

# A Differentiation-Enabled Fortran 95 Compiler

Uwe Naumann

Software and Tools for Computational Engineering, RWTH Aachen University  
52056 Aachen, Germany

and

Jan Riehme

Department of Mathematics, Humboldt University Berlin  
Unter den Linden 6, 10099 Berlin, Germany

---

The availability of first derivatives of vector functions is crucial for the robustness and efficiency of a large number of numerical algorithms. A new version of the differentiation-enabled NAGWare Fortran 95 compiler is described that uses programming language extensions and a semantic code transformation known as automatic differentiation to provide Jacobians of numerical programs with machine accuracy. We describe a new improved user interface as well as the relevant algorithmic details. In particular, we focus on the source transformation approach that generates locally optimal gradient code for single assignments by vertex elimination in the linearized computational graph. Extensive tests show the superiority of this method over the current overloading-based approach. The robustness and convenience of the new compiler-feature is illustrated by various case studies.

Categories and Subject Descriptors: G.1.4 [Quadrature and Numerical Differentiation]: Automatic Differentiation; D.3.4 [Processors]: Compilers

General Terms: Algorithms

Additional Key Words and Phrases: Source transformation, Preaccumulation

---

## 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,  $\mathbf{f}(\mathbf{x}, \mathbf{y})$  that computes  $\mathbf{y}$  as a function of  $\mathbf{x}$ . This subroutine may be very complex, possibly involving calls to other user-defined subroutines or functions. Suppose that the routine `nag_nlin_sys_sol`, that is part of the NAG Fortran 90 Library [The Numerical Algorithms Group 2000], is to be applied to this problem. The solver expects a subroutine  $\mathbf{g}(\mathbf{x}, \mathbf{finish}, \mathbf{y}, \mathbf{dydx})$  that computes the function value  $\mathbf{y}$  and, ideally, the Jacobian  $\mathbf{dydx}$ . Setting the parameter `finish` to `true` inside of  $\mathbf{g}$  tells `nag_nlin_sys_sol` to finish calling  $\mathbf{g}$  as part of the solution process. Our goal is to make the subroutine  $\mathbf{g}$  a simple “generic wrapper” calling  $\mathbf{f}$  such that the derivative code is generated automatically by the compiler.

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 g(x,finish,y,dydx)
  USE ACTIVE_MODULE
  ...
  REAL, DIMENSION(:), INTENT(in) :: x
```

```

REAL, DIMENSION(:), INTENT(out) :: y
REAL, DIMENSION(:, :), ALLOCATABLE, INTENT(out) :: dydx

DIFFERENTIATE
  INDEPENDENT(x)
  CALL f(x,y)
  DEPENDENT(y,DERIVATIVE=dydx)
END DIFFERENTIATE
...
END SUBROUTINE g

```

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 `dydx` of `y` with respect to `x` can be obtained with machine accuracy by passing it as a named parameter to the `DEPENDENT` statement. Both `f` and `g` are compiled with the differentiation-enabled 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 and their implementation. Following a description of the overall approach taken (Section 2), we concentrate in Section 3 on the relevant compiler internals. 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 4 we present several case studies that underline the flexibility and ease of use of the compiler. The improvements in terms of runtime savings due to the new source transformation method are illustrated in Section 5. Conclusions are drawn in Section 6.

## 2. APPROACH

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 differentiation (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, possibly, 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 a reasonable approach. Language extensions, such as those presented in Section 1, give a mechanism to support the use of AD in scientific computation but with most of the AD internals hidden from the user.

In this paper we apply AD to computer programs written in Fortran 95. For ACM Transactions on Mathematical Software, Vol. V, No. N, September 2004.

illustrative purposes we consider a very simple part of a larger program as an example. This section of the code implements a vector function

$$F : \mathbb{R}^2 \rightarrow \mathbb{R} : \mathbf{y} = F(\mathbf{x}(i-1), \mathbf{x}(i))$$

as

