ADOL–F
Automatic Differentiation of Fortran Codes *

D. Shiriaev† A. Griewank‡

Abstract

The package ADOL–F described here is a Fortran 90 interface to the ADOL–C library and facilitates the evaluation of first and higher derivatives of vector functions that are defined by computer programs written in Fortran 90/77.

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

1 Introduction

Automatic differentiation [4, 9] is a non-approximative method just like symbolic differentiation and allows fast and exact (up to machine precision) evaluation of derivatives of any degree. Especially in applications where second- and higher-order derivatives are needed, the ability of automatic differentiation to avoid truncation errors plays a major role.

In contrast the rounding errors in the numerator of divided differences, are amplified by the small denominator, which often results in poor approximations for second derivatives and makes them practically useless for third and higher derivatives. We obtain numerical values of higher-order derivatives of a given function without generating a formula for them, thus avoiding the unnecessary overhead of symbolic differentiation and the truncation errors caused by divided differences.

There are two basic modes of automatic differentiation which are usually referred to as "forward" and "reverse", respectively. The main difference between them is that the complexity of the forward mode (which is easier to implement) depends essentially on the number of independent variables, whereas the complexity of the reverse mode depends essentially on the number of dependent variables. Hence, the forward mode yields the gradient of a given function at a time and storage cost roughly proportional to the number of variables that the function depends upon. This may be quite acceptable for a small number of variables, but may cause difficulties for problems with many variables.

The reverse mode yields the gradient of a scalar-valued function for no more than five times the operations count of evaluating the function itself. This bound is completely independent of the number of independent variables and allows the row-wise computation

*Development of ADOL–F was supported by the DFG, Grant no. Gr. 705/4-1

†Inst. of Scientific Computing, TU Dresden, D–01062 Germany, dima@math.tu-dresden.de

‡Inst. of Scientific Computing, TU Dresden, D–01062 Germany, griewank@math.tu-dresden.de

of Jacobians for at most 5 times the number of independents times the effort of evaluating the underlying vector function. When the number of component functions is larger than the number of variables that they depends upon, then Jacobians can in general be obtained more cheaply column by column through propagating gradients forward.

The ADOL–F [10] package utilizes overloading in Fortran 90, but one can continue to write Fortran programs according to the earlier standard, because all standard conforming Fortran 77 programs remain standard conforming Fortran 90 programs. See e.g. [2] for an overview on conversion from Fortran 77 to Fortran 90. The acronym ADOLF–F stands for **A**utomatic **D**ifferentiation by **O**ver**L**oading in **F**ORTRAN. ADOL–F is a Fortran 90 interface to the ADOL–C [5] library and provides a similar but somewhat restricted functionality due to impossibility to reflect some C++ features in Fortran 90.

Before now, in order to compute derivatives of Fortran codes with the help of ADOL–C the user had to at first to specify his problem as a C programm (e.g. using *f2c* precompiler) and generate an execution trace (*tape*) using a C++ compiler. Afterwards the necessary ADOL–C functions for computation of derivatives could be called direct from Fortran. The whole process has proved to be rather tedious and and was not readily accepted by the users. Because of it we decided to replace all C++ dependent ADOL–C pieces of code with their Fortran 90 counterparts.

ADOL–F 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 like the square of the degree of the derivative but is still completely independent of the numbers of dependent and independent variables in the underlying function. In contrast to some other packages for automatic differentiation e.g. [1, 6, 8] we employ an operator overloading scheme rather than transforming the code using a precompiler. Though loosing some opportunities of compiler-level optimization it gives us the possibility to compute derivatives of any order at a reasonable cost in space and time. A Fortran 90 environment by Kearfott [7] (also based on operator overloading) provides only first and second order derivatives and uses only local storage in the reverse mode thus restricting drastically the size of the problems that can be handled.

