 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 $\pm$InfVal, 0 or NoNum, where InfVal and NoNum are user-defined parameters, see Section 2.2. On IEEE machines InfVal can be set to the special value $\text{Inf} = 1.0/0.0$ and NoNum to $\text{NaN} = 0.0/0.0$. For example, at $a = 0$ the first derivative b' of $b = \text{sqrt}(a)$ is set to

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

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

The general power function $\text{pow}(x, y) = x^y$ is computed whenever it is defined for the corresponding double arguments. If x is negative, however, the partial derivative 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 $\text{erf}(x)$. For the latter, we have included an adouble version in `<adouble.cpp>`, which has been commented out for systems on which the double valued version is not available. The increment and decrement operators $++$, $--$ (prefix and postfix) are available for adoubles. Ambiguous statements like $a \mathrel{+}= a++;$ must be avoided because the compiler may sequence the evaluation of the overloaded expression differently from the original in terms of doubles.

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 9.6.

1.7   Reusing the Tape for Arbitrary Input Values

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 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. Therefore, the routines provided by ADOL-C for the evaluation of derivatives can be used to at arguments x other than the point at which the tape was generated, provided there are no user defined quadratures and all comparisons involving adoubles 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 ADOL-C and if it is not met the ADOL-C routine called for derivative calculations 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.
+1	At least one of the functions fmin, fmax or fabs is evaluated at a tie or zero, respectively. Hence, the function to be differentiated is Lipschitz-continuous but possibly non-differentiable.
0	Some arithmetic comparison involving adoubles yields a tie. Hence, the function to be differentiated 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 3: Return values around the taping point

In Figure 3 these return values are illustrated. If the user finds the return value of an ADOL-C routine to be negative the taping process simply has to be repeated by executing the active section again. 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. Therefore, when there are user-defined quadratures the retaping is necessary at each new point. If there are only branches conditioned on adouble comparisons one may hope that re-taping becomes unnecessary when the points settle down in some small neighborhood, as one would expect for example in an iterative equation solver.

1.8   Conditional Assignments

It appears unsatisfactory that, for example, a simple table lookup of some physical property forces the re-recording 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 introduce the two constructs of conditional assignments and active integers as partial remedies to the branching problem.

In many cases, the functionality of branches can be replaced by conditional assignments. For this purpose, we provide a special function called condassign(a,b,c,d). Its calling sequence corresponds to the syntax of the conditional assignment

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

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

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

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 run time value. Thus the original dependency of b on active variables gets lost, for instance if b is a comparison expression, see Section 1.6. 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 non-positive. The header file <adolc/adtb_types.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.

A generalization of this concept for more than two branches, e.g., akin to a `switch` statement or a cascade of `if...else if`, may be done by enabling ADOLC_ADVANCED_BRANCHING and performing selection on elements of an `advector` with active indices.

1.9   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.

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 <adolc/adtb_types.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.

2   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 three corresponding files. We will refer to these triplets as the tape array or tape file, in general tape, 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 re-taped 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 re-taped at the new point. Otherwise, direct evaluation from the tape by the routine function (Section 3.1) is likely to be faster. The use of quadratures and the results of all comparisons on adoubles are recorded on the tape so that function and other forward routines stop and return appropriate flags if their use without prior re-taping is unsafe. To avoid any re-taping certain types of branches can be recorded on the tape through the use of conditional assignments described before in Section 1.8.

Several tapes may be generated and kept simultaneously. A tape array is used as a triplet of buffers or a tape file is generated if the length of any of the buffers exceeds the maximal array lengths of OBUFSIZE, VBUFSIZE or LBUFSIZE. These parameters are defined in the header file `<adolc/internal/usrparms.h>` and may be adjusted by the user in the header file before compiling the ADOL-C library, or on runtime using a file named `.adolcrc`. Lines in this file must have the form

```
"VARIABLE" = "VALUE"
```

where the quotation marks are mandatory. The filesystem folder, where the tapes files may be written to disk, can be changed by changing the definition of TAPE_DIR in the header file `<adolc/dvlparms.h>` before compiling the ADOL-C library, or on runtime by defining TAPE_DIR in the `.adolcrc` file. By default this is defined to be the present working directory (`.`).

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 array or file called the Taylor stack. This option takes effect if keep = 1 and prepares the scene for an immediately following gradient evaluation by a call to a routine implementing the reverse mode as described in the Section 4 and Section 5. A file is used instead of an array if the size exceeds the maximal array length of TBUFSIZE defined in

`<adolc/internal/usrparms.h>` and may be adjusted in the same way like the other buffer sizes mentioned above. 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 Taylor stack 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 [OLVT]BUFSIZE elements.

After the execution of an active section, if a tape file was generated, i.e., if the length of some buffer exceeded [OLVT]BUFSIZE elements or if the argument file of trace_off was set to 1, the files will be saved in the directory defined as ADOLC_TAPE_DIR (by default the current working directory) under filenames formed by the strings ADOLC_OPERATIONS_NAME, ADOLC_LOCATIONS_NAME, ADOLC_VALUES_NAME and ADOLC_TAYLORS_NAME defined in the header file `<adolc/dvlparms.h>` appended with the number given as the tag argument to trace_on and have the extension .tap.

Later, all problem-independent routines like gradient, jacobian, forward, reverse, and others 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 tags, 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 $= 2$.

2.1   Examining the Tape and Predicting Storage Requirements

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

```
void tapestats(unsigned short tag, size_t* counts)
```

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

counts[NUM_INDEPENDENTS]:	the number of independents, i.e. calls to ≪= ,
counts[NUM_DEPENDENTS]:	the number of dependents, i.e. calls to ≫= ,
counts[NUM_MAX_LIVES]:	the maximal number of live active variables,
counts[TAY_STACK_SIZE]:	the size of Taylor stack (number of overwrites),
counts[OP_BUFFER_SIZE]:	the buffer size (a multiple of eight),

counts[NUM_OPERATIONS]:	the total number of operations recorded,
counts[...]:	other internal information about the tape.

The values maxlive = counts[NUM_MAX_LIVES] and tssize = counts[TAY_STACK_SIZE] determine the temporary storage requirements during calls to the routines implementing the forward and the reverse mode. For a certain degree $deg \geq 0$, the scalar version of the forward mode involves apart from the tape buffers an array of $(deg+1)*maxlive$ doubles in core and, in addition, a sequential data set called the value stack of $tssize*keep$ revreals if called with the option $keep > 0$. Here the type revreal is defined as double or float. 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 `<adolc/internal/usrparms.h>`. The drawback of the economical revreal = float choice is that subsequent calls to reverse mode implementations 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 reverse mode implementation involves the same number of doubles and twice as many revreals as the forward mode implementation. The storage requirements of the vector versions of the forward mode and reverse mode implementation are equal to that of the scalar versions multiplied by the vector length.

The function

$$printTapeStats(FILE* \ stream, \ short \ tag)$$

can be used to write the information above in human-readable form to the specified stream.

The function

$$tape\_doc(short \ tag, \ int \ n, \ int \ m, \ double* \ x, \ double* \ y)$$

tag:	tape number,
n:	number of dependents
m:	number if independents
x:	n values of dependents
y:	m values of indepentens

can be used to write the recorded tape to a LaTeX document named tape_TAG.tex with TAG being the tape number. This should only be used for small tapes to keep the size of the generated document within reasonable limits.

2.2 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 `<adolc/internal/usrparms.h>` to suit his problem and environment.

OBUFSIZE, LBUFSIZE, VBUFSIZE: These integer determines the length of internal buffers (default: 524 288). 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 these values occur 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. It is however recommended to leave the values in `<adolc/internal/usrparms.h>` unchanged and set them using the `.adolcrc` file in the current working directory at runtime.

TBUFSIZE: This integer determines the length of the internal buffer for a Taylor stack (default: 524 288).

TBUFNUM: This integer determines the maximal number of Taylor stacks (default: 32).

fint: The integer data type used by Fortran callable versions of functions.

fdouble: The floating point data type used by Fortran callable versions of functions.

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).

ADOLC_EPS: For testing on small numbers to avoid overflows (default: 10E-20).

DIAG_OUT: File identifier used as standard output for ADOL-C diagnostics (default: stdout).

The following types and options may be set using the command-line options of the `./configure` script.

locint: The range of the integer type locint determines how many adoubles can be simultaneously alive (default: unsigned int). In extreme cases when there are more than 2^{32} adoubles alive at any one time, the type locint must be changed to unsigned long. This can be done by passing `--enable-ulong` to `./configure`.

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. This can be done by passing `--disable-double` to `./configure`.

ATRIG_ERF: The overloaded versions of the inverse hyperbolic functions and the error function are enabled (default: undefined) by passing `--enable-atrig-erf` to `./configure`

ADOLC_ADVANCED_BRANCHING: Enables routines required for automatic branch selection (default: disabled). The boolean valued comparison operators with two `adouble` type arguments will not return boolean values anymore and may not be used in branch control statements (`if`, `while`, `for` etc.). Instead conditional assignments using `condassign` or selection operations on elements of `advector` type should be used. Enabling this option and rewriting the function evaluation using `condassign` or selections of `advector` elements will prevent the need for retracing the function at branch switches. This can be enabled by passing `--enable-advanced-branching` to `./configure`.

2.3 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 create dynamic arrays of adoubles by using the C++ operator new and

to destroy them using delete. For storage efficiency it is desirable that dynamic objects are created and destroyed in a last-in-first-out fashion.

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 <adolc/internal/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 shorts by using more local variables and deleting adoubles created by the new operator in a last-in-first-out fashion. When memory for adoubles is requested through a call to malloc() or other related C memory-allocating functions, the storage for these adoubles is allocated; however, the C++ adouble constructor is never called. The newly defined adoubles are never assigned a location and are not counted in the stack of live variables. Thus, any results depending upon these pseudo-adoubles will be incorrect. For these reasons **DO NOT use malloc() and related C memory-allocating functions when declaring adoubles (see the following paragraph).**

When an adouble 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 adoubles are placed on the top so that vectors of adoubles obtain a contiguous range of locations. While the C++ compiler can be expected to construct and destruct automatic variables in a last-in-first-out fashion, the user may upset this desirable pattern by deleting free-store adoubles too early or too late. Then the adouble stack may grow unnecessarily, but the numerical results will still be correct, unless an exception occurs because the range of locint is exceeded. In general, free-store adoubles 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 adoubles 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 adoubles may cause that the maximal number of adoubles alive at one time will be exceeded if the same active section is called repeatedly. The same effect occurs if static adoubles are used.

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 later 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 may not affect the dependent variables. For example, this is true for multipliers that are used to scale linear equations, but whose values do 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 adoubles to doubles. The deleterious effects of unnecessary activity are partly alleviated by run time activity flags in the derivative routine hov_reverse presented in Section 4.

The adouble default constructor sets to zero the associated value. This implies a certain overhead that may seem unnecessary when no initial value is actually given, however, the implicit initialization of arrays from a partial value list is the only legitimate construct (known to us) that requires this behavior. An array instantiation such as

<div align="center">double x[3]={2.0};</div>

will initialize x[0] to 2.0 and initialize (implicitly) the remaining array elements x[1] and x[2] to 0.0. According to the C++ standard the array element construction of the type changed instantiation

<div align="center">adouble x[3]={2.0};</div>

will use the constructor adouble(const double&); for x[0] passing in 2.0 but will call the adouble default constructor x[1] and x[2] leaving these array elements uninitialized *unless* the default constructor does implement the initialization to zero. The C++ constructor syntax does not provide a means to distinguish this implicit initialization from the declaration of any simple uninitialized variable. If the user can ascertain the absence of array instantiations such as the above then one can configure ADOL-C with the `--disable-stdczero` option , see Section 9.1, to avoid the overhead of these initializations.

3   Easy-To-Use Drivers

For the convenience of the user, ADOL-C provides several easy-to-use drivers that compute the most frequently required derivative objects. Throughout, we assume that after the execution of an active section, the corresponding tape with the identifier `tag` 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. We will denote this functional relation between the input variables x and the output variables y by

$$F : \mathbb{R}^n \mapsto \mathbb{R}^m, \qquad x \to F(x) \equiv y.$$

The return value of all drivers presented in this section indicate the validity of the tape as explained in Section 1.7. The presented 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.

3.1 Drivers for Optimization and Nonlinear Equations

The drivers provided for solving optimization problems and nonlinear equations are proto-typed in the header file `<adolc/drivers/drivers.h>`, which is included automatically by the global header file `<adolc/adolc.h>` (see Section 9.3).

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
double y[m];                     // dependent vector y = F(x)
```

If the original evaluation program is available this double version 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 and m = 1
double x[n];                     // independent vector x
double g[n];                     // resulting gradient ∇F(x)


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
double J[m][n];                  // resulting Jacobian F'(x)


int hessian(tag,n,x,H)
short int tag;                   // tape identification
int n;                           // number of independent variables n and m = 1
```

where the comments use the following inline math: $y = F(x)$, gradient $\nabla F(x)$, Jacobian $F'(x)$, and $m = 1$.

```
double x[n];                    // independent vector x
double H[n][n];                 // resulting Hessian matrix ∇²F(x)
```

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 $i + 1$ doubles 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 five 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
double u[m];                    // range weight vector u
double z[n];                    // result z = uᵀF'(x)
```

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 mode evaluation will be skipped and only reverse mode evaluation with the corresponding arguments is executed resulting in a reduced computational complexity of the function vec_jac.

```
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
double v[n];                    // tangent vector v
double z[m];                    // result z = F'(x)v
```

```
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
double v[n];                    // tangent vector v
double z[n];                    // result z = ∇²F(x)v
```

```
int hess_mat(tag,n,p,x,V,Z)
short int tag;                  // tape identification
```

```
int n;                         // number of independent variables n
int p;                         // number of columns in V
double x[n];                   // independent vector x
double V[n][p];                // tangent matrix V
double Z[n][p];                // result Z = ∇²F(x)V
```

```
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
double v[n];                   // tangent vector v
double u[m];                   // range weight vector u
double h[n];                   // result h = uᵀ∇²F(x)v
```

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

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

On entry, parameter b of the routine jac_solv contains the right-hand side of the equation $F(x)w = b$ to be solved. On exit, b equals the solution w of this equation. If mode $= 0$ only the Jacobian of the function given by the tape labeled with tag is provided internally. The LU-factorization of this Jacobian is computed for mode $= 1$. The solution of the equation is calculated if mode $= 2$. Hence, it is possible to compute the LU-factorization only once. Then the equation can be solved for several right-hand sides b without calculating the Jacobian and its factorization 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 adoubles have the same result as during taping. If this assumption is falsely made all drivers while internally calling the forward mode evaluation will return the value -1 or -2 as already specified in Section 1.7.

3.2   Drivers 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, e.g., $t = 0$, must satisfy the relation

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

with the y_j the Taylor coefficients of its derivative $y(t) = x'(t)$, namely,

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

defined by an autonomous right-hand side F recorded on the tape. Using this relation, one can generate the Taylor coefficients x_i, $i \leq deg$, recursively from the current point x_0. This task is achieved by the driver routine forode defined as follows:

```
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 the driver routine reverse or corresponding low level routines as explained in the Section 5 and Section 4, respectively, to compute the family of square matrices Z[n][n][deg] defined by

$$Z_j \equiv U \frac{\partial y_j}{\partial x_0} \in \mathbb{R}^{q \times n},$$

with double** U$= I_n$ the identity matrix of order n.

