

A Differentiation-Enabled Fortran 95 Compiler

Uwe Naumann,

Mathematics and Computer Science Division, Argonne National Laboratory,
9700 South Cass Ave., Argonne, IL 60349-4844, USA

and

Jan Riehme,

Department of Computer Science, University of Hertfordshire,
College Lane, Hatfield, AL10 9AB, UK

The availability of first derivatives of vector functions is crucial for the robustness and efficiency of a large number of numerical algorithms. A new differentiation-enabled Fortran 95 compiler is described that uses programming language extensions and a semantical code transformation known as automatic differentiation to provide Jacobians of numerical programs with machine accuracy. We describe the user interface as well as the relevant algorithmic details. The feasibility, robustness, and convenience of this approach is illustrated by various case studies.

Categories and Subject Descriptors: D.2.2 [Tools and Techniques]: Software libraries

General Terms: Algorithms

Additional Key Words and Phrases: Automatic differentiation

1. MOTIVATION

Consider the solution of the equation $F(\mathbf{x}) = 0$ where the vector function F is implemented as a Fortran subroutine, for example, `f(x, f_vec)` that computes `f_vec` as a function of `x`. This subroutine may be very complex, possibly involving calls to other subroutines or user-defined functions. Suppose that the routine `nag_nlin_sys_sol` that is part of NAG Fortran 90 Library is to be applied to this problem. The solver expects a subroutine `fun(x, finish, f_vec, f_jac)`¹ that computes the function value `f_vec` and, ideally, the Jacobian `f_jac`. Our goal is to make the subroutine `fun` a simple “generic wrapper” calling `f` such that the derivative code is generated automatically by the compiler.

¹See documentation of the NAG Fortran 90 Library at www.nag.co.uk for further details on `fun` and its arguments.

This work was supported by the U.K. Engineering and Physical Sciences Research Council under grant GR/R55252/01 (“Differentiation-enabled Fortran 95 Compiler Technology”).

Naumann was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 2004 ACM 0098-3500/2004/1200-0001 \$5.00

The first prototype of the differentiation-enabled NAGWare Fortran 95 compiler provides this functionality. A limited number of language extensions is required, as shown in the following example.

```

SUBROUTINE fun(x,finish,f_vec,f_jac)
  USE ACTIVE_MODULE
  ...
  TYPE(JACOBI_TYPE) :: jac

  DIFFERENTIATE
    INDEPENDENT(x)
    CALL f(x,f_vec)
    jac=JACOBIAN(f_vec,x)
  END DIFFERENTIATE

  CALL DERIVTOREAL(jac,f_jac)
  ...
END SUBROUTINE fun

```

The subroutine `f` needs to be called inside the *active section* that is marked by the `DIFFERENTIATE` and `END DIFFERENTIATE` statements. The vector `x` is declared as *independent*, and the Jacobian of `f_vec` with respect to `x` can be obtained by calling the function `JACOBIAN` that is defined in `ACTIVE_MODULE`. The real values of the Jacobian object `jac` are extracted by calling another support routine, named `DERIVTOREAL`. The code is compiled with the differentiation-enabled NAGWare Fortran 95 compiler to ensure that the code for the computation of the Jacobian is generated automatically.

In the main part of this paper, we discuss details of all the new features used and of their implementations. Following a brief introduction (Section 2) to the principles of *forward mode automatic differentiation* we focus in Sections 3 and 4 on the relevant compiler internals. We illustrate the interface in the context of a simple first-derivative arithmetic based on operator overloading (described in greater detail in [Cohen et al. 2003]). Our main focus is on the source transformation solution that is built on the idea of preaccumulating local gradients of scalar assignments. Additional features of the compiler include the ability to exploit sparsity within the Jacobian by *seeding*. In Section 5 we present several case studies that underline the flexibility and ease of use of the compiler. A crucial next step in the development is the automatic generation of *adjoint code* by using the *reverse mode* of automatic differentiation. First thoughts in this direction are presented in Section 6 together with conclusions.

2. FORWARD MODE AUTOMATIC DIFFERENTIATION

Derivatives of vector functions that represent mathematical models of scientific, engineering, or economic problems play an increasingly important role in modern numerical computing. They can be regarded as the enabling key factor allowing for a transition from the pure simulation of the real-world process to the optimization of some specific objective with respect to a set of model parameters. For a given computer program that implements an underlying numerical model, automatic dif-

ferentiation (AD) [Corliss and Griewank 1991; Berz et al. 1996; Corliss et al. 2002; Griewank 2000] provides a set of techniques for transforming the program into one that computes not only the function value for a set of inputs but also the corresponding first and higher derivatives. A large portion of the ongoing research in this field is aimed at improving the efficiency of the derivative code that is generated. Successful methods are often built on a combination of classical compiler algorithms and the exploitation of mathematical properties of the code. Moreover, improving the user-friendliness of software tools for AD is an important issue. The integration of differentiation capabilities into industrial-strength compilers appears to be reasonable approach. Language extensions to support software projects in scientific computing can hide most of the AD internals from the user.