2 Preparing a Section of Fortran Code for Differentiation

ADOL–F was designed so that the user has to make only minimal changes to his undifferentiated code. The main modifications concern variable declarations and input/output operations. A key ingredient of automatic differentiation is the concept of an *active variable*. All programm variables that may at some time during the program execution contain differentiable quantities must be declared to be of an active type. ADOL–F uses one active type, called A_REAL, whose real part is of the type REAL(P_) where the parameter P_ is defined in the module ADOLF_I as INTEGER, PARAMETER :: P_ = KIND(0.0D0) thus all REALs are internal of DOUBLE PRECISION 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* real or integer types.

The actual computational statements of a Fortran code need not be altered for the purposes of automatic differentiation. Legitimate Fortran code can remain completely unchanged, provided the direct output of active variables is avoided. All calculations involving active variables that occur between the subroutine calls

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

are recorded on a sequential data set called *tape*. We will refer to the sequence of statements executed between a particular call to trace_on and the following call to trace_off as an *active section* of the code. The same active section may be entered repeatedly, and one can successively generate several traces on distinct tapes by changing the tag. If the active section involves branches conditioned on a_real comparisons, the validity of a tape is automatically checked so that it may be used safely at other argument points. If the tape is no longer valid the active section must be executed and retaped at the new argument.

After the execution of a sizeable active section, a tape file will be saved in the current working directory. 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. The tape generated by small active sections may remain in a dynamic array unless the user forces the writing of a tape file by calling trace_off with file == .true.

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

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).

3 Implementation Issues

ADOL–C really consists of two separate parts. The first utilizes the overloading capabilities of C++ to translate the function specified by a C++ evaluation programm into the tape format. The second part contains numerous utilities and drivers written in C to repeatedly calculate derivatives from the tape using various arguments including directions and weights. ADOL–F essentially replaces the first part by a corresponding Fortran 90 front end and calls the second with more convenient drivers in Fortran.

Some Fortran 90 features which are important for us are:

- User-defined operators
- Functions and Operators with arbitrary result type
- Overloading of procedures, functions and operators
- Module concept
- Dynamic arrays
- Access to subarrays

ADOL–F overloads all real and integer arithmetic operations, as well as comparison, assignment operators and most intrinsic mathematical functions.

- The Operators $+, -, *, /$ and $**$ with a_real result for two scalars of types
  - both a_real
  - one a_real and one real
  - one a_real and one integer

- The monadic operators $+, -$ with A_REAL result for type A_REAL
- The operators $==, /=, >, >=, <, <=$ with LOGICAL result for two scalars of types
  - both A_REAL
  - one A_REAL and one REAL
  - one A_REAL and one INTEGER
- Assignment to a scalar of type A_REAL from a scalar of type A_REAL, REAL, or INTEGER.
- The functions ABS, MIN, MAX, SQRT, LOG, LOG2, LOG10, EXP,SIN, COS, TAN, ACOS, ASIN, ATAN, COSH, SINH, TANH with A_REAL results for a scalar argument of type A_REAL.

For our purposes the main drawback of Fortran 90 in comparison with C++ is the absence of constructor and destructor operators for user–defined types. Though not being of obvious importance to the end–user of ADOL–F it leads to some restriction in ADOL–F in conrast to ADOL–C. In particular it is currently necessary to initialize all active variables explicitly. Only the next revision of Fortran 90 standard ("Fortran 95") incorporates the possibility to initialize objects of user–defined type. Unfortunately, it appears that even with the next standard, some kinds of variables like function results may still need to be initialized explicitly.

The absence of constructors and destructors leads to the fact that in contrast to ADOL–C local active variables that go out of scope cannot release their 'location' (an internal address used to identify intermediate quantities for the reverse sweep) This effect may lead to an excessively large range of locations, which is likely to increase the number of page faults as well as the memory requirement. This disadvantage will not be serious if the Fortran 90 code is written without recursive subroutines or functions so that the total number of named variables is limited. To deal with the possibily large number of compiler generated temporaries we employ a more sophisticated scheme for the management of active temporary variables than ADOL–C. As a result ADOL–F uses a separate storage space for the values of temporary variables (results of intermediate operations) and release their locations direct in the consuming operation. Each linear operation with an operand being temporary just stores an operation code indicating that the value of the temporary is not needed and is not recorded on the tape.

