ACM Transactions on Mathematical Software (TOMS).
Volume 30, Issue 3, September 2004, pp266-299

Jacobian Code Generated by Source Transformation and Vertex Elimination Can Be as Efficient as Hand-Coding

SHAUN A. FORTH, MOHAMED TADJOUDDINE, and JOHN D. PRYCE
Cranfield University (Shrivenham Campus)
and
JOHN K. REID
JKR Associates

This article presents the first extended set of results from ELIAD, a source-transformation implementation of the vertex-elimination Automatic Differentiation approach to calculating the Jacobians of functions defined by Fortran code (Griewank and Reese, Automatic Differentiation of Algorithms: Theory, Implementation, and Application, 1991, pp. 126–135). We introduce the necessary theory in terms of well known algorithms of numerical linear algebra applied to the linear, extended Jacobian system that prescribes the relationship between the derivatives of all variables in the function code. Using an example, we highlight the potential for numerical instability in vertex-elimination. We describe the source transformation implementation of our tool ELIAD and present results from five test cases, four of which are taken from the MINPACK-2 collection (Averick et al, Report ANL/MCS-TM-150, 1992) and for which hand-coded Jacobian codes are available. On five computer/compiler platforms, we show that the Jacobian code obtained by ELIAD is as efficient as hand-coded Jacobian code. It is also between 2 to 20 times more efficient than that produced by current, state of the art, Automatic Differentiation tools even when such tools make use of sophisticated techniques such as sparse Jacobian compression. We demonstrate the effectiveness of reverse-ordered pre-elimination from the (successively updated) extended Jacobian system of all intermediate variables used once. Thereafter, the monotonic forward/reverse ordered eliminations of all other intermediates is shown to be very efficient. On only one test case were orderings determined by the Markowitz or related VLR heuristics found superior. A re-ordering of the statements of the Jacobian code, with the aim of reducing reads and writes of data from cache to registers, was found to have mixed effects but could be very beneficial.

Categories and Subject Descriptors: G.1.0 [**Numerical Analysis**]: General—*Stability (and instability)*; G.1.4 [**Numerical Analysis**]: Quadrature and Numerical Differentiation—*Automatic*

This work was funded by the UK's EPSRC and MOD under grant GR/R21882.
Authors' addresses: S. A. Forth, M. Tadjouddine (Engineering Systems Department), and J. D. Pryce (Communications and Information Sytems Engineering), Cranfield University (Shrivenham Campus), Shrivenham, Swindon SN6 8LA, UK; email: {S.A.Forth; M.Tadjouddine; J.D.Pryce}@cranfield.ac.uk; J. K. Reid, JKR Associates and Rutherford Appleton Laboratory, Atlas Centre, Rutherford Appleton Laboratory, Didcot, Oxon OX11 OQX, UK; email: J.K.Reid@rl.ac.uk.

differentiation; G.1.5 [**Numerical Analysis**]: Roots of Nonlinear Equations—*Systems of equations*; G.1.6 [**Numerical Analysis**]: Optimization—*Least squares methods*; G.4 [**Mathematical Software**]: *Efficiency*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Jacobian, source transformation, vertex elimination

GLOSSARY

$\mathbf{A}$	Adjoint matrix	p	Number of intermediate variables
$c_{i,j}$	Local derivative $\partial\Phi_i/\partial u_j$	$\mathbf{P}$	Matrix associated with extended system
$\mathbf{C}$	Matrix of local derivatives	q	Number of columns in $\mathbf{S}$
$\mathbf{D}$	Solution of extended Jacobian system	$\mathbf{Q}$	Matrix associated with extended system
$\mathbf{e}_i$	i-th column of unit matrix	$\mathbf{S}$	Seed matrix of order $n \times q$ or $m \times q$
$\mathbf{f}$	Given function	$\mathbf{s}_i$	i-th column of $\mathbf{S}$
$\nabla\mathbf{f}(\mathbf{x})$	Jacobian matrix, of order $m \times n$	u_i	i-th code variable
m	Number of dependent variables	$\mathbf{u}$	All active variables
n	Number of independent variables	$\mathbf{v}$	Variables of the code-list
N	$= n + p + m$	w_i	i-th intermediate variable
$N_{\lvert\mathbf{c}\rvert=1}$	Number of unit entries $c_{i,j} = \pm 1$ in $\mathbf{C}$	$\mathbf{w}$	Intermediate variables
$N_{\lvert\mathbf{c}\rvert\neq 1}$	Number of non-unit entries in $\mathbf{C}$	W	Computational work
N_{evals}	Number of separate evaluations timed	$\mathbf{x}$	Independent variables
N_{repet}	Number of repeats of timing test	$\mathbf{y}$	Dependent variables

1. INTRODUCTION

Automatic Differentiation (AD) concerns the process of taking a function $\mathbf{y} = \mathbf{f}(\mathbf{x})$, with $\mathbf{f}$ defined by a computer code, that maps *independent variables* $\mathbf{x} \in I\!R^n$ to *dependent variables* $\mathbf{y} \in I\!R^m$ and then constructing new code that will also calculate derivatives of $\mathbf{f}$. AD relies on the fact that each statement of the code involving floating-point numbers may be individually differentiated. The *forward mode* of AD creates new code that, for each of the code's variables, calculates the numerical values of the variable and its derivatives with respect to the independent variables. *Reverse* (or *adjoint) mode* AD produces code that passes forward through the original code storing information required for a reverse pass in which the sensitivities of the nominated dependent variables to changes in the values of the code's variables are calculated. A thorough introduction to the field may be found in Griewank [2000] and further theoretical results and applications may be found in the collections [Griewank and Corliss 1991; Berz et al. 1996; Corliss et al. 2001].

AD software tools exist for codes written in Fortran [Bischof et al. 1996a; Giering and Kaminski 1998; Faure and Papegay 1998; Pryce and Reid 1998], C and C++ [Bischof et al. 1997; Bendtsen and Stauning 1996; Griewank et al. 1996] and Matlab [Verma 1998; Forth 2001; Bischof et al. 2002] amongst other programming languages. These tools implement AD in one of two ways, source transformation or operator overloading.

Source transformation Griewank [2000, Section 5.7–5.8] involves use of sophisticated compiler techniques. For the forward mode, new code is produced that, when executed, calculates derivatives as well as values for the dependent variables. In reverse mode, new code is produced that will calculate so-called

adjoint values **backwards** through an enhanced version of the original code in such a way as to calculate $\mathbf{S}^T \nabla \mathbf{f}(\mathbf{x})$ for any matrix $\mathbf{S}$ with m rows.

Alternatively, the *operator overloading* approach [Griewank 2000, Section 5.1–5.6] utilises a feature available in some modern, object-oriented computer languages. New types of variable are defined that store the variable's value and, for forward mode, also store its derivatives. For reverse mode, instead of derivatives, sufficient information is saved (to a so-called *tape*) to enable the required reverse propagation of sensitivities. In both cases, arithmetic operations and intrinsic functions are extended to the new types.

For languages such as Fortran or C (for which optimising compilers exist) and applied to code featuring scalar assignments, it is generally found that the source transformation approach produces more efficient derivative code (see, e.g., Tadjouddine et al. [2001] and Pryce and Reid [1998]).

Our work is motivated by the particular requirement for Jacobian code in solving systems of nonlinear equations via Newton's, or a related method. Such systems arise in applications such as computational fluid dynamics, computational chemistry, and data-fitting. For examples, see the test cases of Averick et al. [1992] and references therein. In such cases, the system Jacobian may be needed for direct solution of a Newton update. Alternatively, when using Krylov-based solvers for which Jacobian-vector products may be evaluated by finite differencing or conventional AD tools, the Jacobian frequently needs to be determined for preconditioning, for example, when using incomplete LU factorisation [Hovland and McInnes 2001].

When discussing sparsity, we will use the term *entry* for a matrix coefficient that we represent explicitly because we cannot be sure that it is zero. An entry may 'accidentally' have the value zero, so the term 'nonzero' is not suitable.

The structure of the rest of this article is as follows: In Section 2 and Section 3, we review the matrix interpretation of the conventional forward and reverse modes of AD and then the vertex elimination approach of Griewank and Reese [1991]. This is necessary to understand the implementation of our vertex elimination tool ELIAD. ELIAD is implemented via source transformation as described in Section 4. Our test environment is explained in Section 5. In Section 6, we present and discuss an extended set of results from ELIAD. For these test cases, we demonstrate that AD via vertex elimination and source transformation enables the calculation of Jacobians as fast as hand-coded Jacobian code and with more than twice the efficiency of present AD tools and techniques. Section 7 presents conclusions and the outlook for extending ELIAD's coverage of Fortran and improving it to produce even faster Jacobian code.

2. MATRIX INTERPRETATION OF AUTOMATIC DIFFERENTIATION

The matrix interpretation of AD [Griewank 2000, Sect. 8.1] allows us to view the standard forward and reverse modes of AD in terms of the well-known forward and back substitution algorithms for systems of linear equations with triangular coefficient matrices. In Section 2.1, we introduce the lower triangular extended Jacobian system and, in Section 2.2, we indicate how this system is solved in forward and reverse mode AD. We then consider the exploitation of

sparsity, the number of floating-point operations involved, and how to treat the temporary variables that occur within statements.