We consider numerical simulation programs written in Fortran 95 that implement nonlinear vector functions

$$F : \mathbb{R}^n \supseteq D \rightarrow \mathbb{R}^m : \mathbf{x} \mapsto \mathbf{y} = F(\mathbf{x}) \quad .$$

The Jacobian matrix of F at a given argument \mathbf{x}_0 is denoted by

$$F' = F'(\mathbf{x}_0) = \left(\frac{\partial y_i}{\partial x_j}(\mathbf{x}_0) \right)_{\substack{i=1,\dots,m \\ j=1,\dots,n}} \quad .$$

The forward mode of AD transforms the program for F into a tangent-linear model \dot{F} such that

$$\dot{\mathbf{y}} = \dot{F}(\mathbf{x}, \dot{\mathbf{x}}) = F'(\mathbf{x})\dot{\mathbf{x}} \quad .$$

Here, $\dot{\mathbf{y}} \in \mathbb{R}^m$ is the directional derivative of F in the direction $\dot{\mathbf{x}} \in \mathbb{R}^n$. Its numerical value is computed with machine accuracy. For example, the whole Jacobian matrix can be accumulated by letting $\dot{\mathbf{x}}$ range over the Cartesian basis vectors in \mathbb{R}^n since (obviously) $F'(\mathbf{x}) = F'(\mathbf{x}) \cdot I_n$, where I_n denotes the identity in \mathbb{R}^n . Let us have a closer look at the way this source code transformation is performed conceptually.

First, F is decomposed into a *code list* by assigning the result of any arithmetic operation (for example, $+$, $*$) or intrinsic function (for example, \sin , \exp) to a unique intermediate variable. Thus, the code list becomes a sequence of (in general) nonlinear assignments

$$v_j = \varphi_j(v_i)_{i \prec j} \quad \text{for } j = 1, \dots, q \text{ and } q = p + m.$$

The number of intermediate variables is denoted by p . Following the notation in [Griewank 2000], we denote the set of arguments of φ_j as $\{v_i : i \prec j\}$, that is, $i \prec j$ if v_i is an argument of φ_j . Furthermore, we set $x_i \equiv v_{i-n}$, $i = 1, \dots, n$, $z_k \equiv v_k$, $k = 1, \dots, p$, and $y_j \equiv v_{p+j}$, $j = 1, \dots, m$.

Forward-mode AD generates a tangent-linear model automatically for a given program for F . This approach relies on the existence of jointly continuous partial derivatives for all elemental functions φ_j , $j = 1, \dots, q$, on open neighborhoods $\mathcal{D}_j \subset \mathbb{R}^{\{i:i \prec j\}}$ of their respective domains. Directional derivatives \dot{v}_i are associated with every code list variable for $i = 1 - n, \dots, q$. The computation becomes

$$\dot{v}_j = \sum_{i \prec j} c_{j,i} \cdot \dot{v}_i \quad , \quad (1)$$

for $j = 1, \dots, q$ and where

$$c_{j,i} \equiv \frac{\partial \varphi_j(v_i)_{i \prec j}}{\partial v_i} \quad .$$

The values \dot{v}_j represent the results of the inner products of the local gradient $\frac{\partial v_j}{\partial \mathbf{x}}$ with $\dot{\mathbf{x}}$. For the computation of several directional derivatives at the same point the *vector forward mode* of AD can be used to transform F into \dot{F} such that

$$\dot{Y} = \dot{F}(\mathbf{x}, \dot{X}) = F'(\mathbf{x})\dot{X} \quad . \quad (2)$$

The columns in $\dot{Y} \in \mathbb{R}^{m \times l}$ are the directional derivatives of F corresponding to the directions that form the columns of $\dot{X} \in \mathbb{R}^{n \times l}$. With $\dot{v}_i \in \mathbb{R}^l$ in Equation (1) the originally scalar multiplication $c_{j,i} \cdot \dot{v}_i$ becomes a product of a scalar with a vector. Potentially, such a product can be implemented very efficiently for languages that provide vector arithmetic such as Fortran 95.

3. LANGUAGE INTERFACE FOR AUTOMATIC DIFFERENTIATION

The first prototype of the differentiation-enabled compiler used operator overloading to compute derivative information in parallel with the evaluation of the function itself. This approach is described in [Cohen et al. 2003]. A new *active data type* is introduced that holds a vector for the directional derivatives in addition to the function value. All arithmetic operators and intrinsic function are overloaded for this new data type, not only to compute the function value but also to perform the computation of the directional derivatives. See [Griewank 2000] for further details on forward-mode AD by operator overloading.

The active data type contains a component to hold the function value `value` and an allocatable array of directional derivatives `deriv`.

```
TYPE ACTIVE_TYPE
  DOUBLE PRECISION :: value
  DOUBLE PRECISION, ALLOCATABLE, DIMENSION(:) :: deriv
END TYPE ACTIVE_TYPE
```

The user of the compiler is never required to use this data type explicitly. The redeclaration of active variables as of type `ACTIVE_TYPE` is done automatically once the independent and dependent variables are known. The active data type is used both in the context of operator overloading and source transformation.

Referring back to the motivating example in Section 1, we observe that six modifications of the original code are required to use the differentiation capability of the compiler.

- (1) The module `ACTIVE_MODULE` needs to be used. It contains the active data type, all overloaded arithmetic operators and intrinsic functions, and various support routines.
- (2) A variable (for example, `jac`) needs to be declared to hold the Jacobian matrix. The special type `JACOBI_TYPE` ensures that `jac` is recognized as passive by the *activity analysis* (see below).
- (3) The active section is enclosed in `DIFFERENTIATE ... END DIFFERENTIATE`. Any derivative computation is restricted to the active section.

- (4) The independent variables are marked by using the INDEPENDENT statement. This results in their type getting switched to “active.” In Jacobian computation mode the derivative components are initialized with the identity in \mathbb{R}^n .
- (5) JACOBIAN is a support routine provided by ACTIVE_MODULE. It is used to access the derivative component(s) of the active variable that forms its argument. This process is equivalent to retrieving its Jacobian with respect to the independent variables. Optionally, the independent variable can be specified explicitly to extract only the corresponding rows of the Jacobian. This may be helpful if there are several independent variables. If several dependent variables exist then the corresponding columns of the Jacobian need to be extracted by separate calls to JACOBIAN.
- (6) The support routine DERIVTOREAL is used to transform jac into a real matrix.