```

y(i+1)=sin(x(i-1))*x(i-1)/x(i)
if (i<j) then
  y(i+1)=exp(y(i+1))
else
  y(i+1)=y(i+1)*(2.0*x(i))
end if

```

For given values of  $i$  and  $j$  the flow of control through this section of the code is determined and  $F$  can be represented as a sequence of assignments. For example, if  $i=j$ , then  $F$  is computed as

$$\begin{aligned} \mathbf{y}(i+1) &= \sin(\mathbf{x}(i-1)) * \mathbf{x}(i-1) / \mathbf{x}(i) \\ \mathbf{y}(i+1) &= \mathbf{y}(i+1) * (2.0 * \mathbf{x}(i)) \end{aligned} \quad (1)$$

AD is based on the assumption that at a given argument the evaluation routine of  $F$  can be decomposed into a sequence of even simpler assignments of the results of scalar *elemental functions*  $\varphi_j$  to unique variables  $v_j$  according to

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

Following the notation in [Griewank 2000]  $i \prec j$  denotes a direct dependence of  $v_j$  on  $v_i$ , that is, there is an assignment to  $v_j$  with  $v_i$  on the right-hand side. For the first statement in our example we obtain

$$\begin{aligned} v_0 &= \mathbf{x}(i-1); \quad v_{-1} = \mathbf{x}(i) \\ v_1 &= \sin(v_0); \quad v_2 = v_1 * v_0; \quad v_3 = v_2 / v_{-1} \\ \mathbf{y}(i+1) &= v_3 \quad . \end{aligned}$$

The *independent* variables  $\mathbf{x}(i-1)$  and  $\mathbf{x}(i)$  are enumerated decreasingly starting with  $v_0$ . Two intermediate variables,  $v_1$  and  $v_2$ , are introduced and the result is stored in the dependent variable  $v_3$  that is equivalent to  $\mathbf{y}(i+1)$ . This representation is often referred to as the *code list*. It induces a directed acyclic computational graph  $\mathbf{G} = (V, E)$  as shown in Figure 1 (a) for the two assignments from Equation (1). Vertices in  $V$  represent the code list variables and edges in  $E$  their mutual dependences according to  $(i, j) \in E \Leftrightarrow i \prec j$ . The values of  $v_3$  and  $v_5$  are assigned to  $\mathbf{y}[i+1]$ , respectively.

The  $\varphi_j$  are assumed to have jointly continuous partial derivatives

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

with respect to all of their arguments  $(v_i)_{i \prec j}$  on open neighborhoods of their respective domains. Moreover we assume that we know how to compute them. In particular, there are well-defined expressions for the partial derivatives of the arithmetic

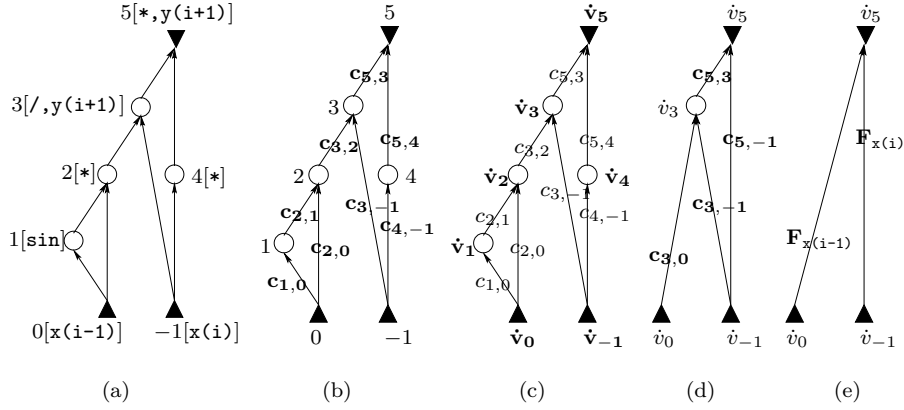


Fig. 1. From Computational Graphs to Gradients

operators and intrinsic functions provided by Fortran 95. Under these assumptions every single assignment in the code list can be augmented with statements to compute the respective partial derivatives. For example,

$$\begin{aligned} c_{1,0} &= \cos(v_0); & v_1 &= \sin(v_0) \\ c_{2,1} &= v_0; & c_{2,0} &= v_1; & v_2 &= v_1 * v_0 \\ c_{3,2} &= 1/v_{-1}; & c_{3,-1} &= -v_2/(v_{-1} * v_{-1}); & v_3 &= v_2/v_{-1} \quad . \end{aligned}$$

The augmented code list is said to be *linearized*. The  $c_{j,i}$  can be regarded as edge labels in  $\mathbf{G}$  as shown in Figure 1 (b).

The forward mode of AD transforms the program for  $F$  into a tangent-linear model such that

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

$\dot{\mathbf{y}} \in \mathbb{R}^m$  is the directional derivative of  $F$  in the direction  $\dot{\mathbf{x}} \in \mathbb{R}^n$ . Its numerical values are computed with machine accuracy. It is well known that the Jacobian matrix

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

can be accumulated by letting  $\dot{\mathbf{x}}$  range over the Cartesian basis vectors in  $\mathbb{R}^n$ .

The transformation into a tangent-linear model is based on the linearized code list. Directional derivatives  $\dot{v}_i$  are associated with every code list variable for  $i = 1 - n, \dots, q$ . These inner products of the gradients of the  $v_i$  with respect to the independent variables with the specified direction in the input space  $\dot{\mathbf{x}}$  are propagated forward as

$$\dot{v}_j = \sum_{i < j} c_{j,i} \cdot \dot{v}_i \quad , \quad (2)$$

for  $j = 1, \dots, q$ . The tangent-linear code that can be generated for the first state-

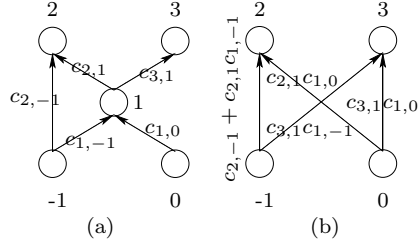


Fig. 2. Vertex Elimination. The elimination of vertex 1 in (a) yields the graph in (b). Four multiplications and one addition are performed.

ment of the example is similar to the following:

$$\begin{aligned}
 c_{1,0} &= \cos(v_0); & v_1 &= \sin(v_0); & \dot{v}_1 &= c_{1,0} * \dot{v}_0 \\
 c_{2,1} &= v_0; & c_{2,0} &= v_1; & v_2 &= v_1 * v_0; & \dot{v}_2 &= c_{2,1} * \dot{v}_1 + c_{2,0} * \dot{v}_0 \\
 c_{3,2} &= 1/v_{-1}; & c_{3,-1} &= -v_2/(v_{-1} * v_{-1}); & v_3 &= v_2/v_{-1}; & \dot{v}_3 &= c_{3,2} * \dot{v}_2 + c_{3,-1} * \dot{v}_{-1} .
 \end{aligned}$$

Alternatively, the computation of directional derivatives can be regarded as the application of the chain rule to  $\mathbf{G}$  as illustrated in Figure 1 (c). For given values of  $\dot{v}_i$  for  $i = 1 - n, \dots, 0$  and  $c_{j,i}$  for  $i, j = 1 - n, \dots, q$ ,  $i < j$ , the  $\dot{v}_j$  are computed for  $j = 1, \dots, q$  as sums over all incoming edges of products of the respective labels with the value of the directional derivative of the edge's source vertex.