2.1 The Extended Jacobian System

For a function $\mathbf{f} : \mathbf{x} \in I\!R^n \mapsto \mathbf{y} \in I\!R^m$ that is defined by computer code and which we wish to differentiate, we define three (sub)groups of the code's variables:

independent variables: $\mathbf{x} = (x_i, i = 1, \ldots, n)$ whose values must be supplied and with respect to which the derivatives $\partial\mathbf{y}/\partial\mathbf{x}$ are required,

dependent variables: $\mathbf{y} = (y_i, i = 1, \ldots, m)$ which must be calculated and whose derivatives are required,

intermediate variables: $\mathbf{w} = (w_i, i = 1, \ldots, p)$ whose values are calculated (perhaps indirectly) from the $\mathbf{x}$ and which are needed to calculate (perhaps indirectly) the $\mathbf{y}$.

Collectively, these variables are termed *active* variables. We define *inactive* variables as those which are not active.

Example 2.1 (*Example Code*). For the code fragment

$$\begin{aligned} w_1 &= \log(x_1 * x_2) \\ w_2 &= x_2 * x_3^2 - a \\ w_3 &= b * w_1 + x_2/x_3 \\ y_1 &= w_1^2 + w_2 - x_2 \\ y_2 &= \sqrt{w_3} - w_2 \end{aligned}$$

we wish to calculate $\partial(y_1, y_2)/\partial(x_1, x_2, x_3)$. Hence, x_1, x_2, x_3 are the $n = 3$ independent variables, y_1, y_2 are the $m = 2$ dependent variables, and w_1, w_2, w_3 are the $p = 3$ intermediate variables. Values of the variables a, b must be supplied but, since we do not require derivatives with respect to them, they are inactive.

For the purposes of analysis, we regard an execution of the function code as defining $N = n + p + m$ internal variables $u_i,\ i = 1, \ldots, N$ in the following manner. First there are n copies of the independent variables to the internal variables,

$$u_i = x_i, \quad i = 1, \ldots n, \tag{1}$$

followed by the $p + m$ statements of the code

$$u_i = \Phi_i(\{u_j\}_{j \prec i}), \quad i = n + 1, \ldots, N, \tag{2}$$

where the precedence relation $j \prec i$ means that the variable u_j is involved in the expression Φ_i. Each Φ_i represents a composition of one or more *elemental / intrinsic functions* or *elemental / intrinsic operators* of the programming language. For the most part, we will assume that no expression for a dependent variable involves another dependent variable, that is, if $i > n + p$ and $j \prec i$, then $j \leq n + p$. This may be ensured by making a copy of any dependent variable used to calculate another dependent variable, but we will also explain how to avoid the need for this.

Example 2.2 (*Example Code*). The code fragment of Example 2.1 may be rewritten in the form of (1) to (2) as

$$\left.\begin{array}{lll} u_1 = x_1 & & \\ u_2 = x_2 & & \\ u_3 = x_3 & & \\ u_4 = \Phi_4(u_1, u_2) & = \log(u_1 * u_2) & \\ u_5 = \Phi_5(u_2, u_3) & = u_2 * u_3^2 - a & \\ u_6 = \Phi_6(u_2, u_3, u_4) & = b * u_4 + u_2/u_3 & \\ u_7 = \Phi_7(u_2, u_4, u_5) & = u_4^2 + u_5 - u_2 & \\ u_8 = \Phi_8(u_5, u_6) & = \sqrt{u_6} - u_5 & \end{array}\right\}.$$

The assignments of (1) and (2) can be written as the following system of nonlinear equations

$$\left.\begin{array}{ll} 0 = x_i - u_i, & i = 1, \ldots, n \\ 0 = \Phi_i(\{u_j\}_{j \prec i}) - u_i, & i = n+1, \ldots, N \end{array}\right\}. \tag{3}$$

We assume that the functions Φ_i have continuous first derivatives, define the gradient operator ∇ by $\nabla = (\partial/\partial x_1, \ldots, \partial/\partial x_n)$, and differentiate (3) with respect to the independent variables $x_1, \ldots, x_n$, to give

$$\left.\begin{array}{ll} -\nabla u_i = -\mathbf{e}_i, & i = 1, \ldots, n \\ \sum_{j \prec i} c_{i,j} \nabla u_j - \nabla u_i = 0, & i = n+1, \ldots, N \end{array}\right\}, \tag{4}$$

where $\mathbf{e}_i$ is the n-vector with unit entry in position i and $c_{i,j}$ are the *local derivatives* $c_{i,j} = \partial \Phi_i / \partial u_j$. On defining the matrix $\mathbf{C} = \{c_{i,j}\}_{1 \le i,j \le N}$ to be composed of all such entries and zeros elsewhere, the linear system (4) can be compactly rewritten as the *extended Jacobian system*

$$(\mathbf{C} - \mathbf{I}_N)\mathbf{D} = -\mathbf{P}, \tag{5}$$

with $\mathbf{D} = \nabla \mathbf{u}$ and

$$\mathbf{P} = \begin{bmatrix} \mathbf{I}_n \\ \mathbf{0}_{(m+p)\times n} \end{bmatrix}. \tag{6}$$

The matrix $\mathbf{C} - \mathbf{I}_N$ is called the *extended Jacobian* and is necessarily lower triangular because each value u_i is calculated from previously calculated values u_j with $j \prec i$. We define $N_{|\mathbf{c}|=1}$ to be the number of entries in $\mathbf{C}$ taking the value ± 1 and $N_{|\mathbf{c}|\neq 1}$ to be the number of other entries.

Example 2.3 (*Extended Jacobian System*). For our example code, the extended Jacobian system is given by

$$\left[\begin{array}{ccc|ccc|c} -1 & & & & & & \\ & -1 & & & & & \\ & & -1 & & & & \\ \hline c_{4,1} & c_{4,2} & & -1 & & & \\ & c_{5,2} & c_{5,3} & & -1 & & \\ & c_{6,2} & c_{6,3} & c_{6,4} & & -1 & \\ \hline & c_{7,2} & & c_{7,4} & c_{7,5} & & -1 \\ & & & & c_{8,5} & c_{8,6} & \end{array}\right] \left[\begin{array}{c} \nabla u_1 \\ \nabla u_2 \\ \nabla u_3 \\ \nabla u_4 \\ \nabla u_5 \\ \nabla u_6 \\ \nabla u_7 \\ \nabla u_8 \end{array}\right] = \left[\begin{array}{ccc} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{array}\right],$$

with entries in the extended Jacobian given by

$$\left.\begin{array}{ll} c_{4,1} = 1/u_1, & c_{6,4} = b \\ c_{4,2} = 1/u_2, & c_{7,2} = -1, \\ c_{5,2} = u_3^2, & c_{7,4} = 2 * u_4, \\ c_{5,3} = 2 * u_2 * u_3, & c_{7,5} = 1, \\ c_{6,2} = 1/u_3, & c_{8,5} = -1 \\ c_{6,3} = -u_2/u_3^2, & c_{8,6} = 1/(2\sqrt{u_6}) \end{array}\right\}. \tag{7}$$

We see that $N_{|\mathbf{c}|=1} = 3$ and $N_{|\mathbf{c}|\neq 1} = 9$.

2.2 Forward and Reverse Mode AD

In the forward method, we solve the lower triangular system (5) for the n columns of $\mathbf{D}$ by forward substitution, then extract the last m rows to obtain the Jacobian. By defining the matrix $\mathbf{Q}$ to be

$$\mathbf{Q} = \begin{bmatrix} \mathbf{0}_{(n+p)\times m} \\ \mathbf{I}_m \end{bmatrix}, \tag{8}$$

we can express the solution of Eq. (5) and the extraction of the final m rows of $\mathbf{D}$ as

$$\nabla\mathbf{f} = \mathbf{Q}^T(\mathbf{C} - \mathbf{I}_N)^{-1}(-\mathbf{P}). \tag{9}$$

If we group the terms as $\nabla\mathbf{f} = \mathbf{Q}^T[(\mathbf{C}-\mathbf{I}_N)^{-1}(-\mathbf{P})]$, we have a formal description of forward mode AD. Alternatively, by grouping Eq. (9) as $\nabla\mathbf{f} = [(-\mathbf{Q}^T)(\mathbf{C} - \mathbf{I}_N)^{-1}]\mathbf{P}$, and defining *adjoint* quantities $\mathbf{A}$ as the solution, by back-substitution, of the upper-triangular system,

$$(\mathbf{C} - \mathbf{I}_N)^T\mathbf{A} = -\mathbf{Q}, \tag{10}$$

we obtain the Jacobian as the first n columns of $\mathbf{A}^T$, or more formally $\nabla\mathbf{f} = \mathbf{A}^T\mathbf{P}$. This is a matrix interpretation of reverse mode AD [Griewank 2000, pp. 161–162].