Refer to the projects Web site

http://www.nag.co.uk/nagware/research/ad_overview.asp

for several simple examples that illustrate the various uses of the new compiler.

In the current version of the compiler, no static data-flow analysis is performed. Conservatively, all variables that occur on the left-hand side of some assignment and all subroutine arguments are made active. This approach ensures the correctness of the derivative code generated; however, it lacks the potential efficiency that can be achieved by using a proper activity analysis [Hascoët et al. 2002]. Activity information can be exploited at the level of variable instances. The compiler can generate code that switches between the active and the passive version of one and the same variable. The addition of these features is highly desirable and is part of our future work.

4. COMPILER AD WITH STATEMENT-LEVEL PREACCUMULATION

The latest prototype of the differentiation-enabled compiler no longer relies on operator overloading to compute directional derivatives in the forward mode of AD. It exploits the idea of statement-level preaccumulation as described in [Naumann 2002b]. Given the gradient f' of some scalar assignment $y = f(x_1, \dots, x_k)$ only the propagation of the directional derivative $\dot{y} = f' \cdot \dot{x}_1 + \dots + f' \cdot \dot{x}_k$ is performed by a subroutine that is defined in ACTIVE_MODULE. The computation of the local gradients is performed by code that is generated explicitly by the compiler. Below we present a more detailed description of this approach.

A tangent-linear model $\dot{\mathbf{y}} = \hat{F}(\mathbf{x}, \dot{\mathbf{x}})$ represents a system of linear equations that can be written in matrix form as follows:

$$\begin{pmatrix} \dot{\mathbf{z}} \\ \dot{\mathbf{y}} \end{pmatrix} = C \cdot \begin{pmatrix} \dot{\mathbf{x}} \\ \dot{\mathbf{z}} \end{pmatrix}, \quad (3)$$

where $C \in \mathbb{R}^{q \times (n+p)}$ is the *extended Jacobian* that is defined as

$$C = (c_{j,i})_{\substack{j=1,\dots,q \\ i=1-n,\dots,p}} \quad (4)$$

with local partial derivatives $c_{j,i}$. The computation of $\dot{\mathbf{y}} = F' \cdot \dot{\mathbf{x}}$ can be interpreted as the solution of Equation (3) for $\dot{\mathbf{y}}$ in terms of $\dot{\mathbf{x}}$ [Naumann and Gottschling 2003].

To solve Equation (3) for $\dot{\mathbf{y}}$ in terms of $\dot{\mathbf{x}}$, we consider elimination techniques on C . Following the notation in [Griewank 2000], we partition the extended Jacobian C as follows.

$$C = \begin{pmatrix} B & L \\ R & T \end{pmatrix}, \quad (5)$$

where $B \in \mathbb{R}^{p \times n}$, $L \in \mathbb{R}^{p \times p}$, $R \in \mathbb{R}^{m \times n}$, and $T \in \mathbb{R}^{m \times p}$. Since the structure of C is induced by a code list, the matrix $L = (l_{j,i})_{j,i=1,\dots,p}$ must be strictly lower diagonal, that is, $l_{j,i} = 0$ if $i \geq j$. Solving Equation (3) for $\dot{\mathbf{y}}$ in terms of $\dot{\mathbf{x}}$ is regarded as the elimination of all nonzero elements in B, L , and T (see also [Griewank 2000]).

The preaccumulation of local gradients of scalar assignments has been investigated in [Naumann 2002b]. There we show that such gradients can be accumulated by using a minimal number of arithmetic floating-point operations by exploiting the property that each intermediate variable is used exactly once. The corresponding algorithm is also used to perform statement-level gradient preaccumulation inside the compiler.

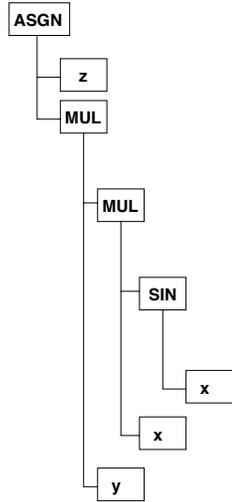


Fig. 1. Abstract syntax tree for $z = \sin(x) \cdot x \cdot y$.

We present an example to illustrate the implementation of the preaccumulation algorithm in the compiler. Consider the scalar assignment $z = \sin(x) \cdot x \cdot y$ whose abstract syntax tree (AST) is shown in Figure 1. Its linearized code list

$$\begin{aligned} c_{1,-1} &= \cos(x); & v_1 &= \sin(x) \\ c_{2,-1} &= v_1; & c_{2,1} &= x; & v_2 &= v_1 \cdot x \\ c_{3,0} &= v_2; & c_{3,2} &= y; & z &= v_2 \cdot y \end{aligned}$$

leads to the following tangent-linear system to be solved for \dot{z} in terms of \dot{x} and \dot{y} :

$$\begin{pmatrix} \dot{v}_1 \\ \dot{v}_2 \\ \dot{z} \end{pmatrix} = \begin{pmatrix} 0 & c_{1,-1} & 0 & 0 \\ 0 & c_{2,-1} & c_{2,1} & 0 \\ c_{3,0} & 0 & 0 & c_{3,2} \end{pmatrix} \cdot \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{v}_1 \\ \dot{v}_2 \end{pmatrix}.$$

The optimal gradient accumulation algorithm chooses *column pivots* [Naumann and Gottschling 2003] $c_{2,1}$ and $c_{3,2}$ to transform the system into

$$\begin{pmatrix} \dot{v}_2 \\ \dot{z} \end{pmatrix} = \begin{pmatrix} 0 & c_{2,-1} + c_{2,1} \cdot c_{1,-1} & 0 \\ c_{3,0} & 0 & c_{3,2} \end{pmatrix} \cdot \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{v}_1 \end{pmatrix}$$

and

$$\dot{z} = (c_{3,0} \quad c_{3,2} \cdot (c_{2,-1} + c_{2,1} \cdot c_{1,-1})) \cdot \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix}.$$

The row vector in the last equation is equal to the transposed gradient of z with respect to x and y . The number of factors in the propagation of directional derivatives is decreased from initially five to two. The preaccumulation itself costs two

multiplications and one addition. For long `deriv` components of the active variables, that is, for large values of l in Equation (2), the local savings converge to a factor of 2.5. Figure 2 illustrates the corresponding transformation of the AST.