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

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

which exist provided F is sufficiently smooth. These matrices can be obtained from the partial derivatives $\partial y_i / \partial x_0$ by an appropriate version of the chain rule. To compute the total derivatives $B = (B_j)_{0 \leq j < d}$ defined in (1), one has to evaluate $\frac{1}{2}d(d-1)$ matrix-matrix products. This can be done by a call of the routine accode after the corresponding evaluation of the hov_reverse function. The interface of accode is defined as follows:

```
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 (1)
short nz[n][n];                 // optional nonzero pattern
```

Sparsity information can be exploited by accode using the array nz. For this purpose, nz has to be set by a call of the routine reverse or the corresponding basic routines as explained below in Section 4 and Section 5, respectively. The non-positive entries of nz are then changed by accode so that upon return

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

In other words, the matrices $B_k = \mathsf{B[\ ][\ ][k]}$ have a sparsity pattern that fills in as k grows. Note, that there need to be no loss in computational efficiency if a time-dependent ordinary differential equation is rewritten in autonomous form.

The prototype of the ODE-drivers forode and accode is contained in the header file `<adolc/drivers/odedrivers.h>`. The global header file `<adolc/adolc.h>` includes this file automatically, see Section 9.3.

An example program using the procedures forode and accode together with more detailed information about the coding can be found in Section 10.4. The corresponding source code `odexam.cpp` is contained in the subdirectory `examples`.

3.3   Drivers for Sparse Jacobians and Sparse Hessians

Quite often, the Jacobians and Hessians that have to be computed are sparse matrices. Therefore, ADOL-C provides additionally drivers that allow the exploitation of sparsity. The exploitation of sparsity is frequently based on *graph coloring* methods, discussed for example in [4] and [5]. The sparse drivers of ADOL-C presented in this section rely on the the coloring package ColPack (`https://github.com/CSCsw/ColPack`) developed by the authors of [4] and [5]. To use the sparse drivers presented below, ColPack must be installed either automatically (using -DENABLE_SPARSE=1) or manually (providing -DColPack_DIR). The drivers are in the namespace ADOLC::Sparse

Sparse Jacobians   To compute the entries of a sparse Jacobian matrix in coordinate format, ADOL-C provides the high-level driver sparse_jac. In contrast to earlier driver interfaces, all algorithmic options are selected at compile time via template parameters rather than by a runtime options array. This allows the compiler to optimize the chosen strategy while keeping the user interface concise.

```
int sparse_jac(tapeId,depen,indep,repeat,basepoint,
                 nnz,rind,cind,values)
short int tapeId;                  // tape identification
int depen;                         // number of dependent variables (rows) m
int indep;                         // number of independent variables (columns) n
int repeat;                        // control of pattern and seed reuse
const double *basepoint;           // independent vector x
int *nnz;                          // number of nonzero entries
unsigned int **rind;               // row indices (coordinate format)
unsigned int **cind;               // column indices (coordinate format)
double **values;                   // numerical nonzero values
```

The behavior of sparse_jac is fully determined by its template parameters:

- SparseMethod selects the method used for sparsity pattern detection (default: index-domain propagation),

- CompressionMode selects the compression strategy (default: column-wise compression),

- ControlFlowMode determines how control flow in the computational graph is handled (default: safe mode),

- BitPatternPropagationDirection selects the direction of bit-pattern propagation when applicable (default: automatic).

If repeat $= 0$, the routine determines the sparsity pattern of the Jacobian, generates a suitable seed matrix, and stores the associated metadata internally on the tape. The output variable nnz is set by the routine to the number of nonzero entries discovered. For repeat $\neq$ 0, the previously cached sparsity and seed information is reused to compute the numerical values of the Jacobian at the given point basepoint. In this case, nnz must match the value obtained during the initial call; otherwise, the routine reports an error. The array basepoint provides the current values of the independent variables. It is required whenever tight control-flow analysis is used; otherwise it may be nullptr. The Jacobian is returned in coordinate format via the arrays rind, cind, and values, each of length nnz. The arrays contain the row indices, column indices, and numerical values of the nonzero entries, respectively. Memory ownership for rind, cind, and values follows the semantics of the underlying routines. If the user supplies non-null pointers, the corresponding user-managed memory is used. Otherwise, the routine allocates unmanaged memory internally, and the caller is responsible for deallocating it using delete[] once the data is no longer needed. The routine returns a non-negative status value on success and a negative error code on failure. The driver routine is prototyped in the header file `<adolc/sparse/sparsedrivers.h>` and is included automatically by the global header file `<adolc/adolc.h>`.

Sparse Hessians   To compute the entries of a sparse Hessian matrix in coordinate format, ADOL-C provides the high-level driver sparse_hess. As for the sparse Jacobian driver, all algorithmic options are selected at compile time via template parameters rather than by a runtime options array.

```
int sparse_hess(tapeId,indep,repeat,basepoint,
                       nnz,rind,cind,values)
short int tapeId;                    // tape identification
int indep;                           // number of independent variables n
int repeat;                          // control of pattern and seed reuse
const double *basepoint;             // independent vector x
int *nnz;                            // number of nonzero entries
unsigned int **rind;                 // row indices (coordinate format)
unsigned int **cind;                 // column indices (coordinate format)
double **values;                     // numerical nonzero values
```

The behavior of sparse_hess is determined entirely by its template parameters:

- ControlFlowMode selects the handling of control flow during sparsity pattern computation (safe or tight),

- RecoveryMethod selects the ColPack recovery algorithm used for numerical Hessian reconstruction (indirect or direct).

The argument repeat controls whether sparsity pattern detection and seed generation are performed or reused. If repeat = 0, the routine computes the Hessian sparsity pattern, generates and caches the corresponding seed information on the tape, and sets nnz to the number of nonzero entries discovered. For repeat $\neq$ 0, the cached sparsity and seed information is reused to compute the numerical values of the Hessian at the point basepoint. In this case, nnz must match the value obtained during the initial call; otherwise, the routine reports an error. The array basepoint provides the current values of the independent variables. It is required when tight control-flow analysis is selected; otherwise it may be nullptr. The Hessian is returned in coordinate format via the arrays rind, cind, and values, each of length nnz. These arrays contain the row indices, column indices, and numerical values of the nonzero entries, respectively. Memory ownership for rind, cind, and values follows the semantics of the underlying recovery routines. If the user supplies non-null pointers, the corresponding user-managed memory is used. Otherwise, unmanaged memory is allocated internally, and the caller is responsible for releasing it using delete[]. The routine returns a non-negative status value on success and a negative error code on failure. The driver routine is prototyped in the header file `<adolc/sparse/sparsedrivers.h>` and is included automatically by the global header file `<adolc/adolc.h>`.

3.3.1   Computation of Sparsity Pattern

Computation of Jacobian Sparsity Pattern   ADOL-C provides a high-level driver for computing the sparsity pattern of a Jacobian matrix in compressed row storage (crs) format: jac_pat. The routine determines which independent variables influence each dependent variable and records this structure without computing numerical derivative values. All algorithmic choices are selected at compile time via template parameters rather than through a runtime options array.

```
int jac_pat(tapeId,depen,indep,basepoint,crs)
short int tapeId;                      // tape identification
int depen;                             // number of dependent variables m
int indep;                             // number of independent variables n
const double *basepoint;               // independent vector x
std::span<uint *> csr[m];              // compressed row sparsity structure
```

The behavior of jac_pat is determined by its template parameters:

- SparseMethod selects the method used to compute the sparsity pattern (index-domain or bit-pattern propagation)

- ControlFlowMode determines how control flow in the computational graph is handled (safe mode, tight mode);

- BitPatternPropagationDirection selects the direction of bit-pattern propagation when applicable (forward, reverse, or automatic selection).

The array basepoint provides the current values of the independent variables and is required when tight control-flow analysis is selected; otherwise it may be nullptr. The sparsity pattern is returned in compressed row storage format. The argument crs must be an array of length depen whose entries are pointers to arrays of type unsigned int. After the call, crs[i][0] contains the number of nonzero entries in row i $(0 \leq i < m)$, and the subsequent entries crs[i][1], ..., crs[i][crs[i][0]] store the zero-based column indices of the independent variables influencing the ith dependent. The routine returns a non-negative value equal to the total number of nonzero entries in the Jacobian sparsity pattern. In case of failure, a negative error code originating from the underlying propagation routine is returned. Ownership of the row buffers allocated for crs is transferred to the caller or the tape infrastructure. The user is responsible for ensuring that this memory is released appropriately once it is no longer required. The driver routine is prototyped in the header file `<adolc/sparse/sparsedrivers.h>` and is included automatically by the global header file `<adolc/adolc.h>`.

Computation of Hessian Sparsity Pattern   ADOL-C provides a driver routine for computing the sparsity pattern of a Hessian matrix in compressed row storage (crs) format: hess_pat. The routine determines which independent variables interact in the second derivatives of a scalar-valued function. All algorithmic choices are selected at compile time via template parameters rather than by runtime control flags.

```
int hess_pat(tapeId,indep,basepoint,crs)
short int tapeId;                      // tape identification
int indep;                             // number of independent variables n
const double *basepoint;               // independent vector x
std::span<uint *> crs[n];              // compressed row sparsity structure
```

The behavior of hess_pat is determined by the template parameter ControlFlowMode, which selects how control flow in the computational graph is handled during sparsity propagation (safe, tight, oldsafe, oldtight). The array basepoint provides the evaluation point for tight control-flow analysis and is required whenever a tight mode is selected; otherwise it may be nullptr. The sparsity pattern is returned in compressed row storage format. The argument crs must be an array of length indep whose entries are pointers to arrays of type unsigned int. After the call, crs[i][0] contains the number of nonzero entries in row i ($0 \le i < n$), and the subsequent entries crs[i][1], ..., crs[i][crs[i][0]] store the zero-based column indices of independent variables that interact with the ith variable in the Hessian. Before performing the computation, the routine resets the contents of crs. Memory allocated for the individual row buffers is owned by the caller or the tape infrastructure and must be released appropriately once it is no longer needed. The routine returns a non-negative status value on success. In case of failure, a negative error code forwarded from the underlying sparsity propagation routine is returned. The driver routine is prototyped in the header file `<adolc/sparse/sparsedrivers.h>` and is included automatically by the global header file `<adolc/adolc.h>`.

3.3.2   Calculation of Seed Matrices

To compute a compressed derivative matrix from a given sparsity pattern, one has to calculate an appropriate seed matrix that can be used as input for the derivative calculation. ADOL-C provides drivers to generate such seed matrices from sparsity patterns given in compressed row storage (crs) form. These drivers are thin wrappers around the ColPack library. In contrast to earlier interfaces, the compression orientation is selected at compile time via template parameters rather than by a runtime options array.

Seed matrix for Jacobians.   To generate a seed matrix for sparse Jacobian recovery, ADOL-C provides

```
void generate_seed_jac(m,n,JP,Seed,p)
int m;                      // number of dependent variables m
int n;                      // number of independent variables n
std::span<uint *> JP;       // crs sparsity pattern of the Jacobian
double ***Seed;             // seed matrix (allocated by ColPack)
int *p;                     // compressed dimension
```

The behavior of generate_seed_jac is determined by the template parameter Compression-Mode selecting the orientation of the compression (column or row). The input variables are the number of dependent variables m, the number of independent variables n, and the Jacobian sparsity pattern JP in compressed row storage form, as for example computed by jac_pat. Based on JP, the routine invokes ColPack's bipartite graph coloring to construct a seed matrix Seed suitable for recovering the full Jacobian from compressed directional evaluations. The output variable p denotes the compressed dimension implied by the coloring. Its interpretation depends on the selected compression orientation: for row compression, p is the number of columns of the seed matrix; for column compression, p corresponds to the compressed dimension in the sense of ColPack's recovery setup. The seed matrix Seed is allocated using ColPack's unmanaged API and returned to the caller through Seed. The caller is responsible for managing and releasing this memory in accordance with ColPack's unmanaged allocation semantics.

Seed matrix for Hessians. To generate a seed matrix for sparse Hessian recovery, ADOL-C provides the driver

```
void generate_seed_hess(n,HP,Seed,p)
int n;                      // number of independent variables n
std::span<uint *>HP[n];     // crs sparsity pattern of the Hessian
double ***Seed;             // seed matrix (allocated by ColPack)
int *p;                     // compressed dimension
```

The behavior of generate_seed_hess is determined by the template parameter RecoveryMethod, which selects the recovery strategy:

- Indirect: indirect Hessian recovery using an acyclic coloring strategy,

- Direct: direct Hessian recovery using a star coloring strategy.

The input variable HP provides the Hessian sparsity pattern in compressed row storage form, for example as computed by hess_pat. Based on this pattern and the selected recovery strategy, the routine invokes the corresponding ColPack coloring algorithm to construct a

seed matrix suitable for Hessian recovery. The output variable p denotes the compressed dimension implied by the coloring. The seed matrix Seed is allocated using ColPack's unmanaged API. Ownership of the returned memory follows ColPack's unmanaged semantics and must be respected by the caller.

The specific criteria used to construct seed matrices depend on whether the target derivative matrix is a Jacobian (nonsymmetric) or a Hessian (symmetric), and on whether the recovery from the compressed representation is performed directly or indirectly. Appropriate recovery routines are provided by ColPack and are used internally by the drivers sparse_jac and sparse_hess. These driver routines are prototyped in `<adolc/sparse/sparsedrivers.h>`, which is included automatically by the global header file `<adolc/adolc.h>` (see Section 9.3).

3.3.3 Legacy interface (runtime options).

Earlier versions of ADOL-C provided non-templated interfaces with runtime options array. These interfaces are retained for backward compatibility and select the underlying strategy at runtime. New code should prefer the typed interface described above and we only refer to the documentation of release 2.7.2 for the legacy interface.

3.4 Higher Derivative Tensors

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

$$\nabla_S^k F(x_0) = \left. \frac{\partial^k}{\partial z^k} F(x_0 + Sz) \right|_{z=0} \in \mathbb{R}^{p^k} \quad \text{for} \quad k = 0, \ldots, d \tag{2}$$

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 (2) in compressed form
double S[n][p];                // seed matrix S
```

Using the symmetry of the tensors defined by (2), 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 [8].

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 the function F, i.e., F_{i+1} for $i = 0, \ldots, m-1$. The sub-arrays 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 [12]. In the familiar quadratic case $d = 2$ the derivative with respect to z_j and z_k with z as in (2) 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 multi-index $j = (j_1, j_2, \ldots, j_d)$, 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 tensor_address(d, j)
int d;                         // highest derivative degree d
int j[d];                      // multi-index j
```

and it may thus be referenced as tensor[i][tensor_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 non-increasing. 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 non-increasing. 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 (2) 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][tensor_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][tensor_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`.

The drivers provided for efficient calculation of higher order derivatives are prototyped in the header file `<adolc/drivers/taylor.h>`, which is included by the global header file `<adolc/adolc.h>` automatically (see Section 9.3). Example codes using the above procedures can be found in the files `taylorexam.C` and `accessexam.C` contained in the subdirectory `examples/additional_examples/taylor`.

3.5   Derivatives of Implicit and Inverse Functions

Frequently, one needs 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

$$F(z) = \begin{pmatrix} G(z) \\ Pz \end{pmatrix} \equiv \begin{pmatrix} 0 \\ Pz \end{pmatrix} \equiv S\,x, \tag{3}$$

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 is 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 3.1 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. Now, 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 tuple 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 presented drivers are prototyped in the header file `<adolc/drivers/taylor.h>`. As indicated before this header is included by the global header file `<adolc/adolc.h>` automatically (see Section 9.3). The example programs `inversexam.cpp`, `coordinates.cpp` and `trigger.cpp` in the directory `examples/additional_examples/taylor` show the application of the procedures described here.