It should be noted that present AD tools, such as TAMC, are designed to calculate an arbitrary Jacobian-matrix product (forward mode) or an arbitrary matrix-Jacobian product (reverse mode). Consequently, they allow for $\mathbf{P}$ to be of the form $\mathbf{P} = \begin{bmatrix} \mathbf{S} \\ \mathbf{0}_{(m+p)\times q} \end{bmatrix}$ for an arbitrary full matrix $\mathbf{S}$ with n rows and q columns and $\mathbf{Q}$ to be of the form $\mathbf{Q} = \begin{bmatrix} \mathbf{0}_{(n+p)\times q} \\ \mathbf{S} \end{bmatrix}$ for an arbitrary full matrix $\mathbf{S}$ with m rows and q columns. Such a matrix $\mathbf{S}$ is termed a *seed matrix*. We denote its i-th column by $\mathbf{s}_i$. Of course, the solutions $\mathbf{D}$ of (5) or $\mathbf{A}$ of (10) are then no longer simple derivatives or adjoints, but are linear combinations $\nabla\mathbf{f}\,\mathbf{s}_i$ of derivatives or $\mathbf{s}_j^T\nabla\mathbf{f}$ of adjoints. It may readily be verified that if $\mathbf{S}$ is full, the solution of (5) or (10) is full (apart from 'accidental' cancellations where two values sum to zero) since each of the last $p+m$ rows and each of the first $n+p$ columns of $\mathbf{C}$ has at least one entry. To calculate the Jacobian $\nabla\mathbf{f}$ using such tools, the seed matrix $\mathbf{S}$ must be set to $\mathbf{I}_n$ (forward mode) or $\mathbf{I}_m$ (reverse mode).

Let us consider these two approaches for our Example 2.3.

Example 2.4 (*Forward mode AD*). Forward mode AD solves the linear system of Example 2.3 using $9 \times 3 = 27$ multiplications and $7 \times 3 = 21$ additions/subtractions, that is, 48 floating-point operations (flops).

Example 2.5 (*Reverse/Adjoint Mode AD*). For Example 2.3 the adjoint system is solved using $9 \times 2 = 18$ multiplications and $5 \times 2 = 10$ additions/subtractions, that is, 28 flops.

For our simple example, reverse mode AD uses fewer arithmetic operations than forward mode. In both examples, we have neglected multiplications by unit entries $c_{i,j} = \pm 1$. We have, however, counted the multiplications and additions by zero that occur when $\mathbf{S}$ is not full. This is because the basic operation is the addition of a multiple of ∇u_i to ∇u_j and testing for zeros in ∇u_i would slow the code.

2.3 Taking Account of Sparsity

If no account is taken of sparsity in $\mathbf{S}$ and if the given function $\mathbf{f}$ takes time $W(\mathbf{f})$, forward mode AD calculates the Jacobian in time $O(n) \times W(\mathbf{f})$ and reverse mode does so in time $O(m) \times W(\mathbf{f})$ [Griewank 2000]. For problems with many independent variables ($n \gg 1$) and few dependents, we see that reverse mode is to be preferred (e.g., large scale optimization).

For Jacobians with known sparsity pattern, there are established techniques for reducing the size of the seed matrix $\mathbf{S}$ needed to calculate all nonzero entries of $\nabla \mathbf{f}(\mathbf{x})$ [Griewank 2000, Chap. 7]. Such techniques, collectively termed *Jacobian compression*, frequently reduce the size, but not to less than q columns in forward mode or q rows in reverse with q the maximum number of entries in any row (forward mode) or column (reverse mode) of the Jacobian.

Another possibility is to employ a data structure that permits all operations with values that are known to be zero to be avoided completely. This is the approach taken by our code ELIAD. Explicit code is generated for each of the elimination operations

$$c_{i,j} = c_{i,j} - c_{i,k}c_{k,j}$$

for which $c_{i,k}$ and $c_{k,j}$ are both entries. No code is generated where is it known a priori that either $c_{i,k}$ or $c_{k,j}$ is always zero.

The techniques discussed so far in this section assume that the sparsity structure is fixed, so that the set of vectors or generated code needs to be determined once, and they are thus static exploitations of the Jacobian sparsity. An alternative is via the dynamic exploitation of sparsity. Here, the data are stored in some sparse format that is adjusted dynamically (at run time). Prime examples of such an approach are the use of the SparsLinC library in ADIFOR [Bischof et al. 1996a, 1996b], use of sparse options in AD01 [Pryce and Reid 1998], and exploitation of the sparse matrix class of Matlab [Coleman and Verma 1998; Forth 2001]. The formal operations count for such an approach is low, but the overhead of manipulating the sparse storage typically makes them uncompetitive unless the sparsity structure is not fixed [Griewank 2000, p. 156]. The approach is also of use in determining a fixed sparsity structure prior to Jacobian compression.

2.4 Computational Cost of Forward and Reverse Mode AD

For a given function $\mathbf{f}$, we assume the computational cost $W(\nabla\mathbf{f})$ of evaluating its derivatives via AD may be written as

$$W(\nabla\mathbf{f}) = W(\mathbf{f}) + W(\mathbf{C}) + W(\text{linear solve}), \tag{11}$$

where:

—$W(\mathbf{f})$ is the cost of evaluating the original function.
—$W(\mathbf{C})$ is the cost of evaluating $\mathbf{C}$, that is, all the local derivatives $c_{i,j}$.
—$W(\text{linear solve})$ is the cost of solving the linear system (5) or (10).

We will measure the cost in floating-point operations (flops), concentrating on $W(\text{linear solve})$ since we can have little influence on the other two costs.

Griewank [2000, Chap. 3] takes account of the time required for data transfers from and to the floating-point registers under rather conservative conditions. However, as we shall see in Sections 3.1 and 4, our optimized derivative code may access extended Jacobian entries in a nonsequential way and this, together with the effects of compiler optimizations make this time difficult to quantify and so, regretfully, we neglect it in this subsection. This approximation is in line with that of previous work [Griewank and Reese 1991; Griewank 2000, Chap. 8].

For forward mode AD and without exploiting Jacobian compression or dynamic sparsity (see Section 2.3), the cost of the linear solve of (5) is given by

$$W(\text{linear solve (forward mode)}) = n(2N_{|\mathbf{c}|\neq 1} + N_{|\mathbf{c}|=1} - p - m). \tag{12}$$

We incur n multiplications when multiplying a row by a $c_{i,j} \neq \pm 1$ and n additions for all entries as we accumulate the vectors to the u_i. We subtract the cost of $n(m+p)$ additions since the first gradient in each line of the forward substitution is assigned and not added to the intermediate's or dependent's derivatives.

For a sparse Jacobian, (forward) Jacobian compression techniques frequently allow the Jacobian $\nabla\mathbf{f}(\mathbf{x})$ to be recovered from $q < n$ Jacobian-vector products $\nabla\mathbf{f}(\mathbf{x})\mathbf{s}_i$, $i = 1, \ldots, q$. Clearly, the cost of calculating this is given by (12), but with n replaced by q.

Similarly, for reverse mode AD, the computational cost of the linear solve associated with (10) is

$$W(\text{linear solve (reverse mode)}) = m(2N_{|\mathbf{c}|\neq 1} + N_{|\mathbf{c}|=1} - p - n). \tag{13}$$

If Jacobian compression is possible, we propagate q vector-Jacobian products $\mathbf{s}_j^T\nabla\mathbf{f}(\mathbf{x})$, $j = 1, \ldots, q$, and extract the Jacobian with a resulting cost for the linear solve given by (13) with m replaced by q.

2.5 Statement-Level Versus Code-List Differentiation

So far in this article, we have differentiated each statement locally with respect to the active variables that appear in its right-hand side. This is termed a *statement-level differentiation*. An alternative, used later in this article, is based on the code list [Griewank 2000], in which the original program is

rewritten so that the right-hand side of each statement has a single unary or binary operation.

Example 2.6 (*Code-List*). A possible code list for Example 2.1 is:

$$
\begin{array}{lll}
v_1 \equiv x_1 & v_7 = v_6 * v_2 & v_{13} = v_8 - v_2 \\
v_2 \equiv x_2 & v_8 = v_7 - a & v_{14} = v_5^2 \\
v_3 \equiv x_3 & v_9 = 1/v_3 & v_{15} = \sqrt{v_{12}} \\
v_4 = v_1 * v_2 & v_{10} = v_2 * v_9 & v_{16} = v_{14} + v_{13} \\
v_5 = \log(v_4) & v_{11} = b * v_5 & v_{17} = v_{15} - v_8 \\
v_6 = v_3^2 & v_{12} = v_{11} + v_{10} &
\end{array}
$$

in which the division has been replaced by the nonlinear reciprocal operation followed by a multiplication, and statements are ordered such that the final two variables v_{16}, v_{17} correspond to the dependent variables y_1, y_2.

3. AUTOMATIC DIFFERENTIATION BY VERTEX ELIMINATION

If there are no intermediate variables ($p = 0$), the second line of Eq. (4) shows that the matrix $\mathbf{C}$ is the Jacobian $\nabla\mathbf{f}(\mathbf{x})$. Griewank and Reese [1991] therefore systematically reduced the extended Jacobian system (5) by Gaussian elimination of the intermediate variables to obtain the Jacobian. Their original analysis actually used the computational graph but was later reinterpreted [Griewank 2000] using the extended Jacobian. We defer the graph interpretation until Section 3.2, first introducing vertex elimination via the extended Jacobian description in Section 3.1 since we believe it is more accessible to the scientific computing community. We discuss the connection to standard AD algorithms in Section 3.3. In Section 3.4, we discuss techniques to determine the order in which we choose to eliminate intermediates from the extended Jacobian. In Section 3.5, we show that a poor choice of ordering may lead to accumulation of roundoff and instability. Section 3.6 describes techniques for statement-level differentiation of the function, which motivate the pre-elimination strategy of Section 3.7.