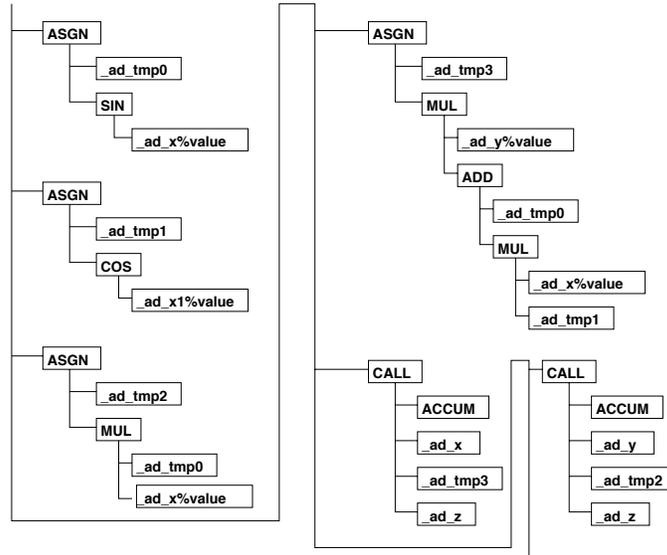


Fig. 2. Modifications of the AST for Statement-Level Preaccumulation. The AST of the original assignment depicted in Figure 1 is replaced by seven new statements. Six of them are relevant for the computation of the local directional derivative. `ad.tmp0` holds the value of `sin(x)`. The derivative of `sin(x)` with respect to `x` is stored in `ad.tmp1`. The second intermediate value, that is, the result of `ad.tmp0 · x`, is stored in `ad.tmp2`, which is equal to the partial derivative of `z` with respect to `y`. The second gradient entry is computed by column pivoting in the tangent-linear system. The propagation of the directional derivatives is performed by calling the appropriate accumulation routine from `ACTIVE_MODULE`. The seventh assignment, that is, `z = ad.tmp2 · y`, is not shown here. However, it needs to be generated by the compiler because gradients of statements to follow are likely to depend on the value of `z`.

Inside the compiler the extended Jacobian is built in terms of variable references. Actual values become available only at run time. Statements for computing these values have already been generated during the construction of the code list as described in Figure 2.

$$\left(\begin{array}{cc|cc} \mathbf{y} & \mathbf{x} & & \\ 0 & _ad_tmp1 & _ad_tmp0 & \\ 0 & _ad_tmp0 & _ad_x\%value & _ad_tmp2 \\ _ad_tmp2 & 0 & 0 & _ad_y\%value \mid \mathbf{z} \end{array} \right)$$

In addition to the extended Jacobian, variable references are stored for the independent variables `x` and `y`, the intermediate variables `_ad.tmp0` and `_ad.tmp2`, and the dependent variable `z`. These references are highlighted in the data structure

```

_adtd_flt_1_ = y_[i_ + -1];
_adtap_flt_1_ = &_ad_x_[1 + -1];
_adtd_flt_2_ = _adtap_flt_1_->value_;
_adtd_flt_3_ = _ad_tmp1_.value_;
_adtd_flt_4_ = _adtd_flt_2_*_adtd_flt_3_;
_adtd_flt_5_ = _ad_tmp2_.value_;
_adtd_flt_6_ = _adtd_flt_4_/_adtd_flt_5_;
_adtd_flt_7_ = 1/_adtd_flt_5_;
_adtd_flt_8_ = -(_adtd_flt_6*_adtd_flt_7_);
_adtd_flt_9_ = _adtd_flt_1_ - _adtd_flt_6_;
_adtd_flt_10_ = _adtd_flt_7_*-1;
_adtd_flt_11_ = _adtd_flt_8_*-1;
_adtd_flt_12_ = _adtd_flt_3*_adtd_flt_10_;
_adtd_flt_13_ = _adtd_flt_2*_adtd_flt_10_;

```

Fig. 3. C code that computes the gradient of the scalar assignment $z=y(i)-x(1)*temp1/temp2$ from the *analysis of an enzyme reaction problem*. All inputs are active except $y(i)$. The principal structure is similar to that described in Figure 2.

above. Column pivoting transforms the extended Jacobian into

$$\left(\begin{array}{cc|c} \mathbf{y} & \mathbf{x} & \\ 0 & _ad_tmp0 + _ad_x\%value \cdot _ad_tmp1 & | _ad_tmp2 \\ _ad_tmp2 & 0 & | _ad_y\%value \mid \mathbf{z} \end{array} \right)$$

and

$$\left(\begin{array}{cc|c} \mathbf{y} & \mathbf{x} & \\ _ad_tmp2 & _ad_tmp3 & | \mathbf{z} \end{array} \right) ,$$

while generating the new assignment

```
_ad_tmp3=_ad_x2%value*(_ad_x1%value*_ad_tmp1+_ad_tmp0) .
```

The NAGWare Fortran 95 front-end that we are working with transforms the Fortran code into C code and uses a native C compiler to generate executable programs. The C code that is generated can be stored in a separate file by activating the corresponding compiler switch. Figure 3 depicts the C code that results from the preaccumulation of the local gradient of the scalar assignment $z=y(i)-x(1)*temp1/temp2$ from the *analysis of an enzyme reaction problem* that is part of the MINPACK test problem collection [Averik et al. 1991] (see also Section 5.1). Furthermore, the differentiation phase of the compiler can generate output that illustrates the single steps performed by the preaccumulation algorithm. Example are shown on the projects Web site.