Each non-linear operation checks its operands and either stores a value of an operand if it has a *temporary* flag (together with an operations code indicating this fact) or just set the *non-linear* flag for this operand.

The *non-linear* flag is checked for the left hand side of an assignment operator and if positive then the value of the operand is recorded on the tape. Accordingly, there are two different operation codes for every assignment operator indicating whether the overwritten value of the left hand side was recorded or not. This way also a user-defined active variable is recorded on the tape only if it is an operand of a non-linear operation.

4 General Mathematical Description

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

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

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

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

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

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

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

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

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

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

and

$$y_2 = F'(x_0)\, x_2 + \tfrac{1}{2} F''(x_0)\, x_1\, x_1 \quad . \tag{5}$$

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

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

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

and their derivatives satisfy the identity

$$\frac{\partial y_j}{\partial x_i} = \frac{\partial y_{j-i}}{\partial x_0} = A_{j-i}(x_0, x_1, \ldots, x_{j-i}) \tag{6}$$

as established in [3]. 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.

5 Function FORWARD

Given the tape of an active section and the coefficients x_j, the resulting y_j can be evaluated by appropriate calls to the overloaded ADOL–F function FORWARD. The scalar version of FORWARD propagates just one truncated Taylor series from the $(x_j)_{j\le d}$ to the $(y_j)_{j\le d}$.

Here we declare the parameters, which occur often :

```
INTEGER, INTENT(IN)  :: TAG   ! Tape identification
INTEGER, INTENT(IN)  :: M     ! Number of dependent variables
INTEGER, INTENT(IN)  :: N     ! Number of independent variables
INTEGER, INTENT(IN)  :: D     ! Highest derivative degree
INTEGER, INTENT(IN)  :: KEEP  ! Flag for reverse sweep
```

```
INTEGER FUNCTION FORWARD(TAG,M,N,D,KEEP,X,Y)
...          ...                ...
REAL(P_), INTENT(IN)    :: X(N,0:D)  ! independent-variable values
REAL(P_), INTENT(OUT)   :: Y(M,0:D)  ! result coefficients as in (3)
```

The rows of the matrix X must correspond to the independent variables in the order of their initialization by the INDEPENDENT subroutine. The columns of $X = \{x_j\}_{j=0..d}$ represent Taylor coefficient vectors as in Equation (1). The rows of the matrix Y must correspond to the dependent variables in the order of their selection by the DEPENDENT subroutine.

The columns of Y= $\{y_j\}_{j=0..d}$ represent Taylor coefficient vectors as in (3). Thus the first column of Y contains the function value $F(x)$ itself, the next column represents the first Taylor coefficient vector of F, and the last column the D-th Taylor coefficient vector. The integer flag KEEP determines whether and how many Taylor coefficients of all intermediate quantities are written by FORWARD into a buffered temporary file in preparation for a subsequent reverse sweep.

If M or D attain their trivial values 1 and 0, respectively, then corresponding dimensions of the arrays X or Y can be omitted, thus eliminating one level of indirection.

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.