3.1 Reduction of the Extended Jacobian

To reduce the extended Jacobian, we apply Gaussian elimination, pivoting on the diagonal entries of the columns corresponding to intermediate variables. In each case, we add multiples of the pivot row to later rows to create zeros below the diagonal in the pivot column. Since the pivot row and pivot column are no longer relevant, we then discard them—this is what makes it Gaussian, rather than the less efficient Jordan, elimination. Note that the form of the extended Jacobian is preserved in the sense that it remains lower triangular with diagonal entries equal to -1.

We illustrate this with the extended Jacobian system of Example 2.3. We start with the system

$$\left[\begin{array}{ccc|ccc|c} -1 & & & & & & \\ & -1 & & & & & \\ & & -1 & & & & \\ \hline c_{4,1} & c_{4,2} & & -1 & & & \\ & c_{5,2} & c_{5,3} & & -1 & & \\ & c_{6,2} & c_{6,3} & c_{6,4} & & -1 & \\ \hline & -1 & & c_{7,4} & 1 & & -1 \\ & & & & -1 & c_{8,6} & & -1 \end{array}\right] \left[\begin{array}{c} \nabla u_1 \\ \nabla u_2 \\ \nabla u_3 \\ \hline \nabla u_4 \\ \nabla u_5 \\ \nabla u_6 \\ \hline \nabla u_7 \\ \nabla u_8 \end{array}\right] = \left[\begin{array}{ccc} -1 & & \\ & -1 & \\ & & -1 \\ \hline & & \\ & & \\ & & \\ \hline & & \\ & & \end{array}\right]. \tag{14}$$

First, pivoting on diagonal 4, we add multiples $c_{6,4}$ and $c_{7,4}$ of row 4 to rows 6 and 7. This creates two new entries, or fill-ins, $c_{6,1} = c_{6,4} * c_{4,1}$, $c_{7,1} = c_{7,4} * c_{4,1}$, and modifies two entries, $c_{6,2} = c_{6,2} + c_{6,4} * c_{4,2}$, $c_{7,2} = -1 + c_{7,4} * c_{4,2}$. Row and column 4 are now discarded to give

$$\left[\begin{array}{ccc|ccc|c} -1 & & & & & & \\ & -1 & & & & & \\ & & -1 & & & & \\ \hline & c_{5,2} & c_{5,3} & -1 & & & \\ c_{6,1} & c_{6,2} & c_{6,3} & & -1 & & \\ \hline c_{7,1} & c_{7,2} & & 1 & & -1 & \\ & & & -1 & c_{8,6} & & -1 \end{array}\right] \left[\begin{array}{c} \nabla u_1 \\ \nabla u_2 \\ \nabla u_3 \\ \hline \nabla u_5 \\ \nabla u_6 \\ \hline \nabla u_7 \\ \nabla u_8 \end{array}\right] = \left[\begin{array}{ccc} -1 & & \\ & -1 & \\ & & -1 \\ \hline & & \\ & & \\ \hline & & \\ & & \end{array}\right]. \tag{15}$$

We now pivot on the new diagonal 4. This requires one modification $c_{7,2} = c_{7,2} + c_{5,2}$ and three fill-ins $c_{7,3} = c_{5,3}$, $c_{8,2} = -c_{5,2}$, $c_{8,3} = -c_{5,3}$. The pivot row and column are now discarded.

For the final pivot step, we have two modifications $c_{8,2} = c_{8,2} + c_{8,6} * c_{6,2}$, $c_{8,3} = c_{8,3} + c_{8,6} * c_{6,3}$ and one fill-in $c_{8,1} = c_{8,6} * c_{6,1}$ to yield

$$\left[\begin{array}{ccc|c} -1 & & & \\ & -1 & & \\ & & -1 & \\ \hline c_{7,1} & c_{7,2} & c_{7,3} & -1 \\ c_{8,1} & c_{8,2} & c_{8,3} & & -1 \end{array}\right] \left[\begin{array}{c} \nabla u_1 \\ \nabla u_2 \\ \nabla u_3 \\ \hline \nabla u_7 \\ \nabla u_8 \end{array}\right] = \left[\begin{array}{ccc} -1 & & \\ & -1 & \\ & & -1 \\ \hline & & \\ & & \end{array}\right].$$

Clearly, we now have

$$\nabla \mathbf{f}(\mathbf{x}) = \begin{bmatrix} c_{7,1} & c_{7,2} & c_{7,3} \\ c_{8,1} & c_{8,2} & c_{8,3} \end{bmatrix}$$

and, neglecting the cost of sign changes, the Jacobian has been calculated with a total cost of 12 floating-point operations (flops). This is a substantial saving over the forward mode of Example 2.4 (48 flops) and reverse mode of Example 2.5 (28 flops).

It is trivial to allow for entries $c_{i,j}$ in the final m columns of $\mathbf{C}$ (when some expressions for dependent variables involve other dependent variables). We simply perform Gaussian elimination on each such column, but do not discard the pivot row.

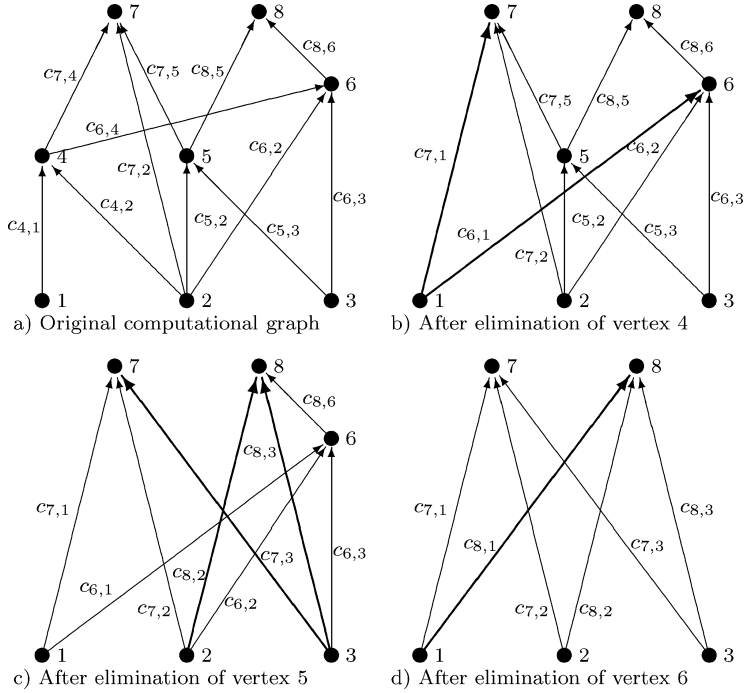

Fig. 1. Graph representation of the vertex elimination of Section 3.1.

3.2 The Computational Graph Description

The description of Section 3.1 suggests that Gaussian elimination, rather than vertex elimination, would be a more appropriate name for the algorithms of this paper. To explain the origins of the term *vertex elimination*, we return to the original graph based description of Griewank and Reese [1991].

In Figure 1(a), we show the computational graph for our example before any eliminations have been performed. The graph has vertices labelled 1 to 8 corresponding to the internal variables u_1 to u_8 of Example 2.2. Vertices 1 to 3, corresponding to the independent variables, are placed at the bottom of the graph and the dependents, labelled 7 and 8, are at the top. Vertices 4 to 6, corresponding to intermediates, lie in the center. There is a directed edge from vertex j to vertex i if $j \prec i$ and the edge is labelled with the associated local derivative $c_{i,j}$ as given in (7). This labelling is what makes it the "linearized" computational graph.

The derivative of any dependent variable i with respect to any independent j is given by the sum of the products of all edge labels for paths connecting j to i in the graph [Griewank 2000, p. 169]. For example, u_7 is connected to u_2 via three paths given by the 3 sets of edge labels $\{c_{7,4}, c_{4,2}\}$, $\{c_{7,2}\}$ and $\{c_{7,5}, c_{5,2}\}$ and so,

$$\frac{\partial u_7}{\partial u_2} = c_{7,4} * c_{4,2} + c_{7,2} + c_{7,5} * c_{5,2}.$$

Consequently, we see that a vertex labelled k and all associated edges $c_{k,j \prec k}$, $c_{i,k \prec i}$ (short for any $c_{k,j}$ where $j \prec k$, respectively, $c_{i,k}$ where $k \prec i$), can be

eliminated from the graph by using the following procedure. We take each *out-edge* labelled $c_{i,k\prec i}$ leaving vertex k and each *in-edge* $c_{k,j\prec k}$ entering and add the product $c_{i,k} * c_{k,j}$ to the edge $c_{i,j}$, creating a new edge $c_{i,j}$ if required. The edges $c_{i,k\prec i}$,$c_{k,j\prec k}$ may now be removed from the graph since their contributions to the Jacobian calculation have been added to other edges of the graph.

Example 3.1 (*Vertex Elimination in the Computational Graph*). Vertex 4 in Figure 1(a) has two in-edges $c_{4,1}$, $c_{4,2}$ and two out-edges $c_{7,4}$, $c_{6,4}$. Consequently, there are four products of out-edges with in-edges $c_{7,4} * c_{4,1}$, $c_{7,4} * c_{4,2}$, $c_{6,4} * c_{4,1}$ and $c_{6,4} * c_{4,1}$. Of these products, two are added to existing edges of the graph $c_{7,2} = c_{7,2} + c_{7,4} * c_{4,2}$, $c_{6,2} = c_{6,2} + c_{6,4} * c_{4,2}$, and two require new edges (shown in bold in Figure 1(b)) to be added, $c_{6,1} = c_{6,4} * c_{4,1}$, $c_{7,1} = c_{7,4} * c_{4,1}$. Once these operations have been performed, vertex 4 and its associated edges may be removed leaving the graph of Figure 1(b).