5. CASE STUDIES

We consider four case studies to illustrate the robustness and flexibility of the new compiler.

5.1 MINPACK Test Problem Collection

To test the correctness of the derivative code generated by the compiler, we implemented a test environment for fifteen vector functions from the MINPACK test problem collection [Averik et al. 1991]. In all cases the numerical values computed

```

n=134; m=252; ldfjac=m
ALLOCATE(x(n)); ALLOCATE(fvec(m)); ALLOCATE(fjac(ldfjac,n))
CALL dctsfj(m,n,x,fvec,fjac,ldfjac,"XS")
...
CALL dctsfj(m,n,x,fvec,fjac,ldfjac,"J")
...
DIFFERENTIATE
  INDEPENDENT(x)
  CALL dctsfj(m,n,x,fvec,fjac,ldfjac,"F")
  compAD_jac=JACOBIAN(fvec,x)
END DIFFERENTIATE
CALL DERIVTOREAL(compAD_jac,fjac)
...
DEALLOCATE(x); DEALLOCATE(fvec); DEALLOCATE(fjac)

```

Fig. 4. Verification of the differentiation-enabled compiler by using the coating thickness standardization problem. Inside the active section the subroutine `dctsfj` is called in function evaluation mode ("F"). The derivative code that evaluates the Jacobian is generated by the compiler. There can be several consecutive active sections in one file. This is exploited for writing similar test environments for all fifteen vector functions from the MINPACK test problem collection.

by the compiler-generated derivative code match those of the hand-written Jacobian code that is part of MINPACK. Figure 4 illustrates the test by using the coating thickness standardization problem. A driver program allocates the independent (`x`) and dependent (`fvec`) variables as well as the Jacobian matrix (`fjac`). It calls the subroutine (`dctsfj`) to compute the input values of `x` ("XS" mode). This value is then used to compute the Jacobian matrix of `fvec` with respect to `x` by using the hand-written Jacobian code ("J" mode). The same argument is then used to compute the Jacobian by using the new differentiation capabilities of the compiler. Both results are written into files and compared by using the UNIX command `diff`. Identical numerical results are obtained by both the overloading and statement-level preaccumulation approaches. We regard the successful application of the compiler to the MINPACK test problems as a good indication for the desired robustness of the compilers differentiation capabilities.

5.2 Integration of Stiff Systems of Explicit ODEs

We consider the routine D02NBF that is part of the NAG Fortran 77 library.² It is a general-purpose routine for integrating the initial value problem for a stiff system of explicit ordinary differential equations (ODEs), $y' = g(t, y(t))$. The function g is nonlinear, and the Jacobian of g with respect to y is expected to be dense. The stiff Robertson problem is considered as an example in the documentation of the library that can be found on NAG's Web site. We present a modified version that illustrates the use of the differentiation-enabled compiler for computing the Jacobian of an approximation for the residual function $r(t, y) = y' - g(t, y)$ with respect to y . D02NBF can be provided with a subroutine that computes the Jacobian $\frac{\partial r}{\partial y}$. Alternatively, a finite difference method is used internally to approximate the Jacobian. Potentially, this method can be replaced by taking an approach similar to the one presented here.

²<http://www.nag.co.uk/numeric/FL/FLdescription.asp>

```

SUBROUTINE FCN(NEQ,T,Y,R,IRES)
USE ACTIVE_MODULE
DOUBLE PRECISION T
INTEGER      IRES, NEQ
DOUBLE PRECISION R(NEQ), Y(NEQ)

R(1) = -0.04D0*Y(1) + 1.0D4*Y(2)*Y(3)
R(2) = 0.04D0*Y(1) - 1.0D4*Y(2)*Y(3) - 3.0D7*Y(2)*Y(2)
R(3) = 3.0D7*Y(2)*Y(2)
RETURN
END

```

Fig. 5. Subroutine for computing $g(t, y)$ for the stiff Robertson problem

```

SUBROUTINE JAC(NEQ,T,Y,H,D,P)
USE ACTIVE_MODULE
INTERFACE
  SUBROUTINE FCN(NEQ,T,Y,R,IRES)
    DOUBLE PRECISION T
    INTEGER      IRES, NEQ
    DOUBLE PRECISION R(NEQ), Y(NEQ)
  END SUBROUTINE
END INTERFACE
DOUBLE PRECISION D, H, T
INTEGER      NEQ,IRES,I
DOUBLE PRECISION P(NEQ,NEQ), Y(NEQ), R(NEQ)
TYPE(JACOB_TYPE) :: J
DOUBLE PRECISION HXD

HXD=H*D
DIFFERENTIATE
  INDEPENDENT(Y)
  CALL FCN(NEQ,T,Y,R,IRES)
  J=JACOBIAN(R,Y)
END DIFFERENTIATE
CALL DERIVToreal(J,P)
P=-HXD*P
DO I=1,NEQ
  P(I,I)=P(I,I)+1
END DO
RETURN
END