FORWARD can be used to evaluate the vector-function F at arguments x other than the point at which the tape was generated. If the active section involves branches conditioned on A_REAL comparisons, the validity of a tape is automatically checked so that it may be used at another argument points. If not valid the active section must be executed and retaped at the new argument. Otherwise the numerical values may (and almost certainly will) be incorrect. Currently, the following return values are used (see also Fig.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 MAX(A,B) returned A we get B at the argument currently used in forward or reverse routines.
+1	At least one of the functions MIN, MAX or ABS is evaluated at a tie or zero, respectively. Hence, F is Lipschitz-continuous but possibly nondifferentiable.
0	Some arithmetic comparison involving A_REALs yields a tie. Hence, F may be discontinuous in the vicinity.
-1	An A_REAL comparison yields different results from the evaluation point at which the tape was generated. Resulting derivative vectors are meaningless

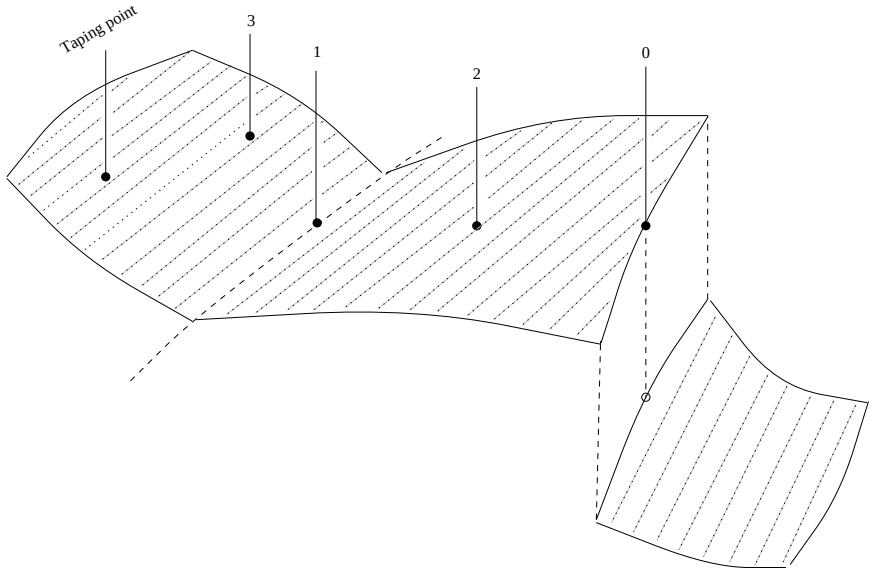

Fig. 1. *Return values around the taping point*

The vector version of FORWARD propagates families of $p \geq 1$ truncated Taylor series in order to reduce the relative cost of the overhead incurred in the tape interpretation. The following parameters are used:

```
INTEGER,  INTENT(IN)   ::  P      ! Number of Taylor series
REAL(P_), INTENT(IN)   ::  X(N)   ! values of independent variables
REAL(P_), INTENT(IN)   ::  Y(M)   ! values of dependent variables
```

```
INTEGER FUNCTION FORWARD(TAG,M,N,D,P,X,X,Y,Y)
  ...          ...                    ...
  REAL(P_),  INTENT(IN)      ::  X(N,P,D)   ! Taylor coeff. of indep. variables
  REAL(P_),  INTENT(OUT)     ::  Y(M,P,D)   ! Taylor coeff. of dep. variables
```

where X and Y hold the Taylor coefficients of first and higher degree and X, Y the common Taylor coefficients of degree 0. There is no option to keep the values of active variables

that are going out of scope or that are overwritten. Therefore this function cannot prepare a subsequent reverse sweep.

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

```
INTEGER FUNCTION FORWARD(TAG,M,N,P,X,X,Y,Y)
...          ...                ...
REAL(P_),    INTENT(IN)     ::  X(N,P)   ! Seed derivatives of indep. variables
REAL(P_),    INTENT(OUT)    ::  Y(M,P)   ! First derivatives of dep. variables
```

When this routine is called with P=N and X the identity matrix, the resulting Y is simply the Jacobian $F'(x)$. In general, one obtains the $m \times p$ matrix $Y = F'(x)\,X$ for the chosen initialization of X.