Comparison of the operations of Example 3.1 with the first Gaussian elimination performed on the extended Jacobian of Section 3.1 demonstrates the equivalence of the two interpretations, matrix and graph, of vertex elimination. Figures 1(c) and 1(d) show the graph after elimination of vertices 5 and 6 respectively. The numerical operations involved to update edge labels are precisely those of Section 3.1 used to update the extended Jacobian entries. Figure 1(d) shows the graph after eliminating all intermediate vertices and associated edges. We see that the graph is *bipartite*, that is, the vertices form two disjoint sets, independent and dependent, and the only edges go from an independent to a dependent. Each edge's label corresponds to a desired partial derivative.

Any entries $c_{i,j}$ in the final m columns of $\mathbf{C}$ correspond to edges between dependent variables. Their elimination (see final paragraph of Section 3.1) corresponds to elimination of edges. For our example, if there is an edge $c_{8,7}$ and it is eliminated last, all the edges to node 8 must be modified: $c_{8,i} = c_{8,i} + c_{7,i} * c_{8,7},\ i = 1, 2, 3$.

3.3 The Connection to Standard AD Algorithms

There is a close relationship between the elimination algorithm with the pivots taken in forward order, and the Forward Mode of AD, equivalent to solving the system (5) by forward substitution (Section 2.2). We may write the extended Jacobian $\mathbf{C} - \mathbf{I}$ in the equivalent block form [Griewank 2000, p. 22],

$$\mathbf{C} - \mathbf{I} = \begin{bmatrix} -\mathbf{I}_n & \mathbf{0} & \mathbf{0} \\ \mathbf{B} & \mathbf{L} - \mathbf{I}_p & \mathbf{0} \\ \mathbf{R} & \mathbf{T} & -\mathbf{I}_m \end{bmatrix}, \tag{16}$$

with $\mathbf{L}$ strictly lower-triangular. We must now solve,

$$\begin{bmatrix} -\mathbf{I}_n & \mathbf{0} & \mathbf{0} \\ \mathbf{B} & \mathbf{L} - \mathbf{I}_p & \mathbf{0} \\ \mathbf{R} & \mathbf{T} & -\mathbf{I}_m \end{bmatrix} \begin{bmatrix} \nabla \mathbf{x} \\ \nabla \mathbf{w} \\ \nabla \mathbf{y} \end{bmatrix} = \begin{bmatrix} -\mathbf{I}_n \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}. \tag{17}$$

Forward substitution, as for conventional forward mode AD, gives

$$\begin{aligned}\nabla\mathbf{x} &= \mathbf{I}_n,\\ \nabla\mathbf{w} &= -(\mathbf{L}-\mathbf{I}_p)^{-1}\mathbf{B}\nabla\mathbf{x} = -(\mathbf{L}-\mathbf{I}_p)^{-1}\mathbf{B},\\ \nabla\mathbf{y} &= \mathbf{R}\nabla\mathbf{x}+\mathbf{T}\nabla\mathbf{w} = \mathbf{R}-\mathbf{T}(\mathbf{L}-\mathbf{I}_p)^{-1}\mathbf{B}.\end{aligned}$$

In the elimination approach, we eliminate subdiagonal entries in the **L** block and all entries in the **T** block from (17),

$$\begin{bmatrix} -\mathbf{I}_n & \mathbf{0} & \mathbf{0}\\ -(\mathbf{L}-\mathbf{I}_p)^{-1}\mathbf{B} & -\mathbf{I}_p & \mathbf{0}\\ \mathbf{R}-\mathbf{T}(\mathbf{L}-\mathbf{I}_p)^{-1}\mathbf{B} & \mathbf{0} & -\mathbf{I}_m \end{bmatrix}\begin{bmatrix}\nabla\mathbf{x}\\ \nabla\mathbf{w}\\ \nabla\mathbf{y}\end{bmatrix} = \begin{bmatrix}-\mathbf{I}_n\\ \mathbf{0}\\ \mathbf{0}\end{bmatrix}, \tag{18}$$

to leave the Jacobian in the lower-left block. The two approaches are thus algebraically equivalent [Griewank 2000, Section 8.1]. We also see that in the forward-substitution approach fill-in is confined to the n columns of the system right-hand side matrix, whereas in the elimination approach it is confined to the first n columns of the extended Jacobian. Note that the matrix $(\mathbf{L}-\mathbf{I}_p)$ is lower triangular and so the product $(\mathbf{L}-\mathbf{I}_p)^{-1}\mathbf{B}$ is determined by forward substitution and not by inverting the matrix and multiplying.

Similarly, the Reverse Mode is equivalent to elimination with the pivots taken in reverse order. This confines fill-in to the **R** and **T** blocks. It is equivalent to solving the transposed system (10) by back-substitution. For our example, this technique also requires 12 flops to calculate the Jacobian.

If full advantage of sparsity is taken, then the forward- and reverse-ordered eliminations will never use more floating-point operations than conventional forward and reverse mode AD, even when conventional techniques use features such as Jacobian compression or a sparse representation of gradients and adjoints [Griewank and Reese 1991].

3.4 Elimination Sequences

From the above discussion it is clear that, instead of using the elimination approach for calculating Jacobians, we could simply implement forward and reverse mode AD and explicitly account for the sparsity of directional derivatives or adjoints. However, an advantage of the elimination approach is that it removes the restriction to monotonic increasing or decreasing pivot orderings and some other *pivot sequence* might be chosen. The use of nonmonotonic pivot sequences is referred to as *cross-country elimination* [Griewank 2000, Chap. 8.] and the extra degrees of freedom so gained give scope for reducing the computational cost. However, we do note that the forward and reverse orderings have the advantage of confining fill-in to blocks **B** and **R** or **R** and **T**. Another pivot ordering is likely to create fill-ins in block **L**. This block is usually far larger than **B**, **R** or **T**, since there are usually far more intermediate variables than independent or dependent variables. There is therefore a danger of producing a large number of fill-ins, which must be removed later in the elimination process and which would compromise efficiency.

The problem of choosing a good pivot sequence for Gaussian elimination of sparse matrices has been well studied [Duff et al. 1989]. One of the earliest

pivot ordering heuristics is due to Markowitz [1957]. The Markowitz heuristic involves, at each elimination step, choosing the next pivot to minimize the product of the number of other entries in its row and the number of other entries in its column. This product, known as the *Markowitz cost*, is an upper bound both on the fill-in at the next elimination step and on the number of multiplications required (noting that pivots are all -1 and that some of the other entries may have the value 1 or -1). Although the Markowitz heuristic is often very successful in reducing the operations count, there are simple examples [Griewank 2000, p. 179] for which it is found to be suboptimal. In his thesis, Naumann [1999] suggested the use of what he called the *Vertex Lowest Relative* (VLR) heuristic also termed *Relatively Greedy Markowitz* by Griewank [2000, p. 179]. This VLR heuristic cost is the difference between the current (Markowitz) cost of a candidate pivot and the cost of eliminating it last in the elimination sequence. We have previously noted [Tadjouddine et al. 2001] that the VLR heuristic tends to eliminate those intermediate variables calculated near the start or end of the code after those in the middle.

In the event of two or more candidate pivots having an equal minimum Markowitz or VLR cost, then a tie-breaking strategy must be used. Griewank and Reese [1991] selected the candidate that resulted in most entries in the extended Jacobian being removed. In our work, we choose the last pivot with minimum cost [Tadjouddine et al. 2001].

3.5 Roundoff

The accuracy of automatic differentiation necessarily depends on the accuracy of the calculation of the function and its local derivatives $c_{i,j}$. If this is ill conditioned, as for example when the "wrong" method for the calculation of a small root of a quadratic equation is employed, the derivative calculation will be ill conditioned too. However, it is important to know if damage might be inflicted by a poor choice of elimination sequence. One of the authors [Reid 2003] has considered this.

The forward method is just forward substitution applied to the system (5) and standard backward error analysis [Wilkinson 1965, pp. 247–248] shows that, for each column of $\mathbf{P}$, we will have solved a nearby problem

$$(\mathbf{C} + \delta\mathbf{C} - \mathbf{I}_N)\mathbf{d} = -\mathbf{p}, \tag{19}$$

with $|\delta c_{i,j}| \leq (3r/2 + 3)|c_{i,j}|$, where r is the largest number of entries in a row of $\mathbf{C}$. Although the perturbations will differ from column to column, we will in each case have done no worse than we would have done for an exact calculation for very slightly perturbed local derivatives $c_{i,j}$. The same very satisfactory result holds for the backward method since it is back-substitution applied to the system (10).

Unfortunately, Reid [2003] has found an (admittedly artificial) example that shows that another pivot sequence can be unstable. If the entries of $\mathbf{C}$ are

$$c_{2k,2k-1} = 2.0,\ c_{2k+1,2k-1} = 2.0,\ c_{2k+1,2k} = -1.0,\ k = 1, 2, \ldots, l, \tag{20}$$

the pivot sequence $1, 3, \ldots, 2l-1, 2, 4, \ldots, 2l$, leads approximately to the doubling of the size of matrix entries at each stage until $2l-1$. The matrix $\mathbf{C} - \mathbf{I}$ is

not ill-conditioned, so the final solution is not large, but large rounding errors will have occurred in intermediate stages.