```

Fig. 6. Wrapper for Jacobian routine for D02NBF

Given a subroutine for computing $g(t, y)$, as, for example, shown in Figure 5, the subroutine in Figure 6 represents a general wrapper that can be used for computing the Jacobian of r with respect to y by exploiting the new capabilities of the compiler. An explicit interface is required for FCN, and both the wrapper and the subroutine that computes the right-hand side of the ODE system need to use ACTIVE_MODULE. Additionally, the wrapper needs to perform some post-processing of the Jacobian that results from the approximation of y' in $r(t, y)$. Details on this can be found

```

DIFFERENTIATE
  INDEPENDENT(x, SEED=s)
  CALL FormFunctionLocal(...,x,f,...)
  jac = DERIVATIVE(f)
END DIFFERENTIATE

```

Fig. 7. Wrapper for Jacobian routine in PETSc. See www.mcs.anl.gov/petsc for further information on PETSc.

on NAG's Web site. Because of the static nature of the interface of FCN, the wrapper can be hidden entirely from the user provided that the library is used in connection with the differentiation-enabled compiler. The practical implementation of such a solution is envisioned.

5.3 Jacobians for the SNES Component of PETSc

The Portable and Extensible Toolkit for Scientific Computing (PETSc) [Balay et al. 2003] contains a module for the solution of systems of nonlinear equations (SNES) that relies on Jacobian matrices for the use in Newton-type algorithms. The user provides the routine `FormFunctionLocal` that implements the corresponding vector function. Additionally, code for computing the Jacobian matrix is required. The use of the AD tools ADIFOR³ and ADIC⁴ in this context is described in [Bischof et al. 1997]. A simple wrapper routine is required when using the new compiler, as shown schematically in Figure 7, where another important feature of the compiler is illustrated.

Assuming that `FormFunctionLocal` is based on some stencil that is used for a given discretization of a partial differential equation, the sparsity pattern of the Jacobian is known a priori. This knowledge can be exploited by the compiler. A compressed seed matrix `s` can be passed as a named parameter to the `INDEPENDENT` statement. The `ACTIVE_MODULE` subroutine `DERIVATIVE` returns a compressed Jacobian. Depending on the particular seeding method, the Jacobian can be restored by using a simple back substitution process [Curtis et al. 1974] or by solving a number of systems of linear equations [Newsam and Ramsdell 1983]. In either case the seed matrix compression is expected to lead to considerable speedup of the Jacobian computation.

As mentioned, the current implementation of using AD with PETSc is driven by ADIFOR and ADIC. In particular, the way in which first-derivative tensors are stored is not completely compatible with the solution provided by the compiler. This leads to some additional copying. We intend to avoid this overhead in a future version that should provide specific support for the differentiation-enabled compiler.

5.4 Gradients for TAO and NEOS

Our last case study represents an application that is not a primary target of the current version of the compiler. The computation of gradients for use in optimization algorithms often requires adjoint models that can be generated by the reverse mode

³<http://www.mcs.anl.gov/adifor>

⁴<http://www.mcs.anl.gov/adic>

```

SUBROUTINE fcn_wrapper(ndim, x, f, g)
USE ACTIVE_MODULE
IMPLICIT NONE

INTEGER ndim
DOUBLE PRECISION x(ndim)
DOUBLE PRECISION f
DOUBLE PRECISION g(ndim)

TYPE(JACOBI_TYPE) :: grad

DIFFERENTIATE
  INDEPENDENT(x)
  CALL fcn(ndim,x,f)
  grad=JACOBIAN(f)
END DIFFERENTIATE

CALL DERIVTOREAL(grad, g )

CONTAINS

#include "fcn.f"

END

```

Fig. 8. Wrapper for Jacobian routine in NEOS. The approach is similar to what has been discussed before. Since the name of the subroutine that contains the function evaluation is fixed, it can be included in the `CONTAINS` block of the wrapper routine that can be part of the solver library. Hence, a fully automated approach is possible. See www.mcs.anl.gov/neos for further information on NEOS.

of AD (see Section 6). With such a model available the gradient can be computed at a cost that is a small constant multiple of the cost of evaluating the function itself (see *cheap gradient result* in [Griewank 2000]). If the number of independent variables becomes very large, then the current solution is unlikely to provide acceptable performance. However, the new feature of the compiler can certainly be applied to small to medium-sized problems, for which the gradient could potentially be approximated by using finite difference quotients.

The new compiler has been tested successfully in the context of the the Toolkit for Advanced Optimization (TAO)⁵ and in the Network Enabled Optimization Server (NEOS).⁶ In particular, it has been used to provide gradients for the BLMVM algorithm for bound-constrained optimization [Benson and Moré 2001]. This solver is part of TAO and can be accessed via NEOS. An experimental NEOS server was set up to test the applicability of the new compiler. Assuming that Fortran is selected as input mode, NEOS works in principle as follows.

- (1) The user uploads files `fcn.f`, `initpt.f`, and `xbound.f` and specifies the dimension of the problem. The user also picks the AD tool to be used choosing, among ADIFOR 2.0 Revision D [Bischof et al. 1996], the differentiation-enabled

⁵<http://www.mcs.anl.gov/tao>

⁶<http://www.mcs.anl.gov/neos>

NAGWare Fortran 95 compiler, and the AD tool TAPENADE,⁷ which is being developed at INRIA Sophia-Antipolis, France, and which presents the only AD tool with reverse mode capabilities in this list. So far, only ADIFOR is accessible through the main NEOS server. It also provides second derivatives, a highly desirable feature in the context of optimization.

- (2) BLMVM provides a driver, for example, `tao_neos_driver.F`, that is linked with the solver library to run the optimization algorithm. The use of AD tools is also taken care of in the driver. For the new compiler we provide a wrapper routine, as shown in Figure 8.
- (3) BLMVM provides a `makefile` that builds the executable (e.g., `tao_neos_driver`) whose output is the result of the optimization.
- (4) The server copies all files into a temporary directory. It builds the executable and runs it, redirecting the output into the file `job.results`. The results are either emailed back to the user or displayed via the browser window. This build process is steered by a script.