6 Function REVERSE

Given a weighting vector $u \in I\!R^m$, the ADOL–F function REVERSE computes the collection of row vectors

$$z_j \equiv u^T \frac{\partial y_j}{\partial x_0} = u^T A_j \in I\!R^n \tag{7}$$

for $j = 0, 1, \ldots, d$. If $j = 0$ and u is the i-th Cartesian basis vector in $I\!R^m$, then (7) yields the i-th row of the Jacobian $F'(x)$. To produce the entire Jacobian in this mode, one may make m calls to REVERSE, setting u to the i-th Cartesian basis vector for $i = 1, 2, \ldots, m$.

After the execution of an active section with KEEP = .TRUE. or a call to FORWARD with any KEEP $\leq$ D+1, one may call the function REVERSE with D = 0 or with D=KEEP-1 and the same tape identifier TAG, respectively. When U is a vector and Z an $n \times (d+1)$ matrix as in (7), REVERSE is executed in the *scalar mode* by the following calling sequence:

```
INTEGER FUNCTION REVERSE(TAG,M,N,D,U,Z)
...          ...                ...
REAL(P_),    INTENT(IN)     ::  U(M)     ! weighting vector
REAL(P_),    INTENT(OUT)    ::  Z(N,0:D) ! result adjoints as in (7)
```

The return values of the reverse routines equal those of the forward routines described before except for the following two differences. First, reverse cannot return negative values since the corresponding forward sweep would have stopped without completing the necessary taylor file. Second, the return value of reverse may be higher than that of the preceeding 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.

If M or D attain their trivial values 1 and 0, respectively, then corresponding dimensions of the arrays U or Z can be omitted, thus eliminating one level of indirection.

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

$$Z_j \equiv U \frac{\partial y_j}{\partial x_0} \in I\!R^{p \times n}, \tag{8}$$

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

When U is a matrix as in (8), REVERSE is executed in the *vector mode* by the following calling sequence:

```
INTEGER FUNCTION REVERSE(TAG,M,N,D,P,U,Z)
  ...          ...          ...
  INTEGER,     INTENT(IN)   ::  P          ! number of weight vectors
  REAL(P_),    INTENT(IN)   ::  U(P,M)     ! domain weight vector
  REAL(P_),    INTENT(IN)   ::  Z(P,N,0:D) ! result adjoints as in (8)
```

If M or D attain their trivial values 1 and 0, respectively, then corresponding dimensions of the arrays U or Z can be omitted, thus eliminating one level of indirection.

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.

The arguments P and U can omitted, and default in this case to m and the identity matrix of order m, respectively.

```
INTEGER FUNCTION REVERSE(TAG,M,N,D,Z)
  ...          ...          ...
  REAL(P_),    INTENT(IN)   ::  Z(M,N,0:D) ! result adjoints as in (8)
```

7 Derivatives for Optimization and Nonlinear Equations

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

$$z_0^T = u^T F'(x_0) \tag{9}$$

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

$$y_1 = F'(x_0)\, x_1, \tag{10}$$

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

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

$$z_1 = \frac{\partial y_1}{\partial x_0} = \frac{\partial F'(x_0)x_1}{\partial x_0} = \nabla^2 F(x_0)x_1 \tag{11}$$

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

Let the parameter X represent the independent vector x_0 as in (1)

```
REAL(P_),    INTENT(IN)   ::  X(N)  ! independent vector x_0
```