While this negative result leads us to be cautious, we have not encountered instability in practical cases.

3.6 Statement-Level Differentiation

Now consider the statement-level differentiated version of the code, ignoring the possibility of array operations permitted, for example, in the Fortran 95 language. One approach, adopted by TAMC [Giering and Kaminski 1998], is to differentiate each statement symbolically. This has the potential disadvantage of generating local derivative code with many common subexpressions; though we would expect these to be eliminated by today's optimising compilers. Alternatively, the right-hand side of a statement may be regarded as composed of the corresponding several lines of the code list, eventually assigning one value to the left-hand side. To obtain the local derivatives $c_{i,j}$ associated with the statement, it is a well-established technique to differentiate the local code list using reverse mode AD. This strategy is used by ADIFOR [Bischof et al. 1996a] within an overall forward mode AD approach.

3.7 Reverse Pre-Elimination

One way to order the eliminations within the extended Jacobian of the code-list is to follow ADIFOR by starting with reverse-ordered elimination of the set of intermediate variables within a statement. This may be seen as a sequence of eliminations of intermediate variables with single successors. This is desirable since each such elimination reduces the number of entries in $\mathbf{C}$ by at least one. To see this, suppose the variable has k predecessors and a single successor, consider the square submatrix of order $k+2$

$$\begin{bmatrix} \mathbf{C}_{11} - \mathbf{I}_k & & \\ \mathbf{C}_{21} & -1 & \\ \mathbf{C}_{31} & \mathbf{C}_{32} & -1 \end{bmatrix} \tag{21}$$

corresponding to all the variables involved where the variable itself is in the middle. There can be no fill outside this submatrix since there are no entries outside it in the row and column of the intermediate variable. Furthermore, all fill takes place within $\mathbf{C}_{31}$. Because of our choice of variables, $\mathbf{C}_{21}$ is a full submatrix of order $1\times k$ and $\mathbf{C}_{32}$ is a nonzero 1×1 submatrix. We therefore lose $k+1$ entries when we discard the pivot row and column and end with k entries in $\mathbf{C}_{31}$. Thus the number of entries is reduced by at least one (more if $\mathbf{C}_{31}$ starts as nonzero).

This suggests that it is desirable to eliminate any intermediate variable with a single successor. We have therefore implemented a *reverse pre-elimination* strategy in which we repeatedly consider all intermediate variables starting from the last and proceeding to the first and successively eliminate each one encountered which has a single successor. Note that this will include the elimination from the code-list extended Jacobian of all intermediate variables from within any statement and so mimics the statement-level reverse strategy of

ADIFOR [Bischof et al. 1996a]. Reverse pre-elimination will also eliminate any variable used in the right-hand side of only one other statement and so reduce the number of arithmetic operations, both in the code-list and the statement-level extended Jacobian. In particular, the result of any unary operation will be eliminated, a process referred to as *hoisting* in Bischof [1991].

The elimination AD techniques described above have been implemented in our source transformation AD tool ELIAD which we now describe.

4. THE ELIAD TOOL

ELIAD is a vertex elimination AD tool, written in Java and uses a front-end (parser) and back-end (pretty-printer) generated by ANTLR [Parr et al. 2000]. Though we still regard ELIAD as a proof-of-concept tool, it is available from the authors. It uses source-transformation to convert Fortran code for a function **f** into Fortran code for **f** and its Jacobian. ELIAD performs a bi-directional data flow analysis to determine active variables from user specified independent and dependent variables [Hascoet et al. 2003]. It then performs a symbolic differentiation of each statement of the function (or its code-list) to obtain each subdiagonal entry $c_{i,j}$ of the extended Jacobian. All such entries, and those generated by fill-in during elimination, become separate (scalar) Fortran variables, for example, $c_{7,1}$ might appear in the code as the variable `c_7_1`. Each elimination operation, following a chosen pivot sequence, becomes a set of separate Fortran statements, for example, the fill-in $c_{7,1} = c_{7,4}c_{4,1}$ becomes

```
c_7_1 = c_7_4*c_4_1
```

Work is in hand to improve ELIAD's ability to suppress the multiplication if one of the operands is known a priori to be ± 1 and perform other such optimizations.

ELIAD allows arrays as input arguments provided their indexing is static, that is, can be calculated a priori. In effect, ELIAD unrolls them. For instance, if the inputs are two arrays each of length 5, then their elements become input variables x_1 to x_5 and x_6 to x_{10}, respectively.

ELIAD allows branching [Tadjouddine et al. 2003], though no such test problems are considered in this paper. Results for one test problem may be found in Forth and Tadjouddine [2003]. Currently, no kind of loop is supported.

The chief advantage of this approach is that it permits sparsity in the elimination to be exploited to the maximum. A disadvantage is that the sparsity pattern of the matrix **C** (or a modest overestimate) must be known a priori. Also, though the generated code runs very fast, its length is roughly proportional to the number of elimination operations, which may be expected to grow more than linearly in the length of the **f** code. However, as we shall see in Section 6, ELIAD has been applied successfully to loop-free subroutines with between 8 to 134 independent, 5 to 252 dependent and over 1000 intermediate variables.

After ELIAD has built the extended Jacobian, it applies some algorithm—currently an external program—to determine a good pivot sequence using the Markowitz or a related heuristic, and uses this to generate the Jacobian code.

4.1 Generating the Jacobian Code

Using the abstract syntax tree of the input code [Aho et al. 1995] and associated symbolic information, ELIAD intersperses the original code with assignments that compute local partial derivatives statement by statement. Then, using the given elimination sequence, it generates a series of scalar assignments that eliminate all coefficients of intermediate variables from the extended Jacobian. Each statement computes a new entry or updates an existing entry of the extended Jacobian as may be seen in the following example.

Example 4.1 (*Jacobian Code*). The Jacobian code using the reverse ordering—that is, eliminating intermediate variables u_6, u_5 and u_4 in that order—from the five nontrivial assignments of Example 2.2 could be built up as follows:

```
! Calculate the values of the variables and local derivatives
c_4_1 = 1/u_1;   c_4_2 = 1/u_2
u_4 = log(u_1*u_2)
c_5_2 = u_3**2;   c_5_3 = 2*u_2*u_3
u_5 = u_2*u_3**2-a
c_6_2=1/u_3;  c_6_3=-u_2/u_3**2;  c_6_4=b
u_6 = b*u_4+u_2/u_3
c_7_2=-1;   c_7_4=2*u_4;   c_7_5=1
u_7 = u_4**2+u_5-u_2
c_8_5=-1;  c_8_6=1/(2*sqrt(u_6))
u_8 = sqrt(u_6)-u_5
! Eliminate entries in row 6
c_8_2=c_8_6*c_6_2;  c_8_3=c_8_6*c_6_3; c_8_4 = c_8_6*c_6_4
! Eliminate entries in row 5
c_7_2 = c_7_2+c_7_5*c_5_2;  c_7_3=c_7_5*c_5_3
c_8_2=c_8_2+c_8_5*c_5_2;  c_8_3=c_8_3+c_8_5*c_5_3
! Eliminate entries in row 4
c_8_1 = c_8_4*c_4_1;  c_8_2=c_8_2+c_8_4*c_4_2
c_7_1 = c_7_4*c_4_1;  c_7_2=c_7_2+c_7_4*c_4_2
```

This leaves the Jacobian's entries in variables `c_7_1`, `c_7_2`, `c_7_3`, `c_8_1`, `c_8_2`, and `c_8_3`.

As a final step, ELIAD generates assignments that copy the Jacobian values into an array before exiting the Jacobian code.

There is much freedom in the order in which the above statements can be placed. For example, `c_4_1` and `c_4_2` are not used until the last two lines. It was found that strategies to reorder the Jacobian code can significantly affect performance on some platforms, see Section 6.1.4.

5. TEST ENVIRONMENT

We wish to compare the performance of Jacobian code produced by our vertex-elimination AD tool ELIAD with that produced by hand and by the conventional AD tools ADIFOR and TAMC.

We selected the five platforms described in Appendix A. We regard these as typical of those presently in use for scientific computing. All the processors are termed *superscalar*, being able to perform several instructions (e.g., adds, multiplies, and loads from memory to arithmetic registers) in parallel via so-called *pipelines*. For arithmetic to be performed in a floating-point pipeline, the necessary data must reside in one of the small number of arithmetic registers. All the processors have a memory hierarchy with relatively fast transfer of data between the arithmetic registers and the level-1 cache. Required data not in the level-1 cache must be transferred from the larger level-2 cache at a slower rate. If the data is not currently in the level-2 cache, it must be transferred from the main memory at an even slower rate.

The optimizing compilers available for these platforms seek to maximise performance by rescheduling arithmetic operations to minimise the number of data transfers between registers and cache. A major constraint in such optimizations is that if another instruction calls for a value not yet loaded to a register, a so-called *stall* occurs and the processor must wait. The ALPHA, SGI and AMD platforms of our study feature *out-of-order execution*, in which the processor maintains a queue of arithmetic operations so that if the one at the head of the queue stalls, it can switch to an operation in the queue that is able to execute. As a result, the efficiency of code running on these highly sophisticated processors is less dependent on compiler optimisation than for other processors. More details on such issues may be found in Goedecker and Hoisie [2001].