An alternative approach to the application of the chain rule to linearized computational graphs has been presented in [Griewank and Reese 1991] and extended in [Naumann 2004]. The accumulation of the Jacobian of a vector function  $F$  can be regarded as a sequence of vertex eliminations and corresponding creations / modifications of edge labels in  $\mathbf{G}$ . A vertex  $j$  is eliminated by creating new edges  $(i, k)$  that connect all its predecessors  $i \in P_j$  with its successors  $k \in S_j$ . Values of the new edge labels are computed as  $c_{k,i} = c_{k,j} \cdot c_{j,i}$ . Labels of already existing edges are incremented as  $c_{k,i} := c_{k,i} + c_{k,j} \cdot c_{j,i}$ . The vertex  $j$  is removed from  $\mathbf{G}$  together with all its incident edges. The number of scalar floating-point multiplications involved in the elimination of a vertex  $j$  is equal to  $|P_j| \cdot |S_j|$ . Furthermore,  $|S_j \cap S_i| \equiv |P_j \cap P_k|$  additions are performed. Vertex elimination is illustrated in Figure 2. A complete vertex elimination sequence transforms  $\mathbf{G}$  into a bipartite graph such that the labels on the remaining edges are exactly the nonzero entries of  $F'$  at the current argument. This situation is shown in Figure 1 (e) for the gradient of our simple example program, where  $\mathbf{F}_{\mathbf{x}(i-1)}$  and  $\mathbf{F}_{\mathbf{x}(i)}$  denote the partial derivatives of  $\mathbf{F}$  with respect to  $\mathbf{x}(i-1)$  and  $\mathbf{x}(i)$ , respectively.

The computation of an optimal vertex elimination sequence that minimizes the number of floating-point operations is conjectured to be np-hard (see [Griewank and Reese 1991; Bischof and Haghghat 1996]). In [Naumann 2002] we present a solution to the optimal vertex elimination problem for programs that exhibit the *single expression use* property. In the corresponding computational graphs all intermediate vertices have exactly one predecessor. In particular, right-hand sides of scalar assignments fit into this category. This fact is exploited inside the compiler to generate efficient code for preaccumulating the gradients of all scalar

assignments. For Equation (1) the vertex elimination sequence (1, 2, 4) gives

$$\begin{aligned} c_{3,0} &= c_{3,2}(c_{2,0} + c_{2,1} \cdot c_{1,0}) \\ c_{5,-1} &= c_{5,4} \cdot c_{4,-1} \quad . \end{aligned}$$

The resulting graph is shown in Figure 1 (d). The gradients of the first and the second assignments are  $(c_{3,0}, c_{3,-1})^T$  and  $(c_{5,3}, c_{5,-1})^T$ , respectively.

The following remarks should be made.

- (1) The AD tool ADIFOR [Bischof et al. 1996] uses statement-level reverse mode to preaccumulate gradients of scalar assignments. This approach is essentially equivalent to reverse vertex elimination in the corresponding computational graph. The number of multiplications required exceeds the minimal number used by our algorithm by at most a factor of two [Naumann 2002].
- (2) Linear parts of the program may lead to constant labels of incident edges that are eliminated by the compiler as described in [Utke and Naumann 2004a]. Alternatively, the constant folding can be left to the optimization phase of the compiler back-end since no combinatorial search algorithm is used to compute an optimal elimination sequence. Search space reduction is not an issue.
- (3) The entire computational graph of Equation (1) has the single expression property. Optimal vertex elimination can be applied globally provided that the graph can be flattened as described in [Utke 2004]. In the presence of ambiguities arising from pointers and array accesses (see *aliasing* in [Aho et al. 1986]) the static construction of dags for basic blocks is a non-trivial task.
- (4) So far, it remains unclear whether a low number of arithmetic operations translates into a decreased runtime. If the elimination sequence yields a highly irregular memory access pattern, then improvements cannot be expected. First results on this topic have been published in [Tadjouddine et al. 2002]. In the present context the limited size of the graphs under consideration (How large do right-hand sides of assignments in numerical codes usually become?) guarantees data locality to some extent.

For the computation of several directional derivatives at the same point the *vector forward mode* of AD can be used to transform  $F$  into a tangent-linear model such that

$$\dot{Y} = F'(\mathbf{x})\dot{X} \quad . \quad (3)$$

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 (2) 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. Preaccumulation becomes particularly useful for  $l \gg 1$ . For a right-hand side whose computational graph has  $n$  input vertices and  $|E|$  edges the cost of propagating  $l$  directional derivatives in vector forward mode is  $l \cdot |E|$  compared to  $l \cdot n + M$ , where  $M$  is the cost of accumulating the local gradient. In practice, the latter is compensated for quickly as  $|E| \geq n$ .

To put the pieces together we present the tangent-linear code that is generated conceptually by the compiler for the first statement in Equation (1). Lines are

numbered for easier reference.