3.6   Drivers of the Toolbox for Lie Derivatives

Nonlinear controller and observer design often require certain types of Lie derivatives. These derivatives also arise in other areas such as classical mechanics and relativity. Lie derivatives are total derivatives of tensor fields along a vector field. The drivers for calculating Lie derivatives in ADOL-C presented in this section were developed by Klaus Röbenack [14] and his co-workers Jan Winkler, Siqian Wang and Mirko Franke [15]. They are prototyped as C functions in the header file `<adolc_lie.h>` and allow the calculation of Lie derivatives of scalar, vector and covector fields. In addition, the calculation of gradients of Lie derivatives of scalar fields is supported.

We consider computational problems occurring in controller and observer design for nonlinear control systems

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}) + \mathbf{g}(\mathbf{x})u, \quad \mathbf{y} = \mathbf{h}(\mathbf{x}), \quad \mathbf{x}(0) = \mathbf{x}_0 \in \Omega$$

with the scalar input u, the state $\mathbf{x}$, and the output $\mathbf{y}$, where the vector fields $\mathbf{f} : \Omega \to \mathbb{R}^n$, $\mathbf{g} : \Omega \to \mathbb{R}^n$ and the map $\mathbf{h} : \Omega \to \mathbb{R}^m$ (for $m = 1$ this map is the scalar field h) are defined on an open subset $\Omega \subseteq \mathbb{R}^n$.

Lie Derivatives of Scalar Fields

Consider the initial value problem

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}), \quad y = h(\mathbf{x}), \quad \mathbf{x}(0) = \mathbf{x}_0 \in \Omega$$

with the vector field $\mathbf{f} : \Omega \to \mathbb{R}^n$. The Lie derivative of the scalar field $h : \Omega \to \mathbb{R}$ along the vector field $\mathbf{f}$ are time derivatives of the curve y, i.e.,

$$\dot{y}(0) = L_{\mathbf{f}}h(\mathbf{x}_0), \ \ddot{y}(0) = L_{\mathbf{f}}^2 h(\mathbf{x}_0), \ \ldots, \ y^{(d)}(0) = L_{\mathbf{f}}^d h(\mathbf{x}_0).$$

Formally, the Lie derivative of the scalar field h along the vector field $\mathbf{f}$ is defined by

$$L_{\mathbf{f}}h(\mathbf{x}) = \frac{\partial h(\mathbf{x})}{\partial \mathbf{x}} \mathbf{f}(\mathbf{x}).$$

Higher order Lie derivatives are given by the recursion

$$L_{\mathbf{f}}^{k+1}h(\mathbf{x}) = \frac{\partial L_{\mathbf{f}}^k h(\mathbf{x})}{\partial \mathbf{x}} \mathbf{f}(\mathbf{x}) \quad \text{with} \quad L_{\mathbf{f}}^0 h(\mathbf{x}) = h(\mathbf{x}).$$

To compute Lie derivatives

$$(L_{\mathbf{f}}^0 h(\mathbf{x}_0), \ldots, L_{\mathbf{f}}^d h(\mathbf{x}_0))$$

we need the trace numbers of the active sections of the vector field $\mathbf{f}$ and the scalar field h, the number of independent variables n, the initial value $\mathbf{x}_0 \in \Omega$ and the highest derivative degree d. The values of the Lie derivatives are then stored in the one dimensional array `result` of length $d + 1$:

```
int lie_scalarc(Tape_F, Tape_H, n, x0, d, result)
short Tape_F;                // trace identification of vector field f
short Tape_H;                // trace identification of scalar field h
short n;                     // number of independent variables n and m = 1
double x0[n];                // values of independent variables x0
short d;                     // highest derivative degree d
double result[d+1];          // resulting Lie derivatives of a scalar field
```

For a smooth vector-valued map $\mathbf{h} : \Omega \to \mathbb{R}^m$ with $\mathbf{h}(\mathbf{x}) = (h_1(\mathbf{x}), \ldots, h_m(\mathbf{x}))^T$ we define the Lie derivative

$$L_{\mathbf{f}}^k \mathbf{h}(\mathbf{x}_0) = (L_{\mathbf{f}}^k h_1(\mathbf{x}_0), \ldots, L_{\mathbf{f}}^k h_m(\mathbf{x}_0))^T \in \mathbb{R}^m \quad \text{for} \quad k = 0, \cdots, d, \tag{4}$$

component-wise by the Lie derivatives $L_{\mathbf{f}}^k h_1(\mathbf{x}_0), \ldots, L_{\mathbf{f}}^k h_m(\mathbf{x}_0)$ of the m scalar fields $h_1, \ldots, h_m : \Omega \to \mathbb{R}$. Note that (4) should not be confused with the Lie derivative of a vector field. To compute Lie derivatives (4) of the vector-valued map $\mathbf{h}$ we additionally need the number of dependent variables m:

```
int lie_scalarcv(Tape_F, Tape_H, n, m, x0, d, result)
short Tape_F;                // trace identification of vector field f
short Tape_H;                // trace identification of vector-valued map h
short n;                     // number of independent variables n
short m;                     // number of dependent variables m
double x0[n];                // values of independent variables x0
short d;                     // highest derivative degree d
double result[m][d+1];       // resulting Lie derivatives of vectorial scalar fields
```

Gradients of Lie Derivatives of Scalar Fields

To compute the gradients

$$(\mathrm{d}L_{\mathbf{f}}^0 h(\mathbf{x}_0), \ldots, \mathrm{d}L_{\mathbf{f}}^d h(\mathbf{x}_0))$$

of the Lie derivatives $L_{\mathbf{f}}^0 h(\mathbf{x}_0), \ldots, L_{\mathbf{f}}^d h(\mathbf{x}_0)$ of the scalar field $h : \Omega \to \mathbb{R}$ along the vector field $\mathbf{f} : \Omega \to \mathbb{R}^n$, the following C function can be used:

```
int lie_gradientc(Tape_F, Tape_H, n, x0, d, result)
short Tape_F;                  // trace identification of vector field f
short Tape_H;                  // trace identification of scalar field h
short n;                       // number of independent variables n and m = 1
double x0[n];                  // values of independent variables x_0
short d;                       // highest derivative degree d
double result[n][d+1];         // resulting gradients of Lie derivatives
                               // of a scalar field
```

For calculating the jacobians $dL_{\mathbf{f}}^{k}\mathbf{h}(\mathbf{x}_0) \in \mathbb{R}^{m \times n}$ of Lie derivatives (4) of the vector-valued map $\mathbf{h} : \Omega \to \mathbb{R}^m$ for $k = 0, \ldots, d$ one can use the following function, where the $d+1$ matrices are stored in the three dimensional array `result`:

```
int lie_gradientcv(Tape_F, Tape_H, n, m, x0, d, result)
short Tape_F;                  // trace identification of vector field f
short Tape_H;                  // trace identification of vector-valued map h
short n;                       // number of independent variables n
short m;                       // number of dependent variables m
double x0[n];                  // values of independent variables x_0
short d;                       // highest derivative degree d
double result[m][n][d+1];      // resulting jacobians of Lie derivatives
                               // of vectorial scalar fields
```

Lie Derivatives of Covector Fields

If we consider the elements of the real vector space $\mathbb{R}^n$ as column vectors, the elements of the associated dual space $(\mathbb{R}^n)^*$ can be represented by row vectors. These row vectors are called adjoint vectors in ADOL-C and covectors in differential geometry. In a program the user does not have to distinguish between a vector field $\mathbf{f} : \Omega \to \mathbb{R}^n$ and a covector field $\boldsymbol{\omega} : \Omega \to (\mathbb{R}^n)^*$, since for both fields the dependent variables are stored in one dimensional arrays. The Lie derivative of a covector field $\boldsymbol{\omega}$ is defined by

$$L_{\mathbf{f}}\boldsymbol{\omega}(\mathbf{x}) = \boldsymbol{\omega}(\mathbf{x})\,\mathbf{f}'(\mathbf{x}) + \mathbf{f}^T(\mathbf{x}) \left(\frac{\partial \boldsymbol{\omega}^T(\mathbf{x})}{\partial \mathbf{x}} \right)^T .$$

The $d+1$ Lie derivatives $L_{\mathbf{f}}^0 \boldsymbol{\omega}(\mathbf{x}), \ldots, L_{\mathbf{f}}^d \boldsymbol{\omega}(\mathbf{x})$ of the covector field $\boldsymbol{\omega}$ along the vector field $\mathbf{f}$ can be computed by

```
int lie_covector(Tape_F, Tape_W, n, x0, d, result)
short Tape_F;                     // trace identification of vector field f
```

```
short Tape_W;              // trace identification of covector field ω
short n;                   // number of independent variables n
double x0[n];              // values of independent variables x₀
short d;                   // highest derivative degree d
double result[n][d+1];     // resulting Lie derivatives of a covector field
```

Lie Derivatives of a Vector Field (Lie Brackets)

Lie derivatives (Lie brackets) of a vector field $\mathbf{g} : \Omega \to \mathbb{R}^n$ along a vector field $\mathbf{f} : \Omega \to \mathbb{R}^n$ are defined by

$$\mathrm{ad}_{\mathbf{f}}\mathbf{g}(\mathbf{x}) = [\mathbf{f}, \mathbf{g}](\mathbf{x}) = \mathbf{g}'(\mathbf{x})\mathbf{f}(\mathbf{x}) - \mathbf{f}'(\mathbf{x})\mathbf{g}(\mathbf{x}).$$

We can calculate iterated Lie brackets $\mathrm{ad}_{\mathbf{f}}^0\mathbf{g}(\mathbf{x}), \ldots, \mathrm{ad}_{\mathbf{f}}^d\mathbf{g}(\mathbf{x})$ at the point $\mathbf{x}_0 \in \Omega$ using the C function listed below:

```
int lie_bracket(Tape_F, Tape_G, n, x0, d, result)
short Tape_F;              // trace identification of vector field f
short Tape_G;              // trace identification of vector field g
short n;                   // number of independent variables n
double x0[n];              // values of independent variables x₀
short d;                   // highest derivative degree d
double result[n][d+1];     // resulting Lie derivatives of a vector field
```

Additional C++ Drivers for Lie Derivatives

For Lie derivatives of scalar fields, the C interface offers two different drivers, namely `lie_scalarc` for $m = 1$ scalar field $h : \Omega \to \mathbb{R}$ and `lie_scalarcv` for $m \geq 1$ scalar fields $h_1, \ldots, h_m : \Omega \to \mathbb{R}$, which are combined in a vector-valued map $\mathbf{h} : \Omega \to \mathbb{R}^m$. Using the polymorphism in C++, both C drivers are unified by the C++ driver `lie_scalar` for the computation of Lie derivatives of scalar fields:

```
int lie_scalar(Tape_F, Tape_H, n, x0, d, result)        // case m = 1
int lie_scalar(Tape_F, Tape_H, n, m, x0, d, result)     // case m ≥ 1
```

The same situation occurs with the gradients or jacobians of Lie derivatives of scalar fields. Here, the C drivers `lie_gradientc` and `lie_gradientcv` are unified by the C++ driver `lie_gradient`:

```
int lie_gradient(Tape_F, Tape_H, n, x0, d, result)        // case m = 1
int lie_gradient(Tape_F, Tape_H, n, m, x0, d, result)     // case m ≥ 1
```

The C++ drivers are also prototyped in the header file <adolc_lie.h>. An example how to use these drivers for the calculation of Lie derivatives can be found in the file GantryCrane.cpp in the directory additional_examples/lie.

4   Basic Drivers for the Forward and Reverse Mode

In this section, we present tailored drivers for different variants of the forward mode and the reverse mode, respectively. For a better understanding, we start with a short description of the mathematical background.

Provided no arithmetic exception occurs, no comparison including fmax or fmin yields a tie, fabs does not yield zero, 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 denoted by $y = F(x)$ is in fact analytic. In other words, we can compute arbitrarily high derivatives of the vector function $F : I\!\!R^n \mapsto I\!\!R^m$ 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

$$x(t) \equiv \sum_{j=0}^{d} x_j t^j \; : \; I\!\!R \; \mapsto \; I\!\!R^n \tag{5}$$

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

$$x_j \; = \; \frac{1}{j!} \left. \frac{\partial^j}{\partial t^j} x(t) \right|_{t=0}$$

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

$$y(t) \; \equiv \; F(x(t)) \; : \; I\!\!R \; \mapsto \; I\!\!R^m \tag{6}$$

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

$$y(t) \; = \; \sum_{j=0}^{d} y_j t^j + O(t^{d+1}). \tag{7}$$

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

$$
\begin{aligned}
y_0 &= F(x_0) \\
y_1 &= F'(x_0)x_1 \\
y_2 &= F'(x_0)x_2 + \frac{1}{2}F''(x_0)x_1x_1 \\
y_3 &= F'(x_0)x_3 + F''(x_0)x_1x_2 + \frac{1}{6}F'''(x_0)x_1x_1x_1
\end{aligned}
\tag{8}
$$

$$\ldots$$

In writing down the last equations 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 identities

$$
\frac{\partial y_j}{\partial x_i} = \frac{\partial y_{j-i}}{\partial x_0} = A_{j-i}(x_0, x_1, \ldots, x_{j-i})
\tag{9}
$$

as established in [3]. This yields in particular

$$
\begin{aligned}
\frac{\partial y_0}{\partial x_0} = \frac{\partial y_1}{\partial x_1} = \frac{\partial y_2}{\partial x_2} = \frac{\partial y_3}{\partial x_3} &= A_0 = F'(x_0) \\
\frac{\partial y_1}{\partial x_0} = \frac{\partial y_2}{\partial x_1} = \frac{\partial y_3}{\partial x_2} &= A_1 = F''(x_0)x_1 \\
\frac{\partial y_2}{\partial x_0} = \frac{\partial y_3}{\partial x_1} &= A_2 = F''(x_0)x_2 + \frac{1}{2}F'''(x_0)x_1x_1 \\
\frac{\partial y_3}{\partial x_0} &= A_3 = F''(x_0)x_3 + F'''(x_0)x_1x_2 + \frac{1}{6}F^{(4)}(x_0)x_1x_1x_1
\end{aligned}
$$

$$\ldots$$

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 forward mode implementations and the ADOL-C reverse mode implementations. The scalar versions of the

forward mode propagate just one truncated Taylor series from the $(x_j)_{j \leq d}$ to the $(y_j)_{j \leq d}$. The vector versions of the forward mode propagate families of $p \geq 1$ such truncated Taylor series in order to reduce the relative cost of the overhead incurred in the tape interpretation. In detail, ADOL-C provides

```
int zos_forward(tag,m,n,keep,x,y)
short int tag;              // tape identification
int m;                      // number of dependent variables m
int n;                      // number of independent variables n
int keep;                   // flag for reverse mode preparation
double x[n];                // independent vector x = x0
double y[m];                // dependent vector y = F(x0)
```

for the **z**ero-**o**rder **s**calar forward mode. This driver computes $y = F(x)$ with $0 \leq$ keep ≤ 1. The integer flag keep plays a similar role as in the call to trace_on: It determines if zos_forward writes the first Taylor coefficients of all intermediate quantities into a buffered temporary file, i.e., the value stack, in preparation for a subsequent reverse mode evaluation. The value keep $= 1$ prepares for fos_reverse or fov_reverse as explained below.