Of our five test cases, four are taken from the MINPACK-2 collection [Averick et al. 1992] of optimization test problems. Hand-coded Jacobian code is provided. The supplied subroutines include branching to allow for the calculation of the function alone, the function and its Jacobian, or a standard optimization start point $\mathbf{x}$. So that measured CPU times do not include the cost of the (expensive) branching, three new subroutines were created from each original MINPACK subroutine to perform these tasks separately. ELIAD cannot deal with loops at present, so a PERL script was used to unroll them all. Two of the test cases (see Section 6.2.3, Section 6.2.4) have sparse Jacobians. In these two cases all assignments of zero values to the Jacobian were removed. The nominal cost in floating-point operations $W(\mathbf{f})$ was obtained from the modified MINPACK code by using a PERL script to count the number of times that `*`, `+` , `-` and `/` operations appear in the source code.

On all platforms, it is possible to arrange compilation so that subroutines are *inlined* [Goedecker and Hoisie 2001, p. 77], that is, the compiler inserts the body of the subroutine directly into the calling routine. Inlining removes the overhead associated with the subroutine call and so improves efficiency. Inlining may dramatically increase the size of the overall program and so may not be possible for larger subroutines. In our test cases, the function evaluation subroutines are sufficiently small as to be readily inlined. In contrast, the Jacobian evaluation routines are larger which typically prevents their inlining. It is usual in AD efficiency analysis to examine the ratio of Jacobian to function evaluation times. Because this ratio would be severely distorted by inlining of the function evaluation and not the Jacobian we explicitly prevented inlining

by compiling all subroutines individually and, on our SGI platform, turning off interprocedural optimizations.

For each test problem, a driver program was written to execute and time different Jacobian evaluation techniques. To check that the Jacobians were calculated correctly, we compared each to one produced by the hand-coded routine if available or ADIFOR otherwise. The checks were made outside the code that we timed to avoid distorting the times. Each set of values for the independent variables was generated by using the Fortran intrinsic routine `random_number`.

The CPU timers on these processors are not able to time short executions with good relative accuracy. It is therefore necessary to calculate many Jacobians in each case. Simply repeating the calculation for a single $\mathbf{x}$ might give unreasonably short times since it would allow more use of level-1 cache than would be possible in a genuine application. Therefore, for each test problem and each platform, we generated and stored many (N_{evals}) vectors $\mathbf{x}$ and calculated Jacobians for them all.

For all the test problems considered, we found that as N_{evals} was increased there came a point where the average time for a Jacobian calculation would markedly increase. By considering the storage requirements for the sets of independent variables, dependent variables and Jacobians, we found that this increase coincided with the storage requirements increasing beyond that of the level-2 cache. Consequently, to ensure that timings are realistic, we chose N_{evals} for each platform and each problem such that all data associated with calculation and storage comfortably fits in the level-2 cache. Since this did not involve enough computation for accurate timing, we repeated this process a number (N_{repet}) of times. Values of N_{evals} and N_{repet} used for each platform and each of the test problems of Section 6 are given in Table XII of Appendix B.

Even with this two-level set of repeated calculations, we found that occasionally the times varied from run to run. Usually, there were two distinct sets of times, with little variation within each set. We believe that this effect is caused by the different placement of arrays in memory at load time affecting the way data is moved in and out of the caches. We therefore ran each test ten times and report the average.

6. TEST-PROBLEMS AND ALGORITHM ENHANCEMENTS

In Section 6.1, we describe in detail the performance of Jacobian code for the the Roe flux CFD problem [Roe 1981], generated both by conventional means and via ELIAD. We have considered this problem previously [Tadjouddine et al. 2002], though not in conjunction with the pre-elimination technique of Section 3.7 which, together with a more careful choice of N_{evals} (cf. Section 5 and Table XII), has improved the timings obtained. Section 6.2 presents the four test cases for which hand-coded Jacobian code is available. In Section 6.3, we discuss common issues arising from all five test cases. Note that function nominal flops $W(\mathbf{f})$, numbers of lines of noncomment code (l.o.c) and measured CPU times for all test cases are presented in Table XIII of Appendix B.

Table I. Ratios of Jacobian to Function Flop Counts, Number of Lines of Code (l.o.c.) and Ratios of Jacobian to Function CPU Times for Roe Flux

Technique	flops ratio $W(\nabla\mathbf{f})/W(\mathbf{f})$	l.o.c.	Ratios of CPU times ALPHA	SGI	Ultra10	PIII	AMD
ADIFOR	15.95	637	9.50	12.87	36.19	8.94	9.07
TAMC-F	21.18	499	9.79	13.15	13.11	9.65	9.80
TAMC-R	12.69	816	7.94	11.16	15.76	11.00	8.40
FD	12.14		11.80	12.14	11.52	11.57	10.91
VE-SL-F	8.89	1534	4.88	6.74	14.12	8.77	5.85
VE-SL-R	7.32	1274	4.25	5.80	9.01	4.87	4.87
VE-CL-F	12.85	3175	4.59	6.50	20.56	12.16	7.49
VE-CL-R	9.50	2433	4.10	5.98	8.66	8.44	5.66
VE-SLP-F	7.85	1412	4.52	5.42	8.24	6.29	5.01
VE-SLP-R	6.78	1261	4.19	4.81	7.58	4.55	4.50
VE-CLP-F	8.35	2123	4.71	5.49	7.74	6.95	5.27
VE-CLP-R	7.28	1917	3.99	5.11	7.21	4.85	4.70
VE-SLP-F-DFT	7.85	1412	4.53	5.70	9.29	5.05	5.32
VE-SLP-R-DFT	6.78	1261	3.99	4.97	7.11	4.55	4.71
VE-CLP-F-DFT	8.35	2123	4.43	5.74	9.21	6.81	5.74
VE-CLP-R-DFT	7.28	1971	4.07	5.45	7.24	4.69	4.96
VE-SLP-Mark	7.35	1317	4.52	5.44	8.57	5.32	4.88
VE-SLP-VLR	6.60	1222	**3.96**	5.27	7.08	4.33	**4.38**
VE-CLP-Mark	7.86	2026	4.56	5.27	9.18	6.31	5.07
VE-CLP-VLR	7.11	1933	4.12	**4.75**	7.65	4.44	4.56
VE-SLP-Mark-DFT	7.35	1317	4.52	5.66	9.96	4.80	4.82
VE-SLP-VLR-DFT	6.60	1222	4.33	5.07	7.68	**4.20**	4.39
VE-CLP-Mark-DFT	7.86	2026	4.57	5.13	9.41	5.63	5.32
VE-CLP-VLR-DFT	7.11	1933	4.15	5.19	**6.93**	4.67	4.68

6.1 Roe Flux

Table I presents performance data for several techniques applied to calculating the 5×10 dense Jacobian $\nabla\mathbf{f}(\mathbf{x})$ of the Roe flux function [Roe 1981]. For each technique, we give $W(\nabla\mathbf{f})/W(\mathbf{f})$, the ratio of the nominal number of floating-point operations within the generated Jacobian code to those in the function code (computed by a simple PERL script), the numbers of (noncomment) lines in the Jacobian code (l.o.c.) and the corresponding ratios of CPU times. The operations counts $W(\nabla(\mathbf{f}))$ and $W(\mathbf{f})$ will, in general, be over-estimates of the number of floating-point operations performed since they do not take account of optimizations performed by the compiler. In particular, the ELIAD generated Jacobian code may contain statements assigning an entry of the extended Jacobian to be a trivial 1 or -1 and elsewhere use this entry to multiply others. Also, such an entry may be added to another trivial entry, resulting in a zero or non-trivial integer value. We assume the compiler performs *constant value propagation and evaluation* [Goedecker and Hoisie 2001, p. 32] to avoid such unnecessary arithmetic operations. For each platform, the entry corresponding to the AD technique with the smallest ratio of CPU times is highlighted in bold and any entry with a ratio that is nearly as small is underlined.

6.1.1 *Established Techniques.* The first four rows of Table I correspond to established techniques: the use of the AD tools ADIFOR and TAMC in forward

mode, TAMC in reverse mode, and one-sided finite differencing. Specifically, ADIFOR refers to ADIFOR 2.0D [Bischof et al. 1998] generated forward mode AD Jacobian code using the following options: `AD_EXCEPTION_FLAVOR = performance` to avoid any calls to ADIFOR's exception handling library; and `AD_SUPPRESS_LDG = true`; `AD_SUPPRESS_NUM_COLS = true` to ensure that all loops within the ADIFOR generated code are of fixed length and hence are candidates for unrolling at high levels of compiler optimization. TAMC-F and TAMC-R refers to TAMC generated forward and reverse mode AD code respectively, both obtained with the option `-jacobian`.

6.1.2 *Vertex Elimination with Forward and Reverse Orderings.* The next four rows, labelled VE-SL-F to VE-CL-R correspond to the vertex elimination (VE) AD code generated by ELIAD. The first pair use statement-level (SL) differentiation and the second pair use code-list (CL) differentiation. In each of these two pairs, the first uses the forward (F) elimination ordering and the second the reverse (R) ordering. It is seen that for each platform, with the exception of the Ultra10, the best of these four results is about twice as fast as the best established technique. For this problem $m < n$ and, as we might expect, the reverse ordered elimination always out-performs the forward ordered with both statement-level and code-list differentiation; TAMC is not as consistent in this respect. On the Ultra10, PIII and AMD platforms the code-list differentiated variants are often significantly less efficient than their statement-level differentiated counterparts.