The planned provision of reverse-mode algorithms inside the compiler will certainly improve its suitability in the context of optimization, as outlined in Section 6.2.

6. CONCLUSION AND FUTURE WORK

We presented a new version of the NAGWare Fortran 95 compiler that provides language extensions to facilitate the computation of first derivatives by an internal code transformation process that is built on the principles of automatic differentiation. The feasibility of this approach was demonstrated by using case studies involving various modern software libraries and systems for numerical computing. A major goal of this development was the construction of an intuitive and hierarchical interface that allows the user to exploit additional information such as sparsity of the Jacobian. We strongly believe that accurate derivative information generated by automatic differentiation should be accessible through an industrial-strength compiler. The user should be given the opportunity to use `JACOBIAN` and, potentially, `HESSIAN` and further derivative access functions in the same way the user uses other intrinsic functions. A first step has been made toward this ambitious goal.

In the following sections we briefly discuss two algorithms that we intend to provide next. Our objective is to demonstrate that most prerequisites have been put into place to facilitate an easier implementation. The existing infrastructure allows us to evolve to basic-block-level preaccumulation (see Section 6.1 and a simple version of the reverse mode of AD (see Section 6.2 without major technical difficulties).

6.1 Basic-Block-Level Preaccumulation

The preaccumulation procedure can be applied to extended Jacobians of entire basic blocks as described in [Griewank and Reese 1991; Naumann 2002a]. The current internal representation can be used without modification. Consider the following

⁷www-sop.inria.fr/tropics

sequence of scalar assignments:

$$v_1 = \cos(x_1); \quad v_2 = v_1 * x_2 \quad y = v_2 * x_3 \quad .$$

The internal version of the extended Jacobian is built statement by statement as follows:

$$\left(\begin{array}{c|c} x_1 & \\ \hline -\sin(x_1) & v_1 \end{array} \right), \left(\begin{array}{ccc|c} x_2 & x_1 & v_1 & \\ \hline & -\sin(x_1) & & v_1 \\ v_1 & 0 & x_2 & v_2 \end{array} \right), \left(\begin{array}{ccc|cc} x_3 & x_2 & x_1 & v_1 & v_2 \\ \hline & & -\sin(x_1) & & \\ v_1 & 0 & & x_2 & v_2 \\ v_2 & 0 & 0 & 0 & x_3 & y \end{array} \right).$$

Various heuristics for pivot selection can be applied. Once the Jacobian of a basic block $\mathbf{y} = F(\mathbf{x})$ has been accumulated, a module routine is called internally to perform the Jacobian matrix product $\dot{Y} = F' \cdot \dot{X}$. Similarly, the local Jacobians can be used for the adjoint version of the compiler to be developed (see Section 6.2). Adjoints of the inputs of the basic block can be computed as products of the transposed local Jacobian with the derivative component of the outputs of the basic block.

A major obstacle for the efficient use of basic-block-level preaccumulation is the absence of a *value-numbering* algorithm [Aho et al. 1986]. The result of such an algorithm tells us that, for example, the v_1 that is written by the first assignment is read by the second assignment. In the presence of pointers and array accesses, this information is not obvious. It is required to assign the correct virtual addresses to the various variable references. Conservatively, one needs to assume that all references are potentially distinct, an assumption that leads to a statement-level algorithm as described in Section 4.

6.2 Adjoint Code by Reverse Mode

The adjoint code generated by the reverse vector mode of AD for a computer program implementing a vector function $\mathbf{y} = F(\mathbf{x})$ with $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$ computes Jacobian transposed times matrix products of the form

$$\bar{X} = (F')^T \bar{Y} \quad . \quad (6)$$

The matrix $\bar{Y} \in \mathbb{R}^{m \times l}$ represents a collection of l adjoints of the dependent variables $\mathbf{y} \in \mathbb{R}^m$. The result of executing the adjoint code is a matrix $\bar{X} \in \mathbb{R}^{n \times l}$ with columns representing the corresponding adjoint values of the independent variables $\mathbf{x} \in \mathbb{R}^n$. If F is scalar (i.e., $m = 1$) then the whole gradient can be obtained cheaply as pointed out before. This feature makes adjoint codes especially interesting for applications in optimization, optimal control, and data assimilation [Singer and Barton 2003; Caillaud and Noailles 2001; Griffith and Nichols 1996].

In reverse mode, adjoint values are propagated backward for all intermediate and the independent variables after initializing them for the dependent variables. The value of some \bar{v}_i is computed as

$$\bar{v}_i = \sum_{j:i \prec j} \frac{\partial v_j}{\partial v_i} \cdot \bar{v}_j \quad . \quad (7)$$

While the local partial derivatives (and therefore the directional gradients as well) could be computed in parallel with the function value in forward mode, they must

be made available in reverse order in adjoint models. Therefore, one may decide to store and restore or to recompute them when they become required. Alternatively, one may choose to store the values of the intermediate variables or even to preaccumulate local gradients of assignments or local Jacobians of basic blocks. Further improvements are discussed in [Naumann 2002c]. A single forward sweep is required to store all the values required for a store-all strategy. A recompute-all approach would increase the computational complexity quadratically and is not preferred. Checkpointing algorithms store a certain amount of information and recompute whatever else is required using this information. The automatic detection of useful checkpoints in a general program is an open problem.

The main challenge for developers of reverse mode AD algorithms is the requirement to reverse the control flow of the program efficiently, thus making the local partial derivatives available in reverse order. For large-scale applications the term *efficiently* is defined as a reasonable trade-off between the number of arithmetic operations performed and the amount of memory required.