To compute first-order derivatives, one has

```
int fos_forward(tag,m,n,keep,x0,x1,y0,y1)
short int tag;              // tape identification
int m;                      // number of dependent variables m
int n;                      // number of independent variables n
int keep;                   // flag for reverse mode preparation
double x0[n];               // independent vector x0
double x1[n];               // tangent vector x1
double y0[m];               // dependent vector y0 = F(x0)
double y1[m];               // first derivative y1 = F'(x0)x1
```

for the **f**irst-**o**rder **s**calar forward mode. Here, one has $0 \leq$ keep ≤ 2, where

$$\text{keep} = \begin{cases} 1 & \text{prepares for fos\_reverse or fov\_reverse} \\ 2 & \text{prepares for hos\_reverse or hov\_reverse} \end{cases}$$

as explained below. For the **f**irst-**o**rder **v**ector forward mode, ADOL-C provides

```
int fov_forward(tag,m,n,p,x0,X,y0,Y)
short int tag;               // tape identification
int m;                       // number of dependent variables m
int n;                       // number of independent variables n
int p;                       // number of directions
double x0[n];                // independent vector x0
```

```
double X[n][p];                 // tangent matrix X
double y0[m];                   // dependent vector y0 = F(x0)
double Y[m][p];                 // first derivative matrix Y = F'(x)X
```

For the computation of higher derivative, the driver

```
int hos_forward(tag,m,n,d,keep,x0,X,y0,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 mode preparation
double x0[n];                   // independent vector x0
double X[n][d];                 // tangent matrix X
double y0[m];                   // dependent vector y0 = F(x0)
double Y[m][d];                 // derivative matrix Y
```

implementing the **higher-order scalar** forward mode. 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 $X = \{x_j\}_{j=1\ldots d}$ represent Taylor coefficient vectors as in (5). 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 $Y = \{y_j\}_{j=1\ldots d}$ represent Taylor coefficient vectors as in (7), i.e., hos_forward computes the values $y_0 = F(x_0)$, $y_1 = F'(x_0)x_1$, ..., where $X = [x_1, x_2, \ldots, x_d]$ and $Y = [y_1, y_2, \ldots, y_d]$. Furthermore, one has $0 \leq \mathsf{keep} \leq d+1$, with

$$\mathsf{keep} \begin{cases} = 1 & \text{prepares for fos\_reverse or fov\_reverse} \\ > 1 & \text{prepares for hos\_reverse or hov\_reverse} \end{cases}$$

Once more, there is also a vector version given by

```
int hov_forward(tag,m,n,d,p,x0,X,y0,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 directions p
double x0[n];                   // independent vector x0
double X[n][p][d];              // tangent matrix X
double y0[m];                   // dependent vector y0 = F(x0)
double Y[m][p][d];              // derivative matrix Y
```

for the **higher-order vector** forward mode that computes $y_0 = F(x_0)$, $Y_1 = F'(x_0)X_1$, ..., where $X = [X_1, X_2, \ldots, X_d]$ and $Y = [Y_1, Y_2, \ldots, Y_d]$.

There are also overloaded versions providing a general forward-call. Details of the appropriate calling sequences are given in Section 5.

Once, the required information is generated due to a forward mode evaluation with an appropriate value of the parameter keep, one may use the following implementation variants of the reverse mode. To compute first-order derivatives one can use

```
int fos_reverse(tag,m,n,u,z)
short int tag;                    // tape identification
int m;                           // number of dependent variables m
int n;                           // number of independent variables n
double u[m];                     // weight vector u
double z[n];                     // resulting adjoint value zᵀ = uᵀF'(x)
```

as **first-order scalar** reverse mode implementation that computes the product $z^T = u^T F'(x)$ after calling zos_forward, fos_forward, or hos_forward with keep=1. The corresponding **first-order vector** reverse mode driver is given by

```
int fov_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 weight vectors q
double U[q][m];                  // weight matrix U
double Z[q][n];                  // resulting adjoint Z = UF'(x)
```

that can be used after calling zos_forward, fos_forward, or hos_forward with keep=1. To compute higher-order derivatives, ADOL-C provides

```
int hos_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];                     // weight vector u
double Z[n][d+1];                // resulting adjoints
```

as **higher-order scalar** reverse mode implementation yielding the adjoints $z_0^T = u^T F'(x_0) = u^T A_0$, $z_1^T = u^T F''(x_0) x_1 = u^T A_1$, ..., where $Z = [z_0, z_1, \ldots, z_d]$ after calling fos_forward or hos_forward with keep $= d + 1 > 1$. The vector version is given by

```
int hov_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];           // weight vector u
double Z[q][n][d+1];      // resulting adjoints
short int nz[q][n];       // nonzero pattern of Z
```

as **h**igher-**o**rder **v**ector reverse mode driver to compute the adjoints $Z_0 = UF'(x_0) = UA_0$, $Z_1 = UF''(x_0)x_1 = UA_1$, ..., where $Z = [Z_0, Z_1, \ldots, Z_d]$ after calling fos_forward or hos_forward with keep $= d + 1 > 1$. After the function call, the last argument of hov_reverse contains information about the sparsity pattern, i.e. each 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	non-smooth

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. "Non-smooth" 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.

There are also overloaded versions providing a general reverse-call. Details of the appropriate calling sequences are given in the following Section 5.

4.1 Drivers for Abs-Normal Form

In this subsection we consider functions $y = F(x) : \mathbb{R}^n \to \mathbb{R}^m$ that are non-smooth because of the occurrence of the absolute value function. The drivers provided generate a piecewise-linear approximation of the function F in a point x_0 and first order derivatives of this second order approximation. The piecewise-linear model will be called piecewise linearization in the following. Further information about the piecewise linearization you can find in [7].

The piecewise linearization is given in *abs-normal* form by

$$\begin{bmatrix} z \\ y \end{bmatrix} = \begin{bmatrix} c \\ b \end{bmatrix} + \begin{bmatrix} Z & L \\ J & Y \end{bmatrix} \begin{bmatrix} x \\ |z| \end{bmatrix}.$$

Here $z \in \mathbb{R}^s$ is a vector of $s \geq 0$ *switching variables* and correspondingly the two vectors and four matrices specifying the function F have the format

$$c \in \mathbb{R}^s, \quad Z \in \mathbb{R}^{s \times n}, \quad L \in \mathbb{R}^{s \times s}, \quad b \in \mathbb{R}^m, \quad J \in \mathbb{R}^{m \times n}, Y \in \mathbb{R}^{m \times s}.$$

The matrix L is strictly lower triangular.

To compute the piecewise linearization $\Delta F(x_0; x)$ the abs-normal mode has to be enabled by using

      enableMinMaxUsingAbs()

before the *trace_on*() command. If one is interested in the number s of absolute value functions occurring in the function evaluation, one can get it by

      int get_num_switches(tag)
      short int tag;   // tape identification

after tracing the function F.

In abs-normal mode several drivers are available. For a start there is a driver to evaluate $y = F(x)$. The driver also returns the values of z which are the arguments of the intermediate absolute value functions. The evaluations of the absolute value function is interpreted as $|z_i| = \sigma_i * z_i$ with $\sigma_i = sign(z_i)$ in the zero order mode. The vector $\sigma \equiv \{-1, 0, 1\}^s$ is called signature vector.

      int zos_pl_forward(tag,m,n,keep,x,y,z)
      short int tag;   // tape identification
      int n;   // number of independent variables n and $m = 1$
      int m;   // number of dependent variables m
      int keep;   // flag for reverse mode preparation
      double x[n];   // independent vector x
      double y[m];   // dependent vector $y = F(x)$
      double z[s];   // argument of **abs**(z)

Additionally, there are drivers for the forward mode to evaluate first order derivatives of the piecewise linearization. For first order derivatives another signature vector generated by

      int firstsign(z_i, $Z[i]$)
      double z[s];   // i-th component of argument of **abs**(z)
      double Z[s][p];   // i-th row of first derivative $Z = z\prime(x)X$

has to be used. *firstsign(u)* of a vector u is defined as the *sign()* of the first non-vanishing component of u if that exists; otherwise the value is zero. More detailed information can be found in [7].

      int fos_pl_forward(tag,m,n,x0,x1,y0,y1,z0,z2)

```
short int tag;                  // tape identification
int m;                          // number of dependent variables m
int n;                          // number of independent variables n
double x0[n];                   // independent vector x
double x1[n];                   // tangent vector x1
double y0[m];                   // dependent vector y0 = F(x0)
double y1[m];                   // first derivative y1 = F'(x0)x1
double z0[s];                   // argument of abs(z0)
double z1[s];                   // first derivative z1 = z'0(x0)x1


int fov_pl_forward(tag,m,n,p,x,X,y,Y,z,Z)
short int tag;                  // tape identification
int m;                          // number of dependent variables m
int n;                          // number of independent variables n
int p;                          // number of directions
double x[n];                    // independent vector x
double X[n][p];                 // tangent matrix X
double y[m];                    // dependent vector y = F(x)
double Y[m][p];                 // first derivative matrix Y = F'(x)X
double z[s];                    // argument of abs(z)
double Z[s][p];                 // first derivative matrix Z = z'(x)X
```

In contrast to the drivers above the reverse mode does not require any directions. It always returns one row of the Jacobian matrix.

```
int fov_pl_reverse(tag,m,n,s,rownum,z)
short int tag;                  // tape identification
int m;                          // number of dependent variables m
int n;                          // number of independent variables n
int s;                          // number of sign switches
int rownum;                     // required row no. of abs-normal form
double z[n];                    // resulting adjoint value z^T = u^T F'(x)
```

One may also compute the sparsity pattern of the extended Jacobian matrix

```
#include <adolc/sparse/sparsedrivers.h>
int absnormal_jac_pat(tag,m,n,s,x,JP)
short int tag;                  // tape identification
int m;                          // number of dependent variables m
int n;                          // number of independent variables n
```

```
int s;                          // number of sign switches
const double* x;                // point of evaluation
unsigned int** JP;              // sparsity pattern of abs-normal form
```

5 Overloaded Forward and Reverse Calls

In this section, the several 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 `<adolc/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 the Section 5.3 are prototyped in the header file `<adolc/sparse/sparsedrivers.h>`, which will be included by the header file `<adolc/interfaces.h>` automatically.

5.1 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 (7)
```

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 (5). 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 (7). 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 determines how many Taylor coefficients of all intermediate quantities are written into the value stack as explained in Section 4. 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. 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 reverse is executed in the scalar mode by the 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
```

to compute the adjoints $z_0^T = u^T F'(x_0) = u^T A_0$, $z_1^T = u^T F''(x_0) x_1 = u^T A_1$, ..., where $Z = [z_0, z_1, \ldots, z_d]$.

5.2   The Vector Case

When U is a matrix 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];                 // weight matrix U
double Z[q][n][d+1];            // resulting adjoints
short nz[q][n];                 // nonzero pattern of Z
```

to compute the adjoints $Z_0 = U F'(x_0) = U A_0$, $Z_1 = U F''(x_0) x_1 = U A_1$, ..., where $Z = [Z_0, Z_1, \ldots, Z_d]$. When the arguments p and U are omitted, they default to m and the identity matrix of order m, respectively.

Through the optional argument nz of reverse one can compute information about the sparsity pattern of Z as described in detail in the previous Section 4.

The return values of reverse calls can be interpreted according to Table 1, but negative return values are not valid, since 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,x0,X,y0,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 x0[n];                   // values of independent variables x0
double X[n][p][d];              // Taylor coefficients X of independent variables
double y0[m];                   // values of dependent variables y0
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 x0, y0 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 above in Section 1.7.

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 x0
double X[n][p];                 // seed derivatives of independent variables X
double y[m];                    // values of dependent variables y0
double Y[m][p];                 // first derivatives of dependent variables Y
```

When this routine is called with p $=$ n and X the identity matrix, the resulting 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 could be done for example using the driver sparse_jac 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.

5.3   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 [1]. Compared to the evaluation 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.

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 $s = 8*\mathsf{sizeof}(\mathsf{bitword\_t})$ 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 $Z \in \mathbb{N}^{q \times n}$ corresponding to

$$\tilde{Y} = \tilde{P}\tilde{X} \qquad \text{and} \qquad \tilde{Z} = \tilde{U}\tilde{P}, \tag{10}$$

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

Typed C++ interface.   The recommended interface selects the control-flow handling at compile time via the template parameter ControlFlowMode:

- Safe: conservative dependence propagation (default),

- Tight: dependence propagation sensitive to control flow, evaluated at a given base-point.

The dependence structures are represented as arrays of bitword_t and are passed as vectors of pointers, i.e. a packed matrix is provided as a list of row pointers.

The forward-mode bit-pattern propagation routine is

```
int forward(tapeId,m,n,p,x,X,y,Y)
short int tapeId;                    // tape identification
int m;                               // number of dependent variables m
int n;                               // number of independent variables n
int p;                               // number of packed words propagated p
double x[n];                         // basepoint values (tight mode only)
bitword_t *X[n];                     // input dependence structure (seed)
double y[m];                         // function values (tight mode only)
bitword_t *Y[m];                     // output dependence structure Y
```

In safe mode, the basepoint x and function values y are not required. In tight mode, x must be provided; then the routine evaluates the taped function at x in order to resolve control-flow dependencies and may write the corresponding function values into y.

The corresponding reverse-mode bit-pattern propagation routine is

```
int reverse(tapeId,m,n,q,U,Z)
short int tapeId;                    // tape identification
int m;                               // number of dependent variables m
int n;                               // number of independent variables n
int q;                               // number of packed words propagated q
bitword_t *U[q];                     // input dependence structure U
bitword_t *Z[q];                     // output dependence structure Z
```

yielding the dependence structure Z for a given dependence structure U. As in forward mode, safe propagation is conservative. Tight reverse propagation is control-flow sensitive and relies on the taped execution state corresponding to the selected basepoint; in practice this is typically established by a prior evaluation in tight mode.

Legacy (mode-selected) interface.   For backward compatibility, ADOL-C also provides the classic entry points documented in earlier versions of this manual. These interfaces select the control-flow handling at runtime using a mode argument: mode = 0 for safe (default) and mode = 1 for tight.

```
int forward(tapeId,m,n,p,x,X,y,Y,mode)
int reverse(tapeId,m,n,q,U,Z,mode)
```

These legacy routines do not allocate or free memory for the dependence structures; all buffers are managed by the caller.

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.

The return values of the bit pattern forward and reverse routines correspond to those described in Table 1.

One can control the storage growth by the factor p or q using "strip-mining" for successive groups of columns or rows at a time, i.e. partitioning X or U appropriately as described for the computation of Jacobians in Section 5.2. The strip-mining batch size can be influenced by the constant PQ_STRIPMINE_MAX in the sparse package configuration.

6 Advanced algorithmic differentiation in ADOL-C

6.1 Differentiating parameter dependent functions

Normally the functions to be differentiated using AD are defined as mappings from $\mathbb{R}^n$ to $\mathbb{R}^m$. The derivatives of the m dependents are then computed w.r.t. all the n independents. However sometimes the application requires only derivatives w.r.t. some of the independents and the others only occur in the evaluation procedure as parameters. One important such use case is Lagrange Multipliers in a nonlinear optimization problem. Declaring such variables as `adouble` and marking them as independent results in the evaluation of superfluous derivative information and increases run time. However, if the parameters are not defined as `adouble` type variables they get hard coded as constants onto the ADOL-C trace. One may want to change these parameters for some evaluations, however the overhead of retracing the function may be very high. In such cases these constants may be specially marked as parameters during tracing using the function

```
mkparam(d)
double d;                        // constant value to be marked as parameter
```

This stores the constant in a different place on the trace than the other unmarked constants and enables the user to replace a whole vector of parameters with a new one before computing derivatives later on. All such marked parameter constants can be used in any evaluation expression where the original unmarked constants were used.