For convenience one may use instead of FORWARD and REVERSE the following functions:

```
INTEGER FUNCTION EVALUATE (TAG,M,N,X,Y)
  ...          ...                ...
  REAL(P_),  INTENT(OUT)  ::  Y(M)   ! result y_0 as in (4)
```

```
INTEGER FUNCTION GRADIENT(TAG,N,X,G)
  ...          ...                ...
  REAL(P_),  INTENT(OUT)  ::  G(N)   ! result z_0 as in (9) for u = 1 ∈ IR
```

```
INTEGER FUNCTION VEC_JAC(TAG,M,N,REPEAT,X,U,Z)
  ...          ...                ...
  INTEGER,   INTENT(IN)   ::  REPEAT  ! has vec_jac been called here ?
  REAL(P_),  INTENT(IN)   ::  U(M)    ! range weight vector
  REAL(P_),  INTENT(OUT)  ::  Z(N)    ! result z_0 as in (9)
```

```
INTEGER FUNCTION JACOBIAN(TAG,M,N,X,J)
  ...          ...                ...
  REAL(P_),  INTENT(OUT)  ::  J(M,N)  ! output Jacobian F'(x_0)
```

```
INTEGER FUNCTION HESSIAN(TAG,N,X,H)
  ...          ...                ...
  REAL(P_),  INTENT(OUT)  ::  H(N,N)  ! Hessian matrix ∇²f(x_0)
```

The following functions get additionally a tangent vector as a parameter

REAL(P_), INTENT(IN) :: V(N) ! tangent vector x_1

```
INTEGER FUNCTION JAC_VEC(TAG,M,N,X,V,Z)
  ...          ...                ...
  REAL(P_),  INTENT(OUT)  ::  Z(M)   ! result y_1 as in (10)
```

```
INTEGER FUNCTION HESS_VEC(TAG,N,X,V,Z)
  ...          ...                ...
  REAL(P_),  INTENT(OUT)  ::  Z(N)   ! result z_1 as in (11)
```

```
INTEGER FUNCTION LAGRA_HESS_VEC(TAG,M,N,X,V,U,H)
  ...          ...                ...
  REAL(P_),  INTENT(IN)   ::  U(M)   ! range weight vector
  REAL(P_),  INTENT(OUT)  ::  H(N)   ! result z_1 as in (7)
```

The scalar-mode drivers GRADIENT and HESS_VEC create a temporary file by an appropriate call to FORWARD with KEEP=1 and then call REVERSE with the corresponding argument. The routine VEC_JAC functions in the same way, except that the internal call to FORWARD is omitted if a nonzero value of the parameter REPEAT indicates that FORWARD has already been called at the same argument. When M=1 EVALUATE, GRADIENT, and HESSIAN can be used to evaluate the scalar function and its derivatives at any argument in its domain if the tape remains valid (see return codes).

8 Derivatives for Differential Equations

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

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

we must have $m = n$. Along any solution path $x(t)$ its Taylor coefficients x_j at some time, say $t = 0$, must satisfy the relation (2) with the y_j the Taylor coefficients of its derivative

$y(t) = x'(t)$, namely,

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

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

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

```
INTEGER FUNCTION FORODE(TAG,N,TAU,DOL,DEG,X)
   INTEGER, INTENT(IN)  :: TAG          ! tape identification
   INTEGER, INTENT(IN)  :: N            ! number of state variables
   REAL(P_), INTENT(IN) :: TAU          ! scaling parameter
   INTEGER, INTENT(IN)  :: DOL          ! degree on previous call
   INTEGER, INTENT(IN)  :: DEG          ! degree on current call
   REAL(P_), INTENT(OUT) :: X(N,0:DEG)  ! Taylor coefficient vectors
```

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

REVERSE(TAG,N,N,DEG-1,Z)

to compute the family of square matrices Z[N,N,DEG] defined in equation (8) of Section 5.1.

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

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

which exist provided F is sufficiently smooth. These matrices can be obtained from the partial derivatives $\partial y_i / \partial x_0$ obtained from REVERSE by an appropriate version of the chain rule. This task is performed by the utility ACCODE, which involves $\frac{1}{2}d(d-1)$ matrix-matrix products.