```

[1]          v0 = x(i-1)%v;  v-1 = x(i)%v
[2]          c1,0 = cos(v0);  v1 = sin(v0)
[3]          c2,1 = v0;  c2,0 = v1;  v2 = v1 * v0
[4]  c3,2 = 1/v-1;  c3,-1 = -v2/(v-1 * v-1);  v3 = v2/v-1
[5]          y(i+1)%v = v3
[6]          c3,0 = c3,2(c2,0 + c2,1 · c1,0)
[7]          v̇0 = x(i-1)%d;  v̇-1 = x(i)%d
[8]          v̇3 = c3,-1 * v̇-1 + c3,0 * v̇0
[9]          y(i+1)%d = v̇3

```

Both  $\mathbf{x}$  and  $\mathbf{y}$  are vectors of a user-defined type with components  $\mathbf{v}$  (for the local function value) and  $\mathbf{d}$  (for the value of the local directional derivative) as described in Section 3. The linearized code list is constructed by the statements on lines [1]-[5]. Elimination of vertex 1 followed by 2 in Figure 1 (b) leads to the Jacobian accumulation code on line [6]. The inner product  $(\dot{v}_0, \dot{v}_{-1}) \cdot (c_{3,0}, c_{3,-1})^T$  is computed on lines [7]-[9]. The copies on lines [1],[5],[7], and [9] have been inserted for better readability. They need not be performed explicitly. In any case, optimizing compilers eliminate such copies as part of their copy propagation phase [Aho et al. 1986].

### 3. IMPLEMENTATION

The first prototype of the differentiation-enabled compiler (from now on referred to as *the 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 functions are overloaded for this new data type, not only to compute the function value but also to perform the computation of the directional derivatives. About 2000 lines of code inside of a Fortran 90 module are required in addition to some modifications of the compiler internals to make the overloading-based approach work.

The active data type contains a component to hold the function value  $\mathbf{v}$  and an allocatable array of directional derivatives  $\mathbf{d}$ .

```

TYPE ACTIVE_TYPE
  DOUBLE PRECISION :: v
  DOUBLE PRECISION, ALLOCATABLE, DIMENSION(:) :: d
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 variables are known. The active data type is used both in the context of operator overloading and source transformation. Refer to [Utke and Naumann 2004b] for a discussion of issues arising from the use of *association by name* via active data types in the semantic transformation of numerical programs.

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

- (1) The module `ACTIVE_MODULE` needs to be used.
- (2) An allocatable variable (for example, `dydx`) needs to be declared to hold the Jacobian matrix.
- (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” within the active section. In Jacobian computation mode the derivative components are initialized with the identity in  $\mathbb{R}^{n \times n}$ .
- (5) The `DEPENDENT` statement is used to access the derivative component(s) of the active variable that forms its argument. The result is stored in the variable that is assigned to the named parameter `DERIVATIVE`, that is, `dydx`. The partial derivative  $\frac{\partial y_i}{\partial x_j}$  is stored in `dydx(i, j)`.

In the current version of the compiler, no static data-flow analysis is performed. Conservatively, all floating-point 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. 2004].