Sometimes the same parameter may be needed in various different expressions. During trace creation the marked parameters are given a running index as they are marked. This

index may me saved by the user and used to access the same stored value in different expressions.

To save the index while marking a parameter use

```
locint mkparam_idx(d)
double d;                        // constant value to be marked
```

The resulting `locint` can be saved in a variable by the user and later used to access the saved parameter from in the trace in any expression

To access a saved parameter inside any evaluation expression use

```
getparam(idx)
locint idx;                      // index of the parameter as returned by mkparam_idx()
```

The above three functions work only during the creation of the trace, i.e., between the invocation of `trace_on()` and `trace_off()`.

For all following derivative computations using the drivers described in the preceding sections, the saved parameter values will be used, unless the user asks to replace them with a different vector of values.

To replace the whole parameter vector one uses the routine

```
set_param_vec(tag, number, parameters)
short int tag;                   // tape identification
size_t number;                   // total number of constants marked as parameters
double* parameters;              // array of constants to be used as replacement
```

All following derivative computations using the drivers described in the preceding sections will now use the new parameters.

After replacing the parameters, however, one cannot directly call a basic reverse mode driver. A basic forward mode driver must be called. However, the easy to use drivers from Section 3 do this automatically.

6.2   Differentiating external functions

Ideally, AD is applied to a given computation as a whole. This is not always possible because parts of the computation may be coded in a different programming language or may a call to an external library. In the former case one may want to differentiate the

parts in question with a different AD tool or provide hand written derivatives. In practice, however, sophisticated projects usually evolve over a long period of time. Within this process, a heterogeneous code base for the project develops, which may include the incorporation of external solutions, changes in programming paradigms or even of programming languages. Equally heterogeneous, the computation of derivative values appears. Hence, different AD-tools may be combined with hand-derived codes based on the same or different programming languages. ADOL-C supports such settings by the concept of externally differentiated functions, that is, a function not differentiated by ADOL-C itself. The required derivatives have to be provided by the user.

For this purpose, it is required that the externally differentiated function (for example named *euler_step*) has the following signature

> int euler_step(int n, double *x, int m, double *y); .

Note that the formal parameters in the signature have **double** type, that is, they are not active as in the original program before the ADOL-C type change. The externally differentiated function has to be *registered*[4] using an ADOL-C method as follows

> ext_diff_fct *edf = reg_ext_fct(euler_step); .

This returns a pointer to an ext_diff_fct instance specific to the registered function. Then, the user has to supply the function pointers for the callback methods (for example here zos_for_euler_step and fos_rev_euler_step) the user implemented to compute the derivatives as follows:

> edf->zos_forward = *zos_for_euler_step*;
> // function pointer for computing Zero-Order-Scalar (=zos)
> // forward information
> edf->fos_reverse = fos_rev_euler_step;
> // function pointer for computing First-Order-Scalar (=fos)
> reverse information

To facilitate the switch between active and passive versions of the parameters x and y one has to provide (allocate) both variants. I.e. if the call to euler_step was originally

> rc=euler_step(n, sOld, m, sNew);

then the ADOL-C typechange for the calling context will turn sOld and sNew in adouble pointers. To trigger the appropriate action for the derivative computation (i.e. creating an external differentiation entry on the trace) the original call to the externally differentiated function must be substituted by

> rc=call_ext_fct(edf, n, sOld, m, sNew); .

---

[4]we record the function pointer

The usage of the external function facility is illustrated by the example `ext_diff_func` contained in `examples/additional_examples/ext_diff_func`. There, the external differentiated function is also a C code, but the handling as external differentiated function decreases the overall required tape size.

6.3 Advanced algorithmic differentiation of time integration processes

For many time-dependent applications, the corresponding simulations are based on ordinary or partial differential equations. Furthermore, frequently there are quantities that influence the result of the simulation and can be seen as control of the systems. To compute an approximation of the simulated process for a time interval $[0, T]$ and evaluated the desired target function, one applies an appropriate integration scheme given by

some initializations yielding x_0
for $i = 0, \ldots, N - 1$
$\quad x_{i+1} = F(x_i, u_i, t_i)$
evaluation of the target function

where $x_i \in \mathbf{R}^n$ denotes the state and $u_i \in \mathbf{R}^m$ the control at time t_i for a given time grid $t_0, \ldots, t_N$ with $t_0 = 0$ and $t_N = T$. The operator $F : \mathbf{R}^n \times \mathbf{R}^m \times \mathbf{R} \mapsto \mathbf{R}^n$ defines the time step to compute the state at time t_i. Note that we do not assume a uniform grid.

When computing derivatives of the target function with respect to the control, the consequences for the tape generation using the "basic" taping approach as implemented in ADOL-C so far are shown in the left part of Figure 4. As can be seen, the iterative process
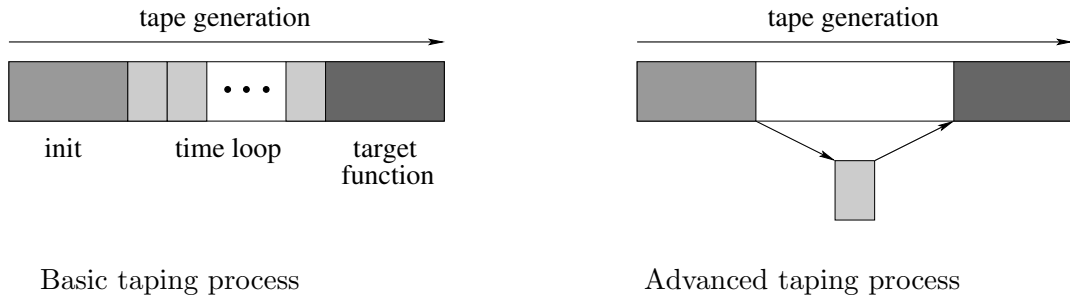

Figure 4: Different taping approaches

is completely unrolled due to the taping process. That is, the tape contains an internal representation of each time step. Hence, the overall tape comprises a serious amount of redundant information as illustrated by the light gray rectangles in Figure 4.

To overcome the repeated storage of essentially the same information, a *nested taping* mechanism has been incorporated into ADOL-C as illustrated on the right-hand side of Figure 4. This new capability allows the encapsulation of the time-stepping procedure such

that only the last time step $x_N = F(x_{N-1}, u_{N-1})$ is taped as one representative of the time steps in addition to a function pointer to the evaluation procedure F of the time steps. The function pointer has to be stored for a possibly necessary retaping during the derivative calculation as explained below.

Instead of storing the complete tape, only a very limited number of intermediate states are kept in memory. They serve as checkpoints, such that the required information for the backward integration is generated piecewise during the adjoint calculation. For this modified adjoint computation the optimal checkpointing schedules provided by **revolve** are employed. An adapted version of the software package revolve is part of ADOL-C and automatically integrated in the ADOL-C library. Based on revolve, c checkpoints are distributed such that computational effort is minimized for the given number of checkpoints and time steps N. It is important to note that the overall tape size is drastically reduced due to the advanced taping strategy. For the implementation of this nested taping we introduced a so-called "differentiating context" that enables ADOL-C to handle different internal function representations during the taping procedure and the derivative calculation. This approach allows the generation of a new tape inside the overall tape, where the coupling of the different tapes is based on the *external differentiated function* described above.

Written under the objective of minimal user effort, the checkpointing routines of ADOL-C need only very limited information. The user must provide two routines as implementation of the time-stepping function F with the signatures

```
int time_step_function(int n, adouble *u);
int time_step_function(int n, double *u);
```

where the function names can be chosen by the user as long as the names are unique.It is possible that the result vector of one time step iteration overwrites the argument vector of the same time step. Then, no copy operations are required to prepare the next time step.

At first, the adouble version of the time step function has to be *registered* using the ADOL-C function

```
CP_Context cpc(time_step_function);.
```

This function initializes the structure cpc. Then, the user has to provide the remaining checkpointing information by the following commands:

```
cpc.setDoubleFct(time_step_function);
// double variante of the time step function
cpc.setNumberOfSteps(N);
// number of time steps to perform
cpc.setNumberOfCheckpoints(10);
// number of checkpoint
cpc.setDimensionXY(n);
```

```
// dimension of input/output
cpc.setInput(y);
// input vector
cpc.setOutput(y);
// output vector
cpc.setTapeNumber(tag_check);
// subtape number for checkpointing
cpc.setAlwaysRetaping(false);
// always retape or not ?
```

Subsequently, the time loop in the function evaluation can be substituted by a call of the function

```
int cpc.checkpointing();
```

Then, ADOL-C computes derivative information using the optimal checkpointing strategy provided by revolve internally, i.e., completely hidden from the user.

The presented driver is prototyped in the header file `<adolc/checkpointing.h>`. This header is included by the global header file `<adolc/adolc.h>` automatically. An example program `checkpointing.cpp` illustrates the checkpointing facilities. It can be found in the directory `examples/additional_examples/checkpointing`.

6.4   Advanced algorithmic differentiation of fixed point iterations

One has to note that AD computes exact derivative information of the representation of F given by the computer program. Hence, in numerous cases AD provides exactly what the user of AD requires, but one has to keep this difference in mind. This is especially important if the computer program to evaluate $F(x)$ contains an iterative process, as for example a Newton iteration or a fixed point iteration. In these cases, the black-box application of AD may not yield the derivative information with the required accuracy. Furthermore, the resulting memory requirement may be unacceptable for real-world problems as for example large CFD codes to simulate a flow over an aircraft wing. For these large scale applications, frequently one introduces a pseudo time-stepping to compute a quasi-steady state using a given set of design parameters. This simulation forms one main ingredient to optimize the airfoil for a particular flight scenario. Hence, derivatives with respect to the design parameters are required if a calculus-based optimization method is applied. For this purpose, the pseudo time-stepping can be interpreted as a fixed point iteration. Then, it is possible to control the accuracy of the derivatives and to reduce the memory requirement using either a two-phase approach or the piggyback method as described for example in [9, Section 11.4].

The method of Reverse Accumulation proposed and analyzed in detail by Christianson [2] is one example for a two-phase approach. The Reverse Accumulation is available in

ADOL-C. Run time measurements for a large scale application, namely the derivative computation for the CFD code TAUij provided by the DLR Braunschweig [11], are presented and analyzed in [16].

6.4.1   Iterative evaluation of derivatives of fixed point iterations

Quite often, the state of the considered system denoted by $x \in \mathbb{R}^n$ depends on some design parameters denoted by $u \in \mathbb{R}^m$. One example for this setting forms the flow over an aircraft wing. Here, the shape of the wing that is defined by the design vector u determines the flow field x. The desired quasi-steady state x_* fulfills the fixed point equation

$$x_* = F(x_*, u) \tag{11}$$

for a given continuously differentiable function $F : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$. A fixed point property of this kind is also exploited by many other applications.

Assume that one can apply the iteration

$$x_{k+1} = F(x_k, u) \tag{12}$$

to obtain a linear converging sequence $\{x_k\}$ generated for any given control $u \in \mathbb{R}^n$. Then the limit point $x_* \in \mathbb{R}^n$ fulfills the fixed point equation (11). Moreover, suppose that $\|\frac{dF}{dx}(x_*, u)\| < 1$ holds for any pair (x_*, u) satisfying equation (11). Hence, there exists a differentiable function $\phi : \mathbb{R}^m \to \mathbb{R}^n$, such that $\phi(u) = F(\phi(u), u)$, where the state $\phi(u)$ is a fixed point of F according to a control u. To optimize the system described by the state vector $x = \phi(u)$ with respect to the design vector u, derivatives of ϕ with respect to u are of particular interest.

6.4.2   Forward and reverse derivative iterations

Applying the chain rule of differentiation to Equation (11) yields

$$\frac{d\phi}{du}(u) = \frac{dx_*}{du} = \frac{dF}{dx}(x_*, u)\frac{dx_*}{du} + \frac{dF}{du}(x_*, u). \tag{13}$$

This formula is a fixed point equation for the derivative $\frac{d\phi}{du}(u) = \frac{dx_*}{du}$. Analytically, the derivative $\frac{d\phi}{du}(u)$ can be expressed as

$$\frac{d\phi}{du}(u) = \left(I - \frac{dF}{dx}(\phi(u), u)\right)^{-1} \frac{dF}{du}(\phi(u), u). \tag{14}$$

However, a numerical computation according to this formula is not appropriate. Using Automatic Differentiation (AD), directional derivatives $\dot{\phi} = \frac{d\phi}{du}\dot{u} \in \mathbb{R}^n$ are evaluated for a given $\dot{u} \in \mathbb{R}^m$ with the forward mode. The reverse mode yields adjoints $\bar{u}^T = \bar{x}^T\frac{d\phi}{du} \in \mathbb{R}^m$

for a given $\dot{u} \in \mathbb{R}^m$. Denoting $\dot{x}_k = \frac{dx_k}{du}\dot{u}$ for $k = 0, 1, \ldots$ and applying the forward mode differentiation, we obtain a fixed-point iteration for Equation (13)

$$\dot{x}_{k1} = \frac{dF}{dx}(x_*, u)\dot{x}_k + \frac{dF}{du}(x_*, u)\dot{u} = F'(x_*, u)\begin{pmatrix} \dot{x}_k \\ \dot{u} \end{pmatrix}. \tag{15}$$

If the derivative $\frac{dF}{dx}(x_*, u)$ describes a contraction, this iteration converges to the unique fixed point $\dot{x}_* = \frac{d\phi(u)}{du}\dot{u}$. As analyzed already in [2], Equation (14) yields, for any given $\bar{x} \in \mathbb{R}^n$, the identity

$$\bar{x}^T \frac{dx_*}{du} = \bar{x}^T \left(I - \frac{dF}{dx}(x_*, u) \right)^{-1} \frac{dF}{du}(x_*, u).$$

Setting $\zeta^T := \bar{x}^T \left(I - \frac{dF}{dx}(x_*, u) \right)^{-1}$, we get the fixed point equation

$$\zeta_*^T = \zeta_*^T \frac{dF(x_*, u)}{dx} + \bar{x}^T$$

for the adjoint information. Hence, one obtains for $k = 0, 1, \ldots$ the iteration

$$(\zeta_{k+1}^T, \bar{u}_{k+1}^T) = \left(\zeta_k^T \frac{dF(x_*, u)}{dx} + \bar{x}^T, \ \zeta_k^T \frac{dF(x_*, u)}{du} \right)$$
$$= \zeta_k^T F'(x_*, u) + (\bar{x}^T, 0^T). \tag{16}$$

which converges to the unique fixed point consisting of $\zeta_*^T = \bar{x}^T \left(I - \frac{dF}{dx}(x_*, u) \right)^{-1}$ and $\bar{u}_* = \zeta_*^T \frac{dF}{du}(x_*, u) = \bar{x}^T \frac{dx_*}{du}$. As shown in [2], the uniform rate of convergence of the derivative fixed point iteration (16) equals the asymptotic rate of convergence of the original fixed point iteration (12).