As a first step we plan to store the local gradients of all scalar assignments on a stack and restore them during the reverse pass to compute the adjoints of the inputs of the assignment incrementally⁸ as a function of the adjoint of its left-hand side. Similarly, we can store and restore the Jacobians of basic blocks. Further improvements of such an approach represent the storage of intermediate values [Hascoët et al. 2002] as well as checkpointing strategies. In any case we need to think carefully about the design of the user interface, which is supposed to remain as simple and intuitive as possible.

REFERENCES

- AHO, A., SETHI, R., AND ULLMAN, J. 1986. *Compilers. Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA.
- AVERIK, B., CARTER, R., AND MORÉ, J. 1991. The MINPACK-2 test problem collection (preliminary version). Tech. Rep. 150, Mathematical and Computer Science Division, Argonne National Laboratory.
- BALAY, S., BUSCHELMAN, K., GROPP, W., KAUSHIK, D., KNEPLEY, M., CURFMAN-MCINNIS, L., SMITH, B., AND ZHANG, H. 2003. PETSc 2.0 users manual. Tech. Rep. ANL-95/11 - Revision 2.1.6, Argonne National Laboratory. Aug. See <http://www.mcs.anl.gov/petsc>.
- BENSON, S. AND MORÉ, J. 2001. A limited memory variable metric algorithm for bound constrained minimization. Tech. Rep. ANL/MCS-P909-0901, Mathematics and Computer Science Division, Argonne National Laboratory.
- BERZ, M., BISCHOF, C., CORLISS, G., AND GRIEWANK, A., Eds. 1996. *Computational Differentiation: Techniques, Applications, and Tools*. Proceedings Series. SIAM, Philadelphia.
- BISCHOF, C., CARLE, A., KHADEMI, P., AND MAURER, A. 1996. The ADIFOR 2.0 system for Automatic Differentiation of Fortran 77 programs. *IEEE Comp. Sci. & Eng.* 3, 3, 18–32.
- BISCHOF, C., HOVLAND, P., AND WU, P.-T. 1997. Using ADIFOR and ADIC to provide a Jacobian for the SNES component of PETSc. Technical Memorandum ANL/MCS-TM-233, Mathematics and Computer Science Division, Argonne National Laboratory.
- CAILLAU, J.-B. AND NOAILLES, J. 2001. Optimal control sensitivity analysis with AD. In *Automatic Differentiation of Algorithms: From Simulation to Optimization*, G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, Eds. Computer and Information Science. Springer, New York, Chapter 11, 109–115.

⁸See [Griewank 2000] for details on the incremental version of the reverse mode.

- COHEN, M., NAUMANN, U., AND RIEHME, J. 2003. Towards differentiation-enabled Fortran 95 compiler technology. In *Proceedings of the 2003 ACM Symposium on Applied Computing*. ACM.
- CORLISS, G., FAURE, C., GRIEWANK, A., HASCOET, L., AND NAUMANN, U., Eds. 2002. *Automatic Differentiation of Algorithms – From Simulation to Optimization*. Springer, New York.
- CORLISS, G. AND GRIEWANK, A., Eds. 1991. *Automatic Differentiation: Theory, Implementation, and Application*. Proceedings Series. SIAM, Philadelphia.
- CURTIS, A., POWELL, M., AND REID, J. 1974. On the estimation of sparse Jacobian matrices. *J. Inst. Math. Appl.*
- GRIEWANK, A. 2000. *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Applied Mathematics. SIAM, Philadelphia.
- GRIEWANK, A. AND REESE, S. 1991. On the calculation of Jacobian matrices by the Markovitz rule. In *[Corliss and Griewank 1991]*. SIAM, 126–135.
- GRIFFITH, A. AND NICHOLS, N. 1996. Accounting for model error in data assimilation using adjoint models. In *[Berz et al. 1996]*. SIAM, 195–204.
- HASCOËT, L., NAUMANN, U., AND PASCUAL, V. 2002. TBR analysis in reverse-mode Automatic Differentiation. Preprint ANL-MCS/P936-0202, Mathematics and Computer Science Division, Argonne National Laboratory.
- NAUMANN, U. 2002a. Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph. Preprint ANL-MCS/P943-0402, Mathematics and Computer Science Division, Argonne National Laboratory. To appear in *Math. Prog.*
- NAUMANN, U. 2002b. Optimal pivoting in tangent-linear and adjoint systems of nonlinear equations. Preprint ANL-MCS/P944-0402, Mathematics and Computer Science Division, Argonne National Laboratory.
- NAUMANN, U. 2002c. Statement-level optimality of tangent-linear and adjoint models. Preprint ANL-MCS/P1021-0103, Mathematics and Computer Science Division, Argonne National Laboratory.
- NAUMANN, U. AND GOTTSCHLING, P. 2003. Simulated annealing for optimal pivot selection in Jacobian accumulation. In *Stochastic Algorithms, Foundations and Applications – SAGA’03*, A. Albrecht and K. Steinhöfel, Eds. Number 2827 in LNCS. Springer, Berlin, 83–97.
- NEWSAM, G. AND RAMSDELL, J. 1983. Estimation of sparse Jacobian matrices. *SIAM J. Alg. Dis. Meth.* 4, 404–417.
- SINGER, A. AND BARTON, P. 2003. Global solution of linear dynamic embedded optimization problems, part I: Convex problems. *Journal of Optimization Theory and Applications*. In press.

Received ...; revised ...; accepted ...

<p>The submitted manuscript has been created by the University of Chicago as Operator of Argonne National Laboratory ("Argonne") under Contract No. W-31-109-ENG-38 with the U.S. Department of Energy. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.</p>
--