The preaccumulation of the local gradients of all scalar assignments is implemented as an elimination procedure on a customized version of the tangent-linear system as follows. A tangent-linear model  $\dot{y} = F' \cdot \dot{x}$  of a scalar assignment represents a system of linear equations that can be written in matrix form as follows:

$$\begin{pmatrix} \dot{z} \\ \dot{y} \end{pmatrix} = C \cdot \begin{pmatrix} \dot{x} \\ \dot{z} \end{pmatrix}, \quad (4)$$

where  $C \in \mathbb{R}^{(p+1) \times (n+p)}$  is defined as

$$C = (c_{j,i})_{\substack{j=1,\dots,p+1 \\ i=1-n,\dots,p}} \quad (5)$$

with local partial derivatives  $c_{j,i}$ . The computation of  $\dot{y}$  can be interpreted as the solution of Equation (4) for  $\dot{y}$  in terms of  $\dot{x}$  by eliminating all dependences of  $\dot{y} \in \mathbb{R}$  on the intermediate variables  $\dot{z} \in \mathbb{R}^p$ . Griewank [Griewank 2000] presents an approach to the accumulation of  $F'$  by Gaussian elimination on the extended Jacobian  $C - I$ . Our approach is similar in that the equivalent to vertex elimination is applied to  $C$ .

Consider the first assignment in Equation (1). Shift-reduce parsing [Aho et al. 1986] processes the right-hand side from left to right. Enumeration of the independent variables starting with  $v_0$  decreasingly and of the intermediate and dependent variables starting with  $v_1$  increasingly allows for an incremental approach to building  $C$  while parsing the expression as follows.

<code>x(i-1)</code>	$\rightarrow v_0$ (add column 0)
<code>sin(x(i-1))</code>	$\rightarrow v_1$ (add row/column 1; $c_{1,0}$ )
<code>(x(i-1))</code>	$\rightarrow v_0$
<code>sin(x(i-1))*x(i-1)</code>	$\rightarrow v_2$ (add row/column 2; $c_{2,1}$ and $c_{2,0}$ )
<code>x(i)</code>	$\rightarrow v_{-1}$ (add column -1)
<code>sin(x(i-1))*x(i-1)/x(i)</code>	$\rightarrow v_3$ (add row 3; $c_{3,2}$ and $c_{3,-1}$ )

The vertex elimination problem in the linearized computational graph is equivalent to solving the tangent-linear system

$$\begin{pmatrix} \dot{v}_1 \\ \dot{v}_2 \\ \dot{v}_3 \end{pmatrix} = \begin{pmatrix} 0 & c_{1,0} & 0 & 0 \\ 0 & c_{2,0} & c_{2,1} & 0 \\ c_{3,-1} & 0 & 0 & c_{3,2} \end{pmatrix} \cdot \begin{pmatrix} \dot{v}_{-1} \\ \dot{v}_0 \\ \dot{v}_1 \\ \dot{v}_2 \end{pmatrix}$$

for  $\dot{v}_3$  in terms of  $\dot{v}_{-1}$  and  $\dot{v}_0$  by eliminating the dependence of  $\dot{v}_3$  on  $\dot{v}_1$  and  $\dot{v}_2$ . The optimal vertex elimination sequence [1, 2] is equivalent to the transformation of the system into

$$\begin{pmatrix} \dot{v}_2 \\ \dot{v}_3 \end{pmatrix} = \begin{pmatrix} 0 & c_{2,0} + c_{2,1} \cdot c_{1,0} & 0 \\ c_{3,-1} & 0 & c_{3,2} \end{pmatrix} \cdot \begin{pmatrix} \dot{v}_{-1} \\ \dot{v}_0 \\ \dot{v}_2 \end{pmatrix}$$

and

$$\dot{v}_3 = (c_{3,-1} \ c_{3,2} \cdot (c_{2,0} + c_{2,1} \cdot c_{1,0})) \cdot \begin{pmatrix} \dot{v}_{-1} \\ \dot{v}_0 \end{pmatrix} \quad (6)$$

as described in [Naumann 2002]. The accumulation of  $\dot{v}_3$  can be performed incrementally as

$$\begin{aligned} \dot{v}_3 &= c_{3,-1} \cdot \dot{v}_{-1} \\ \dot{v}_3 &= \dot{v}_3 + c_{3,2} \cdot (c_{2,0} + c_{2,1} \cdot c_{1,0}) \cdot \dot{v}_0 \end{aligned}$$

by several calls of a saxpy routine that is defined in `ACTIVE_MODULE`. For large local gradients the repeated subroutine calls are likely to slow down the execution of the derivative code. To overcome this problem we use various propagation routines for different sizes of the local gradients (see also Section 5).

The row vector in Equation (6) is equal to the transposed gradient of  $v_3$  with respect to  $v_{-1}$  and  $v_0$ . The number of partial derivatives required for the propagation of the directional derivative is decreased from  $|E| = 5$  to  $n = 2$ . The preaccumulation itself costs two multiplications and one addition. For large values of  $l$  in Equation (3), the local savings converge to a factor of  $2.5 = \lim_{l \rightarrow \infty} \frac{5 \cdot l}{2 \cdot l + 2}$ .

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 linearized code list as described in Section 2.

The NAGWare Fortran 95 front-end transforms the Fortran code into C 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. Furthermore, the differentiation phase of the compiler can generate output that illustrates the single steps performed by the preaccumulation algorithm.