To compute the total derivatives B=$(B_j)_{0 \le j < d}$ defined in (12), one may finally call the following:

```
INTEGER FUNCTION ACCODE(N,TAU,DEG,Z,B)
   INTEGER, INTENT(IN)  :: N            ! number of state variables
   REAL(P_), INTENT(IN) :: TAU          ! scaling parameter
   INTEGER, INTENT(IN)  :: DEG          ! degree on current call
   REAL(P_), INTENT(IN) :: Z(N,N,DEG)   ! partials of coefficient vectors
   REAL(P_), INTENT(OUT) :: B(N,N,DEG)  ! results as defined in (12)
```

9 Example

9.1 $y = f(x) = \prod_{i=0}^{n-1} x_i$

Evaluating of the gradient and a Hessian-vector product for the function

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

```
PROGRAM  Product
 USE ADOLF_I
```

```
 USE ADOLF_COND
 USE ADOLF
 INTEGER, PARAMETER  ::  N = 5, tag = 1
 INTEGER :: I,J, ERR = 0
 REAL(P_) :: XP (N) ,Grad (N),  Hess (N,N), JAcob (1,N),YP = 0.0_P_
 TYPE(A_Real), Dimension (N) :: X = (/(A_Real(-1, .FALSE.), I=1,N)/)
 TYPE(A_Real) :: Y = A_Real(-1, .FALSE.)

 DO I=1, N
   XP(I) = I
 END DO
 CALL Trace_On(tag) ! Begin of the active section, KEEP == .TRUE.
 Y = 1.0_P_
 DO I=1, N
   CALL Independent (X(I), XP(I))! Initialization of active variables
   Y = Y * X(I)
 END DO
 CALL  Dependent ( YP, Y)
 CALL Trace_Off(.TRUE.) ! End of the active section
 err = EVALUATE(tag,1,N, XP, GRAD)! Evaluate the function anew
 Err = gradient (tag,N, XP, Grad)! Compute the gradient
 Err = hessian(tag, N, XP, hess)! Compute the hessian

 DO I=1, N ! Estimate the error of the gradient's computation
   Grad_Error= Grad_Error + ABS(Grad(i)-YP/XP(i))
 END DO
 errh =0.0_P_
 DO I=1, N ! Estimate the error of the hessian's computation
   DO J=1, N
IF (i > j)   errh = errh + abs( hess(i,j)-grad(i)/xp(j))
   END DO
 END DO
 WRITE (*,*) 'Gradient  Error=', Grad_Error
 WRITE (*,*) 'Consistency check',  errh
END PROGRAM Product
```

References

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

[2] A. G. Buckley, *Conversion to Fortran 90: A Case Study*, ACM TOMS, 20 (1994), pp. 308–353.

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

[4] A. Griewank and G. F. Corliss, (eds.), *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, SIAM, Philadelphia, Penn., 1991.

[5] A. Griewank, A., D. Juedes, and J. Utke, *ADOLC–C: A Package for the Automatic Differentiation of Algorithms Written in C/C++*, ACM TOMS, (1996), to appear.

[6] J. E. Horwedel, *GRESS, a Preprocessor for Sensitivity Studies of Fortran Programs*, in [4], 1991, pp. 243–250.

[7] R. B. Kearfott, *A Fortran 90 Environment for Research and Prototyping of Enclosure Algorithms for Nonlinear Equations and Global Optimization*, ACM TOMS 21 (1995), pp. 63–78.

[8] K. Kubota, *PADRE2, a Fortran precompiler yielding error estimates and second derivatives*, in [4], 1991, pp. 251–262.

[9] D. Shiriaev, *Fast automatic differentiation for vector processors and reduction of the spatial complexity in a source translation environment*, Dissertation, Mathematik, Universität Karlsruhe, 1993. ftp://ftp.math.tu-dresden.de/pub/reports/wir/dima/diss.ps.Z

[10] D. Shiriaev, A. Griewank, and J. Utke, *A User Guide to ADOL–F: Automatic Differentiation of Fortran Codes*, Technical Report, Institute of Scientific Computing, TU Dresden. ftp://ftp.math.tu-dresden.de/pub/reports/wir/dima/adolf_giude.ps.Z