Using not a black-box approach but a slightly advanced technique, AD provides for the fixed point iteration (12) given as a computer program the efficient evaluation of the terms $F'\dot{x}$ and $\zeta^T F'$ in equation (15) and (16)) respectively. For this purpose, we apply a two-phase approach, i.e., a splitting of the original iteration (12) and the derivative iteration (15) and (16) respectively. Then the iterations computing the derivatives use information of $F(x_*, u)$ at the previously computed fixed point x_*. This approach to compute adjoint information was proposed in [2] and applied for example in [6]. As alternative, the derivative information can be computed in combination with the original iteration to evaluate x_*. This is the so-called piggy-back approach [9].

6.4.3   Embedded fixed point iteration

Usually fixed point iterations are part of a larger computational context. This fact can be described in a mathematical way as a composition of mappings

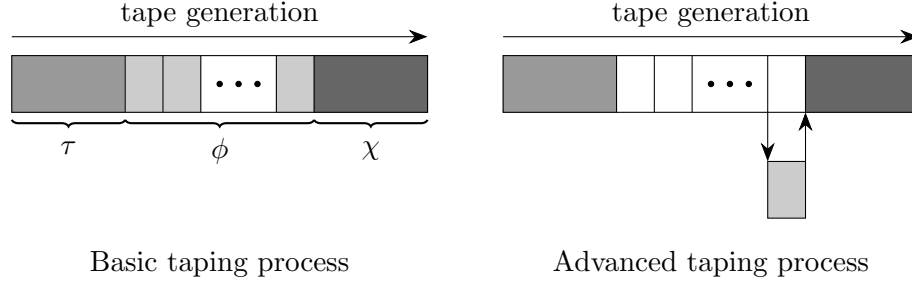$$\underbrace{p \xrightarrow{\tau} (x_0, u) \xrightarrow{\phi} (x_*, u) \xrightarrow{\chi} y}_{y=J(p)}, \tag{17}$$

Figure 5: Different taping approaches

where to p is a set of given parameters. We refer τ as *initialization* calculating the initial value x_0 of the fixed point iteration and the design vector u from the parameter values p. The function ϕ denotes the *fixed point iteration*, and χ represents the *evaluation* of a target function to compute the target value y. In the remainder of this subsection, we describe the application of the reverse mode of AD to the computational procedure (17). The forward mode can be derived analogously.

The differentiation of $J(p) = (\chi \circ \phi \circ \tau)(p)$ using the reverse mode of AD for a weight vector $\bar{y}$ yields the value $\bar{p} = \bar{y}^T \bar{J}(p)$. Its computation can be seen as a composition of the following functions:

$$\begin{aligned}
(\bar{x}_*, \bar{u}) &= \bar{y}^T \bar{\chi}(x_*, u), \\
\bar{u} &= \bar{u} + \bar{\phi}(u, \bar{x}_*), \\
\bar{p} &= \bar{\tau}(p, \bar{u}).
\end{aligned}$$

Applying ADOL-C in a "black-box" fashion to the computation of $J(p) = (\chi \circ \phi \circ \tau)(p)$ is not efficient since it neglects any structure of the fixed point iteration in ϕ. According to the evaluation of $\bar{\phi}(u, \bar{x}_*) = \bar{u}$ in (16), we only require the repeated evaluation of the terms $\zeta_k^T F'(x_*, u)$. Therefore the taping process is stopped during the evaluation of the fixed point iteration $\phi()$. An internal function representation of $F(x_*, u)$ is created only for the last iterate, which is assumed to be sufficiently close to x_*. This internal representation of one iteration of the fixed point process can be seen as a subtape. An illustration of this advanced taping approach in comparison to the basic taping, i.e., the black-box application of ADOL-C, is given in Figure 5. During the evaluation of the adjoint information $\bar{p} = \bar{y}^T \bar{J}(p)$ the tape is interpreted in the reverse order starting at the end. Exploiting the advanced taping mechanism, the original derivative computation is interrupted when the fixed point iteration is reached. The reverse mode differentiation of the fixed point iteration according to (16) is performed by repeatedly computing the reverse mode of $F(x_*, u)$ for different weight vectors ζ_k. After the computation of $\bar{u}$, the original derivative computation is started again to evaluate the overall derivative information $\bar{p}$. This nested tape evaluation is illustrated in Figure 6.
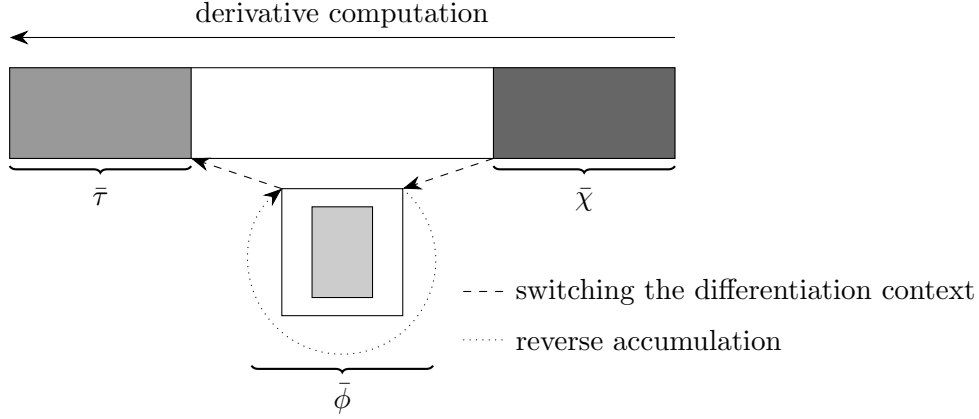
Figure 6: Derivative computation exploiting the fixed point structure

6.4.4   Implementation

To exploit the advanced algorithmic differentiation of such fixed point iterations ADOL-C provides the special functions `fp_iteration(...)`. It has the following interface:

```
int fp_iteration(sub_tape_num,double_F,adouble_F,norm,norm_deriv,epsilon,epsilon_deriv,
               N_max,N_max_deriv,x_0,u,x_fix,dim_x,dim_u)
int sub_tape_num;              // tape identification for sub_tape
int (*double_F)(double*, double* ,double*, int, int);
                               // pointer to a function that computes for x and u
                               // the value y = F(x, u) for double arguments
int (*adouble_F)(adouble*, adouble*, adouble*, int, int);
                               // pointer to a function that computes for x and u
                               // the value y = F(x, u) for adouble arguments
double (*norm)(double*, int);
                               // pointer to a function that computes
double (*norm_deriv)(double*, int);
                               // pointer to a function that computes
int *norm_deriv;               // pointer to a function that computes
                               // the norm of a vector
double epsilon;                // termination criterion for fixed point iteration
double epsilon_deriv;          // termination criterion for adjoint fixed point iteration
int N_max;                     // maximal number of iterations for state computation
int N_max_deriv;               // maximal number of iterations for adjoint computation
adouble *x_0;                  // initial state of fixed point iteration
adouble *u;                    // value of u
```

```
adouble *x_fic;           // final state of fixed point iteration
int dim_x;                // dimension of x
int dim_u;                // dimension of u
```

Here `sub_tape_num` is an ADOL-C identifier for the subtape that should be used for the
fixed point iteration. `double_F` and `adouble_F` are pointers to functions, that compute for
x and u a single iteration step $y = F(x, u)$. Thereby `double_F` uses `double` arguments and
`adouble_F` uses ADOL-C `adouble` arguments. The parameters `norm` and `norm_deriv` are
pointers to functions computing the norm of a vector. The latter functions together with
`eps`, `eps_deriv`, `N_max`, and `N_max_deriv` control the iterations. Thus the following loops
are performed:

$$
\begin{array}{ll}
\text{do} & \text{do} \\
\quad k = k + 1 & \quad k = k + 1 \\
\quad x = y & \quad \zeta = \xi \\
\quad y = F(x, u) & \quad (\xi^T, \bar{u}^T) = \zeta^T F'(x_*, u) + (\bar{x}^T, 0^T) \\
\text{while } \|y - x\| \geq \varepsilon \text{ and } k \leq N_{max} & \text{while } \|\xi - \zeta\|_{deriv} \geq \varepsilon_{deriv} \\
& \text{and } k \leq N_{max,deriv}
\end{array}
$$

The vector for the initial iterate and the control is stored in `x_0` and `u` respectively. The
vector in which the fixed point is stored is `x_fix`. Finally `dim_x` and `dim_u` represent the
dimensions n and m of the corresponding vectors.

The presented driver is prototyped in the header file `<adolc/fixpoint.h>`. This header
is included by the global header file `<adolc/adolc.h>` automatically. An example code
that shows also the expected signature of the function pointers is contained in the directory
`examples/additional_examples/fixpoint_exam`.

6.5 Advanced algorithmic differentiation of OpenMP parallel programs

ADOL-C allows to compute derivatives in parallel for functions containing OpenMP parallel
loops. This implies that an explicit loop-handling approach is applied. A typical situation
is shown in Figure 7, where the OpenMP-parallel loop is preceded by a serial startup
calculation and followed by a serial finalization phase.

Initialization of the OpenMP-parallel regions for ADOL-C is only a matter of adding
a macro to the outermost OpenMP statement. Two macros are available that only differ
in the way the global tape information is handled. Using `ADOLC_OPENMP`, this information,
including the values of the augmented variables, is always transferred from the serial to the
parallel region using *firstprivate* directives for initialization. For the special case of iterative
codes where parallel regions, working on the same data structures, are called repeatedly
the `ADOLC_OPENMP_NC` macro can be used. Then, the information transfer is performed only
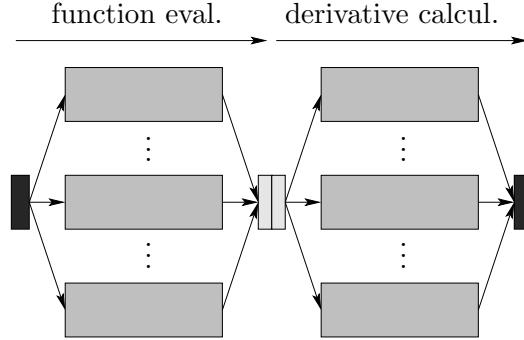
Figure 7: Basic layout of mixed function and the corresponding derivation process

once within the iterative process upon encounter of the first parallel region through use of the *threadprivate* feature of OpenMP that makes use of thread-local storage, i.e., global memory local to a thread. Due to the inserted macro, the OpenMP statement has the following structure:

> #pragma omp ... ADOLC_OPENMP   or
> #pragma omp ... ADOLC_OPENMP_NC

Inside the parallel region, separate tapes may then be created. Each single thread works in its own dedicated AD-environment, and all serial facilities of ADOL-C are applicable as usual. The global derivatives can be computed using the tapes created in the serial and parallel parts of the function evaluation, where user interaction is required for the correct derivative concatenation of the various tapes.

For the usage of the parallel facilities, the `configure`-command has to be used with the option `--with-openmp-flag=FLAG`, where `FLAG` stands for the system dependent OpenMP flag. The parallel differentiation of a parallel program is illustrated by the example program `openmp_exam.cpp` contained in `examples/additional_examples/openmp_exam`.

7   Traceless forward differentiation in ADOL-C

Up to version 1.9.0, the development of the ADOL-C software package was based on the decision to store all data necessary for derivative computation on traces (tapes), where large applications require the traces to be written out to corresponding files. In almost all cases this means a considerable drawback in terms of run time due to the excessive memory accesses. Using these traces enables ADOL-C to offer multiple functions. However, it is not necessary for all tasks of derivative computation to do that.

Starting with version 1.10.0, ADOL-C now features a traceless forward mode for computing first order derivatives in scalar mode, i.e., $\dot{y} = F'(x)\dot{x}$, and in vector mode, i.e.,

$\dot{Y} = F'(x)\dot{X}$. This traceless variant coexists with the more universal trace based mode in the package. The following subsections describe the source code modifications required to use the traceless forward mode of ADOL-C.

7.1 Modifying the Source Code

Let us consider the coordinate transformation from Cartesian to spherical polar coordinates given by the function $F : \mathbb{R}^3 \to \mathbb{R}^3$, $y = F(x)$, with

$$y_1 = \sqrt{x_1^2 + x_2^2 + x_3^2}, \qquad y_2 = \arctan\left(\sqrt{x_1^2 + x_2^2}/x_3\right), \qquad y_3 = \arctan(x_2/x_1),$$

as an example. The corresponding source code is shown in Figure 8. Changes to the source

```
#include <iostream>
using namespace std;

int main() {
    double x[3], y[3];

    for (int i=0; i<3; ++i)                    // Initialize xᵢ
        ...

    y[0] = sqrt(x[0]*x[0]+x[1]*x[1]+x[2]*x[2]);
    y[1] = atan(sqrt(x[0]*x[0]+x[1]*x[1])/x[2]);
    y[2] = atan(x[1]/x[0]);

    cout << "y1=" << y[0] << " , y2=" << y[1] << " , y3=" << y[2] << endl;

    return 0;
}
```

Figure 8: Example for traceless forward mode

code that are necessary for applying the traceless forward mode of ADOL-C are described in the following two subsections, where the vector mode version is described as extension of the scalar mode.

The scalar mode

To use the traceless forward mode, one has to include only the header file `adtl.h` since it does not include the trace based functions defined in other header files.

As in the trace based forward version of ADOL-C all derivative calculations are introduced by calls to overloaded operators. Therefore, similar to the trace-based version all independent, intermediate and dependent variables must be declared with type adouble. The whole traceless functionality provided by `adtl.h` was written as complete inline intended code due to run time aspects, where the real portion of inlined code can be influenced by switches for many compilers. Likely, the whole derivative code is inlined by default. Our experiments with the traceless mode have produced complete inlined code by using standard switches (optimization) for GNU and Intel C++ compiler.

To avoid name conflicts resulting from the inlining the traceless version has its own namespace `adtl`. As a result four possibilities of using the adouble type are available for the traceless version:

- Defining a new type

```
typedef adtl::adouble adouble;
...
adouble tmp;
```

  This is the preferred way. Remember, you can not write an own adouble type/class with different meaning after doing the typedef.

- Declaring with namespace prefix

```
adtl::adouble tmp;
```

  Not the most handsome and efficient way with respect to coding but without any doubt one of the safest ways. The identifier adouble is still available for user types/classes.

- Trusting macros

```
#define adouble adtl::adouble
...
adouble tmp;
```

  This approach should be used with care, since standard defines are text replacements.

- Using the complete namespace

```
using namespace adtl;
...
adouble tmp;
```

  A very clear approach with the disadvantage of uncovering all the hidden secrets. Name conflicts may arise!

After defining the variables only two things are left to do. First one needs to initialize the values of the independent variables for the function evaluation. This can be done by assigning the variables a double value. The ad-value is set to zero in this case. Additionally, the traceless forward mode variant of ADOL-C offers a function named setValue for setting the value without changing the ad-value. To set the ad-values of the independent variables ADOL-C offers two possibilities:

- Using the constructor

  ```
  double seedOne = 1., seedZero = 0.;
  adouble x1(2,&seedOne), x2(4,&seedZero), y;
  ```

  This would create three adoubles x_1, x_2 and y. Obviously, the latter remains uninitialized. In terms of function evaluation x_1 holds the value 2 and x_2 the value 4 whereas the derivative values are initialized to $\dot{x}_1 = 1$ and $\dot{x}_2 = 0$.

- Setting point values directly

  ```
  double seedOne = 1., seedZero = 0.;
  adouble x1=2, x2=4, y;
  ...
  x1.setADValue(&seedOne);
  x2.setADValue(&seedZero);
  ```

  The same example as above but now using setADValue-method for initializing the derivative values. It is important to note that in both cases the derivative values have to be set by passing an address of a variable or a pointer of the double type (instead of passing the value directly). The reason for doing this will become clear in the next subsection.