#### 4. CASE STUDIES

To test the correctness of the derivative code generated by the compiler, we implemented a test environment for fifteen vector functions from the MINPACK-2 test problem collection [Averik et al. 1991]. In all cases the numerical values computed by the compiler-generated derivative code match those of the hand-written Jacobian code that is part of MINPACK-2. 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-2 test problems as a good indication of the desired robustness of the compiler's differentiation capabilities. A performance analysis of the compiler for the *flow in a driven cavity* problem is presented in Section 5.2.

##### 4.1 Integration of Stiff Systems of Explicit ODEs

The routine D02NBF that is part of the NAG Fortran 77 library [The Numerical Algorithms Group 2002; Hopkins and Phillips 1988] 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 [The Numerical Algorithms Group 2002]. We have used the differentiation-enabled compiler successfully 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. We investigate the possible combination of the NAG Fortran 77/90 libraries with the compiler to make automatic differentiation the default method for computing derivatives that are required in a variety of numerical algorithms, the integration of stiff systems of explicit ODEs being an example.

##### 4.2 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 ADIFOR [Bischof et al. 1996] and ADIC [Bischof et al. 1997] 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 3, 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 seed matrix  $\mathbf{s}$  can be passed as a named parameter to the `INDEPENDENT` statement. The variable assigned to the `DERIVATIVE` parameter of the `DEPENDENT` statement contains a compressed Jacobian. Depending on the particular seeding method, the Jacobian can be restored by using a simple back substitution process [Curtis et al.

```

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

```

Fig. 3. Wrapper for Jacobian routine in PETSc. See [www.mcs.anl.gov/petsc](http://www.mcs.anl.gov/petsc) for further information on PETSc.

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 illustrated in Section 5.2.

### 4.3 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 of AD. Recently, a first experimental version has been added to the compiler as described in [Naumann and Riehme 2004]. 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-size 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)[Benson et al. 2000] and in the Network Enabled Optimization Server (NEOS)[Moré 2001]. 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.

## 5. PERFORMANCE TESTS

To verify the runtime improvement due to preaccumulation compared with pure overloading (denoted OL in the tables below) we run a number of tests on practically relevant problems. We observe the savings in user time obtained by using preaccumulation combined with propagation routines for the directional derivatives for local gradients of size  $k$  (denoted P  $k$ ) for increasing values for  $k$ . Additionally we keep track of the cpu usage that is defined as the ratio between the sum of user and system times and the total elapsed time. Standard compiler optimization (-O) is used.

### 5.1 Roe-Flux

Our first test problem defines the numerical fluxes of mass, energy, and momentum across a cell face in a finite volume compressible flow calculation. Roe's numerical flux [Roe 1981] takes  $n = 10$  inputs describing the flow on either side of a cell. It returns  $m = 5$  outputs for the numerical flux. The  $5 \times 10$  Jacobian is computed 10000 times by forward propagation of the  $10 \times 10$  identity. A comparably large

	$n$	saved	user	cpu
OL	10		3.49	98.67%
P 1	10	<b>24.83%</b>	2.62	98.00%
P 2	10	<b>30.66%</b>	2.42	98.00%
P 6	10	<b>33.52%</b>	2.32	98.33%
P 10	10	<b>34.00%</b>	2.30	98.00%

Table I. Roe's Flux

	$n$	saved	user	cpu	$F+$	$F-$
OL	400		0.30	96.00%	0	3362
P 1	400	<b>32.97%</b>	0.20	95.00%	0	3373
P 2	400	<b>30.77%</b>	0.21	97.00%	0	3370
P 5	400	<b>30.77%</b>	0.21	96.33%	0	3370
OL	900		1.65	99.00%	0	15955
P 1	900	<b>35.56%</b>	1.06	98.67%	0	15715
P 2	900	<b>33.94%</b>	1.09	99.00%	0	15712
P 5	900	<b>34.34%</b>	1.08	97.33%	0	15712
OL	1600		6.15	98.00%	2	49623
P 1	1600	<b>36.40%</b>	3.91	99.00%	0	49126
P 2	1600	<b>36.13%</b>	3.93	99.00%	0	49123
P 5	1600	<b>35.64%</b>	3.96	99.00%	0	49123
OL	2025		10.27	23.67%	7010	168412
P 1	2025	<b>37.74%</b>	6.40	17.00%	7133	167930
P 2	2025	<b>36.86%</b>	6.49	18.00%	7308	169975
P 5	2025	<b>36.31%</b>	6.54	18.33%	7007	167997

Table II. Flow in a Driven Cavity: Accumulation of the Jacobian

number of evaluations is performed in practical situations too. Table I summarizes our observations.

The separate computation of the scalar contributions to the inner product that defines the local directional derivative in mode “P 1” reduces the potential savings considerably. On average savings of about 30% can be observed. The contribution to the runtime due to the propagation of the directional derivatives is significant. Future development will target a possible decrease of this overhead by avoiding the call of module routines. Explicit code can be generated instead.

An in-depth investigation of various Jacobian accumulation techniques and AD tools applied to Roe's flux can be found in [Tadjouddine et al. 2001].

## 5.2 Flow in a Driven Cavity

The second test problem is from the MINPACK-2 test problem collection [Averik et al. 1991]. The 2D flow in a driven cavity is formulated as a boundary value problem, which is discretized by standard finite difference approximations to obtain a system of nonlinear equations. We choose equal numbers of steps in both the  $x$ - and  $y$ -directions. The number of independent variables is equal to the number of mesh points that are not part of the boundary.

The results for accumulation of the whole Jacobian without exploiting sparsity are listed in Table II. Again, a speedup of more than 30% can be observed. The

	$n$	$l$	saved	user	cpu	$F+$	$F-$
OL	400	81		0.12	95.00%	0	982
P 1	400	81	<b>36.11%</b>	0.08	92.67%	0	986
P 2	400	81	<b>38.89%</b>	0.07	92.67%	0	983
P 5	400	81	<b>36.11%</b>	0.08	92.33%	0	983
OL	900	121		0.34	97.00%	0	2828
P 1	900	121	<b>28.43%</b>	0.24	96.33%	0	2832
P 2	900	121	<b>27.45%</b>	0.25	97.00%	0	2829
P 5	900	121	<b>27.45%</b>	0.25	97.00%	0	2829
OL	1600	161		0.79	98.67%	0	6505
P 1	1600	161	<b>30.25%</b>	0.55	98.00%	0	6509
P 2	1600	161	<b>30.25%</b>	0.55	98.00%	0	6506
P 5	1600	161	<b>27.73%</b>	0.57	98.00%	0	6506
OL	2500	201		1.43	99.00%	0	12057
P 1	2500	201	<b>26.51%</b>	1.05	98.67%	0	12061
P 2	2500	201	<b>25.81%</b>	1.06	98.67%	0	12058
P 5	2500	201	<b>25.12%</b>	1.07	98.67%	0	12058
OL	6400	321		5.80	99.00%	0	48666
P 1	6400	321	<b>24.50%</b>	4.38	98.67%	0	48670
P 2	6400	321	<b>23.58%</b>	4.43	99.00%	0	48667
P 5	6400	321	<b>22.43%</b>	4.50	99.00%	0	48667
OL	8100	361		8.43	73.00%	23	69590
P 1	8100	361	<b>29.67%</b>	5.93	76.33%	20	69311
P 2	8100	361	<b>28.56%</b>	6.02	73.33%	21	69453
P 5	8100	361	<b>27.53%</b>	6.11	81.67%	15	69256
OL	10000	401		11.11	26.67%	6933	182773
P 1	10000	401	<b>26.79%</b>	8.13	21.67%	6824	184174
P 2	10000	401	<b>25.86%</b>	8.24	22.00%	7006	183272
P 5	10000	401	<b>24.90%</b>	8.34	20.00%	7399	184758

Table III. Flow in a Driven Cavity: Accumulation of the Compressed Jacobian

$F-$  column lists the number of cash misses that increases with the problem size. The upper limit of the available main memory is reached when the number of steps taken in each direction grows larger than 40. In this case the computer goes into paging mode requiring an increasing number of hard disc accesses ( $F+$  column). The effect on the cpu usage as well as the overall elapsed time is as expected.

The Jacobian of the flow in a driven cavity problem is sparse exhibiting a regular pattern. Seeding [Curtis et al. 1974; Newsam and Ramsdell 1983] can be used to compute a compressed Jacobian by propagation of  $l \ll n$  directional derivatives in forward mode. The minimization of  $l$  can be regarded as a vertex coloring problem in the column incidence graph, which is known to be np-hard [Garey and Johnson 1979]. A possible seed matrix consists of  $4\sqrt{n} + 1$  column vectors with entries that are equal to 0 or 1. This seed matrix represents a considerable improvement over the  $n \times n$  identity that is propagated forward otherwise. Consequently, we are able to compute much larger problems before reaching the memory limits as illustrated in Table III. The savings due to preaccumulation are decreasing for growing values of  $n$  due to the reduced relative number of directional derivatives to be propagated.

## 6. CONCLUSION

We presented a version of the differentiation-enabled NAGWare Fortran 95 compiler that provides new language extensions and source transformation techniques to facilitate the computation of first derivatives. The local preaccumulation of gradients of scalar assignments resulted in an average speedup of about 30% for two practically relevant test problems.

A major goal of this development is the construction of an intuitive and hierarchical interface that allows the user to exploit additional information such as sparsity of the Jacobian. The convenience of how derivative information can be obtained through the compiler is expected to contribute to its wide acceptance as an essential tool for scientific computing. Further effort has to be put on improving the efficiency of the derivative code. Ongoing work aims to extend the capabilities of the compiler toward the computation of adjoints in reverse mode automatic differentiation [Naumann and Riehme 2004].

## 7. ACKNOWLEDGMENT

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 and by NSF under ITR contract OCE-0205590.

This work would have been impossible without the Numerical Algorithms Group’s support while setting up the interface between the compiler and the automatic differentiation algorithms. Malcolm Cohen in particular made considerable contributions to the underlying software infrastructure.