The derivatives can be obtained at any time during the evaluation process by calling the getADValue-method which returns a pointer (instead of a value), and therefore, it can be assigned to another pointer or it can be dereferenced immediately in order to read the value:

```
adouble y;
...
cout << *(y.getADValue());
```

Figure 9 shows the resulting source code incorporating all required changes for the example given above.

```
#include <iostream>
using namespace std;

#include <adolc/adtl.h>
typedef adtl::adouble adouble;

int main() {
    adouble x[3], y[3];

    for (int i=0; i<3; ++i)                        // Initialize x_i
        ...

    double seed = 1.;
    x[0].setADValue(&seed);                        // derivative of f with respect to x_1
    y[0] = sqrt(x[0]*x[0]+x[1]*x[1]+x[2]*x[2]);
    y[1] = atan(sqrt(x[0]*x[0]+x[1]*x[1])/x[2]);
    y[2] = atan(x[1]/x[0]);

    cout << "y1=" << y[0].getValue() << " , y2=" << y[1].getValue ... ;
    cout << "dy2/dx1 = " << *(y[1].getADValue()) << endl;
    return 0;
}
```

Figure 9: Example for traceless scalar forward mode

The vector mode

In scalar mode only one direction element has to be stored per adouble whereas a field of p elements is needed in the vector mode to cover the computations for the given p directions. The resulting changes to the source code are described in this section.

In order to use the vector mode one has to call adtl::setNumDir(N). This function takes the maximal number of directions to be used within the resulting vector mode. Changing the number of directions in the middle of executing code containing active variables is undesirable and can lead to segmentation violations, hence such an operation will result in an error. adtl::setNumDir(N) must therefore be called before declaring any adtl::adouble objects.

Setting and getting the derivative values is done in the same manner as in the scalar case, by passing and retrieving the pointers, as illustrated in the following example:

```
adtl::setNumDir(10);
adouble x, y;
double *ptr1 = new double[10];
double *ptr2;
...
x1=2;
x1.setADValue(ptr1);
...
ptr2=y.getADValue();
```

Now using the pointers in the methods setADValue and getADValue makes more sense since the vector mode requires the field of p elements per adouble which holds the derivative values. Additionally, the traceless vector forward mode of ADOL-C offers two new methods for setting/getting the derivative values. Setting a derivative value of the desired vector element is possible by passing a double value and its position in the vector (integer). Getting a derivative value of the desired vector element requires as argument only its position in the vector. One can note that the pointers are not used in these cases, as illustrated in the following example:

```
adtl::setNumDir(10);
...
adouble x, y;
...
x1=2;
x1.setADValue(5,1);                  // set the 6th point value of x to 1.0
...
cout << y.getADValue(3) << endl;     // print the 4th derivative value of y
```

The resulting source code containing all changes that are required is shown in Figure 10

7.2   Compiling and Linking the Source Code

After incorporating the required changes, one has to compile the source code and link the object files to get the executable. As long as the ADOL-C header files are not included in the absolute path the compile sequence should be similar to the following example:

g++ -I/home/username/adolc_base/include -c traceless_scalar.cpp

The `-I` option tells the compiler where to search for the ADOL-C header files. This option can be omitted when the headers are included with absolute path or if ADOL-C is installed in a "global" directory.

```cpp
#include <iostream>
using namespace std;

#include <adolc/adtl.h>
typedef adtl::adouble adouble;

int main() {
    adtl::setNumDir(3);
    adouble x[3], y[3];

    for (int i=0; i<3; ++i) {
        ...                            // Initialize x_i
        for (int j=0; j<3; ++j) if (i==j) x[i].setADValue(j,1);
    }

    y[0] = sqrt(x[0]*x[0]+x[1]*x[1]+x[2]*x[2]);
    y[1] = atan(sqrt(x[0]*x[0]+x[1]*x[1])/x[2]);
    y[2] = atan(x[1]/x[0]);

    cout << "y1=" << y[0].getValue() << " , y2=" << y[1].getValue ... ;
    cout << "jacobian : " << endl;
    for (int i=0; i<3; ++i) {
        for (int j=0; j<3; ++j)
            cout << y[i].getADValue(j) << " ";
        cout << endl;
    }
    return 0;
}
```

Figure 10: Example for traceless vector forward mode

Although originally designed to be completely inline, certain global variables need to be initialized in the compiled part of the library and therefore the ADOL-C library must be linked with the compiled code. The example started above could be finished with the following command:

```
g++ -o traceless_scalar traceless_scalar.o
-Wl,--rpath -Wl,/home/username/adolc_base/lib -L/home/username/adolc_base/lib -ladolc
```

The mentioned source codes traceless_scalar.c and traceless_vector.c illustrating the use of the for traceless scalar and vector mode can be found in the directory examples.

7.3   Concluding Remarks for the Traceless Forward Mode Variant

As many other AD methods the traceless forward mode provided by the ADOL-C package has its own strengths and drawbacks. Please read the following section carefully to become familiar with the things that can occur:

- Advantages:
  - Code speed
    Increasing computation speed was one of the main aspects in writing the traceless code. In many cases higher performance can be expected this way. However this is not guaranteed for repeated evaluations.
  - Smaller overall memory requirements
    Traceless ADOL-C does not write traces anymore, as the name implies. Loop "unrolling" can be avoided this way. Considered main memory plus disk space as overall memory requirements the traceless version can be executed in a more efficient way.
- Drawbacks:
  - Main memory limitations
    The ability to compute derivatives to a given function is bounded by the main memory plus swap size when using traceless ADOL-C. Computation from swap should be avoided anyway as far as possible since it slows down the computing time drastically. Therefore, if the program execution is terminated without error message insufficient memory size can be the reason among other things. The memory requirements M can be determined roughly as followed:
    * Scalar mode: $M =$(number of adoubles)$*2 + M_p$
    * Vector mode: $M =$(number of adoubles)*(NUMBER_DIRECTIONS$+1) + M_p$

    where the storage size of all non adouble based variables is described by M_p.
  - Compile time
    As discussed in the previous sections, the traceless forward mode of the ADOL-C package is implemented as inline intended version. Using this approach results in a higher source code size, since every operation involving at least one adouble stands for the operation itself as well as for the corresponding derivative code after the inlining process. Therefore, the compilation time needed for the traceless version may be higher than that of the trace based code.
  - Code Size
    A second drawback and result of the code inlining is the increase of code sizes for the binaries. The increase factor compared to the corresponding trace based program is difficult to quantify as it is task dependent. Practical results have shown that values between 1.0 and 2.0 can be expected. Factors higher than 2.0 are possible too and even values below 1.0 have been observed.

8   Traceless forward differentiation in ADOL-C using Cuda

One major drawback using the traceless version of ADOL-C is the fact that several function evaluations are needed to compute derivatives in many different points. More precisely, to calculate the Jacobian for a function $F : \mathbb{R}^n \to \mathbb{R}^m$ in M points, M function evaluations are needed for the traceless vector mode and even $M * n$ for the traceless scalar mode. Depending on the size of the function this can result in a long run time. To achieve a better performance one can use parallelisation techniques as the same operations are performed during a function evaluation. One possibility is to use GPUs since they are optimized for data parallel computation. Starting with version 2.3.0 ADOL-C now features a traceless forward mode for computing first order derivatives in scalar mode on GPUs using the general purpose parallel computing architecture Cuda.

The idea is to include parallel code that executes in many GPU threads across processing elements. This can be done by using kernel functions, that is functions which are executed on GPU as an array of threads in parallel. In general all threads execute the same code. They are grouped into blocks which are then grouped into grids. A kernel is executed as a grid of blocks of threads. For more details see, e.g., the NVIDIA CUDA C Programming Guide which can be downloaded from the web page www.nvidia.com. To solve the problem of calculating the Jacobian of F at M points it is possible to let each thread perform a function evaluation and thus the computation of derivatives for one direction at one point. The advantage is that the function is evaluated in different points in parallel which can result in a faster wall-clock run time.

The following subsection describes the source code modifications required to use the traceless forward mode of ADOL-C with Cuda.

8.1   Modifying the source code

Let us again consider the coordinate transformation from Cartesian to spherical polar coordinates given by the function $F : \mathbb{R}^3 \to \mathbb{R}^3$, $y = F(x)$, with

$$y_1 = \sqrt{x_1^2 + x_2^2 + x_3^2}, \qquad y_2 = \arctan\left(\sqrt{x_1^2 + x_2^2}/x_3\right), \qquad y_3 = \arctan(x_2/x_1),$$

as an example. We now calculate the Jacobian at $M = 1024$ different points. The source code for one point is shown in Figure 8. This example has no real application but can still be used to show the combination of the traceless version of ADOL-C and Cuda.

For the use of this mode a Cuda toolkit, which is suitable for the graphic card used, has to be installed. Furthermore, it is important to check that the graphic card used supports double precision (for details see e.g. NVIDIA CUDA C Programming Guide). Otherwise the data type employed inside of the `adouble` class has to be adapted to `float`. To use the traceless forward mode with Cuda, one has to include the header files `adoublecuda.h`

and `cuda.h`. The first one contains the definition of the class `adouble` and the overloaded operators for this version of ADOL-C. As in the other versions all derivative calculations are introduced by calls to overloaded operators. The second header file is needed for the use of Cuda.

One possibility to solve the problem above is the following. First of all three `double` arrays are needed: `x` for the independent variables, `y` for the dependent variables and `deriv` for the values of the Jacobian matrices. The independent variables have to be initialized, therefore the points at which the function should be evaluated are saved in a row in the same array of length $3 * M$. For the computation on GPUs one also has to allocate memory on this device. Using the syntax in Cuda one can allocate an array of length $3 * M$ (number of independent variables times number of points) for the independent variables as follows:

```
double * devx;
cudaMalloc((void**)&devx, 3*M*sizeof(double));
```

The arrays for the dependent variables and the values of the Jacobian matrices are allocated in the same way. Then the values of the independent variables have to be copied to the GPU using the following command

```
cudaMemcpy(devx, x, sizeof(double)*3*M, cudaMemcpyHostToDevice);
```

The argument `cudaMemcpyHostToDevice` indicates that the values are copied from the host to the GPU. In this case the values stored in `x` are copied to `devx`.

Now all required information has been transferred to the GPU. The changes in the source code made so far are summarized in Figure 11.

In the next step the user has to specify how many blocks and threads per block will be needed for the function evaluations. In the present example this is done by the call of the function `kernellaunch`, see Figure 12. In this case the blocks are two dimensional: the x-dimension is determined by the number of points M at which the Jacobian matrix has to be calculated while the y-dimension is given by the number of independent variables, i.e., 3. Since a block cannot contain more than 1024 threads, the x-dimension in the example is 64 instead of $M = 1024$. Therefore $1024/64 = 16$ blocks are needed. The described division into blocks is reasonable as each thread has to perform a function evaluation for one point and one direction, hence $M * 3$ threads are needed where 3 denotes the number of independent variables corresponding to the number of directions needed for the computation of the Jacobian in one point.

We can now perform the function evaluations together with the calculation of the Jacobians. The corresponding code is illustrated in Figure 13. Since the function evaluations should be performed on a GPU a kernel function is needed which is defined by using the `__global__` declaration specifier (see Figure 13). Then each thread executes the operations that are defined in the kernel.

```
#include <iostream>
#include <cuda.h>
#include <adoublecuda.h>
using namespace std;

int main() {
    int M=1024;
    double* deriv = new double[9*M];
    double* y = new double[3*M];
    double* x = new double[3*M];

    // Initialize x_i
    for (int k=0; k<M; ++k){
        for (int i=0; i<3; ++i)
            x[k*3+i] =i + 1/(k+1);}

    // Allocate array for independent and dependent variables
    // and Jacobian matrices on GPU
    double * devx;
    cudaMalloc((void**)&devx, 3*M*sizeof(double));
    double * devy;
    cudaMalloc((void**)&devy, 3*M*sizeof(double));
    double * devderiv;
    cudaMalloc((void**)&devderiv, 3*3*M*sizeof(double));

    // Copy values of independent variables from host to GPU
    cudaMemcpy(devx, x, sizeof(double)*3*M, cudaMemcpyHostToDevice);

    // Call function to specify amount of blocks and threads to be used
    kernellaunch(devx, devy, devderiv,M);

    // Copy values of dependent variables and Jacobian matrices from GPU to host
    cudaMemcpy(y, devy, sizeof(double)*3*M,cudaMemcpyDeviceToHost);
    cudaMemcpy(deriv, devderiv, sizeof(double)*3*3*M, cudaMemcpyDeviceToHost);
}
```

Figure 11: Example for traceless scalar forward mode with Cuda

In this example each thread is assigned the task of calculating the derivatives in one point with respect to one independent variable. Therefore some indices are needed for the

```
cudaError_t kernellaunch(double* inx, double* outy, double* outderiv, int M) {
    // Create 16 blocks
    int Blocks=16;
    // Two dimensional (M/Blocks)×3 blocks
    dim3 threadsPerBlock(M/Blocks,3);

    // Call kernel function with 16 blocks with (M/Blocks)×3 threads per block
    kernel <<< Blocks, threadsPerBlock >>>(inx, outy, outderiv);
    cudaError_t cudaErr = cudaGetLastError();

    return cudaErr;
}
```

Figure 12: Example for traceless scalar forward mode with Cuda

implementation. Each thread has a unique thread ID in a block consisting of an x and an y-dimension. The blocks have an ID as well. In the example the following indices are used.

- index = threadIdx.x denotes the x-dimension of a thread (ranges from 0 to 63 in the example)

- index1 = threadIdx.y denotes the y-dimension of a thread (ranges from 0 to 2 in the example)

- index2 = blockIdx.x denotes the block index (ranges from 0 to 15 in the example)

- index3 = blockDim.x denotes the x-dimension of a block (always 64 in the example)

- index4 = blockDim.x*blockDim.y denotes the size of a block (always $64 * 3 = 192$ in the example)

For the calculation of derivatives the function evaluation has to be performed with adoubles. Therefore the dependent and the independent variables have to be declared as adoubles as in the other versions of ADOL-C (see, e.g., Section 7). Similar to the traceless version without Cuda the namespace adtlc is used. Now each thread has to read out the point at which the function evaluation is then performed. This is determined by the blockindex and the x-dimension of the thread, therefore the independent variables in a thread have the values

$$x[i]=inx[index2*index4+index*3+i];$$

```
__global__ void kernel(double* inx, double* outy, double* outderiv) {
    const int index = threadIdx.x ;
    const int index1 = threadIdx.y;
    const int index2 = blockIdx.x;
    const int index3 = blockDim.x;
    const int index4 = blockDim.x*blockDim.y;

    // Declare dependent and independent variables as adoubles
    adtlc::adouble y[3], x[3];
    // Read out point for function evaluation
    for(int i=0; i < 3; i++)
        x[i]=inx[index2*index4+index*3+i];
    // Set direction for calculation of derivatives
    x[index1].setADValue(1);

    // Function evaluation
    y[0] = sqrt(x[0]*x[0]+x[1]*x[1]+x[2]*x[2]);
    y[1] = atan(sqrt(x[0]*x[0]+x[1]*x[1])/x[2]);
    y[2] = atan(x[1]/x[0]);

    for(int i=0; i < 3; i++)
        outy[(index2*index3+index)*3+i]=y[i].getValue();
    for(int i=0; i < 3; i++)
        outderiv[(index2*index4+index*3+index1)*3+i]=y[i].getADValue();
}
```

Figure 13: Example for traceless scalar forward mode with Cuda

for i=0,1,2 where inx corresponds to the vector devx. The direction for the derivatives is given by index1.

$$x[index1].setADValue(1);$$

The functions for setting and getting the value and the derivative value of an adouble are the same as in the traceless version of ADOL-C for first order derivatives (see Section 7). The function evaluation is then performed with adouble x on GPU and the results are saved in adouble y. The function evaluation itself remains unchanged (see Figure 13). Then we store the values of the function evaluations in each point in a row in one array:

$$outy[(index2*index3+index)*3+i]=y[i].getValue();$$

for i=0,1,2. The values of a Jacobian are stored column by column in a row:

$$\text{outderiv}[(\text{index2*index4}+\text{index*3}+\text{index1})\text{*3}+\text{i}]=\text{y[i].getADValue();}$$

for i=0,1,2.