The flexibility of the University of Hertfordshire in Hatfield, UK, regarding the management of the project has been another important factor.

## REFERENCES

- AHO, A., SETHI, R., AND ULLMAN, J. 1986. *Compilers. Principles, Techniques, and Tools*. Addison-Wesley, Reading.
- AVERIK, B., CARTER, R., AND MORÉ, J. 1991. The MINPACK-2 test problem collection (preliminary version). Technical Memorandum ANL/MCS-TM-150, Mathematics 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, Mathematics and Computer Science Division, Argonne National Laboratory. Aug. See <http://www.mcs.anl.gov/petsc>.
- BENSON, S., MCINNIS, L., AND MORÉ, J. 2000. TAO users manual. Tech. Rep. ANL/MCS-TM-242, Mathematics and Computer Science Division, Argonne National Laboratory. See [www.mcs.anl.gov/tao](http://www.mcs.anl.gov/tao).
- 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.
- ACM Transactions on Mathematical Software, Vol. V, No. N, September 2004.

- 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. AND HAGHIGHAT, M. 1996. Hierarchical approaches to automatic differentiation. In *[Berz et al. 1996]*. SIAM, 82–94.
- 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.
- BISCHOF, C., ROH, L., AND MAUER, A. 1997. ADIC — An extensible automatic differentiation tool for ANSI-C. *Software: Practice and Experience* 27, 12, 1427–1456.
- COHEN, M., NAUMANN, U., AND RIEHME, J. 2003. Toward differentiation-enabled Fortran 95 compiler technology. In *Proceedings of the 2003 ACM Symposium on Applied Computing*. 143–147.
- 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.* 13, 117–119.
- GAREY, M. AND JOHNSON, D. 1979. *Computers and Intractability - A Guide to the Theory of NP-completeness*. W. H. Freeman and Company, San Francisco.
- 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.
- HASCOËT, L., NAUMANN, U., AND PASCUAL, V. 2004. TBR analysis in reverse-mode Automatic Differentiation. In *Future Generation Computer Systems – Special Issue on Automatic Differentiation*, M. Bücker, Ed. Elsevier. To appear.
- HOPKINS, T. AND PHILLIPS, C. 1988. *Numerical methods in practice using the NAG library*. International computer science series. Addison-Wesley, Reading.
- MORÉ, J. 2001. Automatic differentiation tools in optimization software. In *[Corliss et al. 2002]*. 25–34.
- NAUMANN, U. 2002. 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. 2004. Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph. *Math. Prog.* 99, 3 (April), 399–421.
- NAUMANN, U. AND RIEHME, J. 2004. Computing adjoints with the NAGWare Fortran 95 compiler. Presentation at Fourth International Conference on Automatic Differentiation, Chicago, July, 2004. Article is under review for post-conference book.
- NEWSAM, G. AND RAMSDELL, J. 1983. Estimation of sparse Jacobian matrices. *SIAM J. Alg. Dis. Meth.* 4, 404–417.
- ROE, P. 1981. Approximating Riemann solvers, parameter vectors, and difference schemes. *J. Comp. Physics* 43, 357–372.
- TADJOUDDINE, M., FORTH, S., AND PRYCE, J. 2001. AD tools and prospects for optimal AD in CFD flux calculations. In *[Corliss et al. 2002]*. 255–261.
- TADJOUDDINE, M., FORTH, S., PRYCE, J., AND REID, J. 2002. Performance issues for vertex elimination methods in computing Jacobians using automatic differentiation. In *Computational Science – ICCS 2002, Proceedings of the International Conference on Computational Science, Amsterdam, The Netherlands, April 21–24, 2002. Part II*, P. Sloot, C. Tan, J. Dongarra, and A. Hoekstra, Eds. Lecture Notes in Computer Science, vol. 2330. Springer, Berlin, 1077–1086.
- THE NUMERICAL ALGORITHMS GROUP, L. 2000. NAG Fortran 90 library. online documentation, Oxford, UK. <http://www.nag.co.uk/numeric/FN/manual/html/FNlibrarymanual.asp>.

- THE NUMERICAL ALGORITHMS GROUP, L. 2002. The NAG Fortran library manual, mark 20. online documentation. <http://www.nag.co.uk/numeric/f1/manual/html/FLlibrarymanual.asp>.
- UTKE, J. 2004. Flattening of basic blocks. Presentation at Fourth International Conference on Automatic Differentiation, Chicago, July, 2004. Article is under review for post-conference book.
- UTKE, J. AND NAUMANN, U. 2004a. Optimality-preserving elimination of linearities in jacobian accumulation. Presentation at SIAM Workshop on Combinatorial Scientific Computing, San Francisco, February, 2004. Article is under review for special issue on combinatorial scientific computing of the Electronic Transactions on Numerical Analysis (ETNA), dedicated to Professor Alan George.
- UTKE, J. AND NAUMANN, U. 2004b. Separating language-dependent und independent tasks for the semantic transformation of numerical programs. In *Software Engineering and Applications*, M. Hamza, Ed. ACTA Press. To appear.

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