   Now there is one thing left to do. The values calculated on the GPU have to be copied back to host. For the dependent variables in the example this can be done by the following call:

$$\text{cudaMemcpy(y, devy, sizeof(double)*3*M,cudaMemcpyDeviceToHost);}$$

see Figure 11. The argument cudaMemcpyDeviceToHost determines that the values are copied from GPU to host.

8.2   Compiling and Linking the Source Code

After incorporating the required changes, one has to compile the source code and link the object files to get the executable. For the compilation of a Cuda file the nvcc compiler is needed. The compile sequence should be similar to the following example:

$$\text{nvcc -arch=sm\_20 -o traceless\_cuda traceless\_cuda.cu}$$

The compiler option `-arch=sm_20` specifies the compute capability that is assumed, in this case one that supports double precision.

   The mentioned source code traceless_cuda.cu illustrating the use of the forward traceless scalar mode with Cuda and a further example liborgpu.cu can be found in the directory examples. The second example is an adaption of the OpenMP example to the traceless version with Cuda.

9   Installing and Using ADOL-C

9.1   Generating the ADOL-C Library

The currently built system is best summarized by the ubiquitous gnu install triplet

```
configure - make - make install .
```

Executing   this   three   steps   from   the   package   base   directory
`</SOMEPATH/adolc-2.7.3-trunk>` will compile the static and the dynamic ADOL-C

library with default options and install the package (libraries and headers) into the default installation directory `<$HOME/adolc_base>`. Inside the install directory the subdirectory `include` will contain all the installed header files that may be included by the user program, the subdirectory `lib` will contain the 32-bit compiled library and the subdirectory `lib64` will contain the 64-bit compiled library. Depending on the compiler only one of `lib` or `lib64` may be created.

Before doing so the user may modify the header file `usrparms.h` in order to tailor the ADOL-C package to the needs in the particular system environment as discussed in Section 2.2. The configure procedure which creates the necessary `Makefile`s can be customized by use of some switches. Available options and their meaning can be obtained by executing `./configure --help` from the package base directory.

All object files and other intermediately generated files can be removed by the call `make clean`. Uninstalling ADOL-C by executing `make uninstall` is only reasonable after a previous called `make install` and will remove all installed package files but will leave the created directories behind.

The sparse drivers are included in the ADOL-C libraries if the `-DENABLE\_SPARSE1=` command. The ColPack library is either automatically downloaded or can be provided specifying `-DColPack\_DIR`.

The option `--disable-stdczero` turns off the initialization in the `adouble` default constructor. This will improve efficiency but requires that there be no implicit array initialization in the code, see Section 2.3.

Support for MPI and the AdjoinableMPI API (see `http://www.mcs.anl.gov/~utke/AdjoinableMPI/AdjoinableMPIDox/index.html`) may be enabled using the option `--enable-ampi`. This requires the presence of MPI compiler wrappers `mpicc` and `mpicxx` in the `$PATH` and the AdjoinableMPI libraries in the standard locations. If MPI is installed in a nonstandard path one may specify this using the `--with-mpi=PATH` option. Similarly if the AdjoinableMPI libraries are in an nonstandard path this may be specified using the `--with-ampi=PATH` option. When MPI and AdjoinableMPI support is compiled into ADOL-C the generated library will be unsuitable for linking with non-MPI programs and is called `libadolc_ampi`. Therefore the user programs should use the flag `-ladolc_ampi` instead of `-ladolc` in this case. An advanced user may in fact specify whatever name the resulting ADOL-C library should have using the `--with-soname=SONAME` option.

9.2   Compiling and Linking the Example Programs

The installation procedure described in Section 9.1 also provides the `Makefile`s to compile the example programs in the directories `<adolc-2.7.3-trunk>/ADOL-C/examples` and the additional examples in `<adolc-2.7.3-trunk>/ADOL-C/examples/additional_examples`. However, one has to execute the `configure` command with appropriate options for the

ADOL-C package to enable the compilation of examples. Available options are:

`--enable-docexa`	build all examples discussed in this manual (compare Section 10)
`--enable-addexa`	build all additional examples (See file `README` in the various subdirectories)

Just calling `make` from the packages base directory generates all configured examples and the library if necessary. Compiling from subdirectory `examples` or one of its subfolders is possible too. At least one kind of the ADOL-C library (static or shared) must have been built previously in that case. Hence, building the library is always the first step.

For Compiling the library and the documented examples on Windows using Visual Studio please refer to the `<Readme_VC++.txt>` files in the `<windows/>`, `<ThirdParty/ColPack/>` and `<ADOL-C/examples/>` subdirectories.

9.3 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 2 the provided *global* header file `<adolc/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.

`<adolc/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.
`<adolc/internal/usrparms.h>`	→ user customization of ADOL-C package (see Section 2.2); • after a change of user options the ADOL-C library `libadolc.*` has to be rebuilt (see Section 9.1); • is included by all ADOL-C header files and thus by all user programs.

Table 2: 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 respectively needed header files are indicated throughout the manual. Existing application determined dependencies 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 Table 3 and Table 4.

`<adolc/adtb_types.h>`	→ provides the interface to the basic active scalar data type of ADOL-C: class adouble (see Section 1);
`<adolc/taputil.h>`	→ provides functions to start/stop the tracing of active sections (see Section 1.3) as well as utilities to obtain tape statistics (see Section 2.1); • is included by the header `<adolc/adtb_types.h>`.

Table 3: Important header files: tracing/taping

9.4   Compiling and Linking C/C++ Programs

To compile a C/C++ program or single module using ADOL-C data types and routines one has to ensure that all necessary header files according to Section 9.3 are included. All modules involving *active* data types as adouble 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 `libadolc.*` must be used (see Section 9.1).

9.5   Interfacing with Python and R

For generating the SWIG based interface to Python and R programming languages the prerequisite is an installation of the SWIG tool (at least newer than version 3.0.8). Furthermore, for installing and using the R interface please refer to the documentation and software at `https://github.com/sriharikrishna/radolc`. Installation of the Python interface can be done by changing into the `ADOL-C/swig` subdirectory of the ADOL-C source code and calling the command

```
python setup.py install --user
```

This will internally cause the compilation and installation of the ADOL-C library and header files using default configuration options as described in Section 9.1.

For a description on the working of these interfaces refer to the following article:

K. Kulshreshtha et al. "Efficient computation of derivatives for solving optimization problems in R and Python using SWIG generated interfaces to ADOL-C". in: *Optimisation Methods and Software* 33.4–6 (2018), pp. 1173–1191

`<adolc/interfaces.h>`	→	provides interfaces to the **forward** and **reverse** routines as basic versions of derivative evaluation (see Section 5);
	●	comprises C++, C, and Fortran-callable versions;
	●	includes the header `<adolc/sparse/sparsedrivers.h>`;
	●	is included by the header `<adolc/drivers/odedrivers.h>`.
`<adolc/drivers.h>`	→	provides "easy to use" drivers for solving optimization problems and nonlinear equations (see Section 3.1);
	●	comprises C and Fortran-callable versions.
`<adolc/sparse/` `  sparsedrivers.h>`	→	provides the "easy to use" sparse drivers to exploit the sparsity structure of Jacobians (see Section 3.3);
	→	provides interfaces to C++-callable versions of **forward** and **reverse** routines propagating bit patterns (see Section 5.3);
	●	is included by the header `<adolc/interfaces.h>`.
`<adolc/sparse/` `  sparse_fo_rev.h>`	→	provides interfaces to the underlying C-callable versions of **forward** and **reverse** routines propagating bit patterns.
`<adolc/drivers/` `  odedrivers.h>`	→	provides "easy to use" drivers for numerical solution of ordinary differential equations (see Section 3.2);
	●	comprises C++, C, and Fortran-callable versions;
	●	includes the header `<adolc/interfaces.h>`.
`<adolc/drivers/` `  taylor.h>`	→	provides "easy to use" drivers for evaluation of higher order derivative tensors (see Section 3.4) and inverse/implicit function differentiation (see Section 3.5);
	●	comprises C++ and C-callable versions.
`<adolc/adalloc.h>`	→	provides C++ and C functions for allocation of vectors, matrices and three dimensional arrays of **doubles**.

Table 4: Important header files: evaluation of derivatives

9.6   Adding Quadratures as Special Functions

Suppose an integral

$$f(x) = \int\limits_{0}^{x} g(t)dt$$

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.cpp>`, include the full code defining
   double myintegral(double& x), and add the line

   $$\text{extend\_quad(myintegral, integrand);}$$

   This macro is extended to the definition of adouble myintegral(adouble& arg). Then remake the library `libadolc.*` (see Section 9.1).

2. In the definition of the class ADOLC_DLL_EXPORT adouble in `<adolc/adtb_types.h>`, add the statement

   $$\text{friend adouble myintegral(adouble\&).}$$

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

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

can be added to the file `<adouble.cpp>`. 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.cpp>`, so that one may also introduce iterated integrals.

10   Example Codes

The following listings are all simplified versions of codes that are contained in the example subdirectory `<adolc-2.7.3-trunk>/ADOL-C/examples` of ADOL-C. In particular, we have left out timings, which are included in the complete codes.

10.1   Speelpenning's Example (`speelpenning.cpp`)

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 <adolc/adtb_types.h>              // use of active doubles and taping
#include <adolc/drivers/drivers.h>      // use of "Easy to Use" drivers
                                    // gradient(.) and hessian(.)
#include <adolc/taping.h>                 // use of taping
...
void main() {
int n,i,j;
size_t tape_stats[STAT_SIZE];
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];
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]; y *= x[i]; }
  y >>= yp;
  delete[] x;
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];
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]); } }
cout << yp-1/(1.0+n) << " error in function \n";
cout << errg <<" error in gradient \n";
cout << errh <<" consistency check \n";
}                                   // end main
```

10.2   Power Example (`powexam.cpp`)

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/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 doubles including the return variable to adoubles. The new version can now be called from within any active section, as in the following main program.

```
#include ...                        // as above
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] << " - " << y.value() << " = " << Y[0][i]-y.value()
    << " (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.

10.3 Determinant Example (`detexam.cpp`)

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 <adolc/adtb_types.h>                    // use of active doubles and taping
#include <adolc/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 sub-matrix
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 += A[i][i].value();         // value() 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<<sum.value()<<" "; }               // 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.

10.4 Ordinary Differential Equation Example (odexam.cpp)

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 [10]. The following source code computes the corresponding values of $y' \in \mathbb{R}^3$:

```
#include <adolc/adtb_types.h>                    // use of active doubles and taping
#include <adolc/drivers/odedrivers.h>      // use of "Easy To use" ODE drivers
#include <adolc/adalloc.h>                       // use of ADOL-C allocation utilities

void tracerhs(short int tag, double* py, double* pyprime) {
adouble y[3];                                 // this time we left the parameters
adouble yprime[3];                            // passive and use the vector types
trace_on(tag);
for (int i=0; i<3; i++)
     y[i] <<= py[i];                          // 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();
}                                             // 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

$$
\begin{array}{rrr}
3 & -1 & 4 \\
1 & 2 & 2 \\
3 & 2 & 4
\end{array}
$$

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

Acknowledgements

References

[1] Christian H. Bischof, Peyvand M. Khademi, Ali Bouaricha, and Alan Carle. "Efficient computation of gradients and Jacobians by dynamic exploitation of sparsity in

automatic differentiation". In: *Optimization Methods and Software* 7.1 (1996), pp. 1–39.

[2]   B. Christianson. "Reverse accumulation and attractive fixed points". In: *Optimization Methods and Software* 3 (1994), pp. 311–326.

[3]   Bruce Christianson. "Reverse accumulation and accurate rounding error estimates for Taylor series". In: *Optimization Methods and Software* 1 (1992), pp. 81–94.

[4]   Assefaw Gebremedhin, Fredrik Manne, and Alex Pothen. "What color is your Jacobian? Graph coloring for computing derivatives". In: *SIAM Review* 47.4 (2005), pp. 629–705.

[5]   Assefaw Gebremedhin, Arijit Tarafdar, Fredrik Manne, and Alex Pothen. "New Acyclic and Star Coloring Algorithms with Applications to Hessian Computation". In: *SIAM Journal on Scientific Computing* 29.3 (2007), pp. 1042–1072.

[6]   R. Giering, T. Kaminski, and T. Slawig. "Applying TAF to a Navier-Stokes solver that simulates an Euler flow around an airfoil". In: *to appear in Future Generation Computer Systems* (2003).

[7]   Andreas Griewank. "On stable piecewise linearization and generalized algorithmic differentiation". In: *Optimization Methods and Software* 28.6 (2013), pp. 1139–1178.

[8]   Andreas Griewank, Jean Utke, and Andrea Walther. "Evaluating higher derivative tensors by forward propagation of univariate Taylor series". In: *Mathematics of Computation* 69 (2000), pp. 1117–1130.

[9]   Andreas Griewank and Andrea Walther. *Evaluating Derivatives, Principles and Techniques of Algorithmic Differentiation*. Second edition. SIAM, 2008.

[10] Ernst Hairer and Gerhard Wanner. *Solving Ordinary Differential Equations II*. Springer, 1991.

[11] R. Heinrich. "Implementation and Usage of Structured Algorithms within an Unstructured CFD-Code". In: *Notes on Numerical Fluid Mechanics and Multidisciplinary Design* 92 (2006).

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

[13] K. Kulshreshtha, S.H.K. Narayanan, J. Bessac, and K. MacIntyre. "Efficient computation of derivatives for solving optimization problems in R and Python using SWIG generated interfaces to ADOL-C". In: *Optimisation Methods and Software* 33.4–6 (2018), pp. 1173–1191.

[14] Klaus Röbenack. "Computation of Lie Derivatives of Tensor Fields Required for Nonlinear Controller and Observer Design Employing Automatic Differentiation". In: *PAMM* 5.1 (2005), pp. 181–184.

[15] Klaus Röbenack, Jan Winkler, and Siqian Wang. "LIEDRIVERS - A Toolbox for the Efficient Computation of Lie Derivatives Based on the Object-Oriented Algorithmic Differentiation Package ADOL-C". In: *Proc. of the 4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. Ed. by F. E. Cellier, D. Broman, P. Fritzson, and E. A. Lee. Vol. 56. Linköping Electronic Conference Proceedings. Zürich, 2011, pp. 57–66.

[16] Sebastian Schlenkrich, Andrea Walther, Nicolas R. Gauger, and Ralf Heinrich. "Differentiating Fixed Point Iterations with ADOL-C: Gradient Calculation for Fluid Dynamics". In: *Modeling, Simulation and Optimization of Complex Processes*. Ed. by Bock H.G., Kostina E., Phu H.X., and Rannacher R. Springer, 2008, pp. 499–508. DOI: 10.1007/978-3-540-79409-7_36.