6.1.3 *Reverse Pre-Elimination.* In Table I, rows VE-SLP-F and VE-SLP-R correspond to a statement-level symbolic differentiation, followed by a reverse pre-elimination (Section 3.7) and then a forward- and reverse-ordered eliminations of all remaining vertices. Rows VE-CLP-F and VE-CLP-R are the code-list differentiated equivalents. We see that reverse pre-elimination reduces the nominal flops count and subsequently improves performance for all bar one case.

6.1.4 *Depth-First Traversal (DFT) Statement Re-Ordering.* In Section 5, we briefly described how optimising compilers may re-schedule floating-point operations in an algebraically consistent manner, in an attempt to keep the processor's floating-point pipelines full and improve performance. We also remarked that for those platforms that support out-of-order execution, this optimization is less important. In Tadjouddine et al. [2002], we reported that changing the order of the statements in the Jacobian code generated by ELIAD could dramatically affect the performance of the code on platforms that do not support floating-point out-of-order execution. We conjectured, and showed for one example, that this was due to better instruction scheduling resulting in fewer reads and writes from cache to registers and hence fewer stalls in the floating-point pipelines.

To assess the impact of statement ordering in the derivative code, we reordered the assignment statements without altering the data dependencies within the code, with the aim of using each assigned value soon after its assignment. This was done by a using modified version of the depth-first traversal

(DFT) algorithm [Knuth 1997]. Namely, we regarded the statements in the derivative code as the vertices of an acyclic graph, with the output statements at the top and an edge from s to t if the variable assigned by statement s appears in the right-hand side expression in statement t. Then, we arranged the statements in the postorder produced by a depth-first traversal of this graph.

Rows VE-SLP-F-DFT to VE-CLP-R-DFT of Table I show the result of employing our DFT strategy to the codes of rows VE-SLP-F to VE-CLP-R. In contrast to previous results [Tadjouddine et al. 2002], obtained without pre-elimination, we see that the results of employing DFT are mixed. For example, timings on the PIII platform are improved but those on the SGI and AMD are worsened.

6.1.5 *Cross-Country Vertex Elimination.* Rows VE-SLP-Mark to VE-CLP-VLR of Table I show the result of employing the Markowitz and VLR heuristics of Section 3.4 to order the vertex elimination, after the reverse pre-elimination of Section 3.7. The resulting VE-SLP-VLR has the lowest nominal flop count obtained for this problem and is the fastest executing on the ALPHA and AMD platforms. The VE-CLP-VLR case also has a low nominal flop count and is fastest on the SGI.

Employing DFT reordering to the cross-country elimination Jacobian code (rows VE-SLP-Mark-DFT to VE-CLP-VLR-DFT) again had mixed results. However, on the Ultra10 and PIII platforms use of DFT ensures that one of the low flop cross-country techniques, VE-CLP-VLR-DFT on Ultra10 and VE-SLP-VLR-DFT on PIII, becomes the fastest.

6.2 Further Test Cases

Given the encouraging results for the Roe case, we now consider four further test cases taken from Averick et al. [1992] and for which hand-coded Jacobians are available.

6.2.1 *HHD—Human Heart Dipole.* Table II presents performance data for calculating the dense 8×8 Jacobian of this test problem. The same techniques are used as in Section 6.1 with the addition of a row for the hand-coded Jacobian results. Note that Markowitz and VLR orderings when applied to either the statement-level or code-list differentiation produced equivalent elimination sequences (the codes differed only in statement ordering) and so only the Markowitz results are shown.

6.2.2 *CPF—Combustion of Propane (Full Formulation).* Table III gives data for calculating the 53 entry, 11×11 Jacobian of the CPF problem. The Markowitz and VLR orderings are equivalent to the forward ordering and so not shown.

6.2.3 *CTS—Coating Thickness Standardization.* The CTS problem has a 252×134 sparse Jacobian with a maximum of 6 entries per row and 63 per column. The coding is such that a statement-level differentiation has no intermediate variables, that is, $p = 0$. In contrast, the code-list uses $p = 1386$ intermediates. Table IV gives the nominal flop and CPU ratios as for our previous examples, except that compression is used for the conventional methods:

Table II. Ratios of Jacobian to Function Flop Counts, Numbers of Lines of Code (l.o.c.) and Ratios of Jacobian to Function CPU Times for Human Heart Dipole Problem

Technique	flops ratio $W(\nabla\mathbf{f})/W(\mathbf{f})$	l.o.c.	Ratios of CPU times ALPHA	SGI	Ultra10	PIII	AMD
Hand-coded	2.00	205	2.97	3.88	4.79	2.22	2.64
ADIFOR	13.88	229	15.30	21.26	18.61	14.59	13.80
TAMC-F	18.31	196	16.93	24.27	24.85	16.62	14.90
TAMC-R	20.79	324	20.25	37.82	27.07	26.36	23.09
FD	9.19		12.26	12.58	13.46	10.90	12.28
VE-SL-F	3.05	362	3.05	4.28	3.75	3.32	3.82
VE-SL-R	3.00	356	3.26	3.92	3.83	3.35	3.79
VE-CL-F	5.00	826	2.92	4.50	4.01	3.83	4.26
VE-CL-R	3.95	721	2.79	4.01	3.52	3.74	4.33
VE-SLP-F	3.05	348	2.75	3.83	3.50	**3.21**	3.73
VE-SLP-R	3.05	349	**2.61**	3.76	3.43	3.29	3.70
VE-CLP-F	3.90	706	2.79	3.90	3.80	3.61	4.24
VE-CLP-R	3.90	707	2.79	3.74	**3.19**	3.63	4.09
VE-SLP-F-DFT	3.05	348	2.99	3.80	3.54	3.22	3.52
VE-SLP-R-DFT	3.05	349	2.62	**3.71**	3.88	3.28	3.53
VE-CLP-F-DFT	3.90	706	3.19	3.88	3.86	3.56	3.80
VE-CLP-R-DFT	3.90	707	3.25	3.93	**3.19**	3.43	**3.46**
VE-SLP-Mark	3.00	344	2.70	3.85	3.36	3.26	3.69
VE-CLP-Mark	3.86	701	2.82	3.91	4.02	3.83	4.21
VE-SLP-Mark-DFT	3.00	344	2.99	3.81	3.53	3.29	3.54
VE-CLP-Mark-DFT	3.86	701	3.17	3.88	3.86	3.43	3.77

Table III. Ratio of Jacobian to Function Flop Counts, Number of Lines of Code (l.o.c.) and Ratio of Jacobian to Function CPU Times for Combustion of Propane (Full Formulation)

Technique	flops ratio $W(\nabla\mathbf{f})/W(\mathbf{f})$	l.o.c.	Ratios of CPU times ALPHA	SGI	Ultra10	PIII	AMD
Hand-coded	2.24	237	1.97	2.42	3.66	2.23	2.86
ADIFOR	14.44	517	5.89	6.63	10.69	5.76	8.81
TAMC-F	24.76	115	6.60	7.41	11.52	6.95	12.92
TAMC-R	27.03	163	9.49	10.49	19.85	9.85	12.53
FD	15.56	—	14.56	13.70	14.15	13.43	14.42
VE-SL-F	3.04	342	1.77	1.91	2.53	2.15	3.12
VE-SL-R	2.41	291	1.82	1.94	2.60	2.23	3.14
VE-CL-F	3.81	607	1.77	1.91	2.55	2.23	3.24
VE-CL-R	2.78	523	1.68	1.89	2.35	2.18	3.11
VE-SLP-F	2.37	225	1.74	**1.49**	1.93	2.02	2.94
VE-SLP-R	2.40	226	1.66	1.64	1.84	2.04	2.93
VE-CLP-F	2.75	453	1.62	1.50	1.87	1.97	2.86
VE-CLP-R	2.78	455	1.64	1.60	1.85	1.98	2.86
VE-SLP-F-DFT	2.37	225	1.37	1.53	1.85	1.85	**2.27**
VE-SLP-R-DFT	2.40	226	1.34	1.52	1.95	1.88	**2.27**
VE-CLP-F-DFT	2.75	453	**1.32**	1.72	1.89	**1.84**	2.32
VE-CLP-R-DFT	2.78	455	1.35	1.54	**1.81**	**1.84**	2.31

ADIFOR(cmp), TAMC-F(cmp) and FD(cmp). We use the DSM software [Coleman et al. 1984] to obtain a column compression for the supplied sparsity pattern allowing the Jacobian to be reconstructed from $q = 6$ Jacobian-vector products. Note that since we unroll loops prior to differentiation for ELIAD's benefit,

Table IV. Ratios of Jacobian to Function Flop Counts. Numbers of Lines of Code (l.o.c.) and Ratios of Jacobian to Function CPU Times for Coating Thickness Standardization

Technique	Ratios of flops $W(\nabla\mathbf{f})/W(\mathbf{f})$	l.o.c.	Ratios of CPU times ALPHA	SGI	Ultra10	PIII	AMD
Hand-coded	1.85	1419	4.62	3.79	9.63	3.89	3.18
ADIFOR(cmp)	6.38	2311	37.42	48.84	75.80	25.40	30.23
TAMC-F(cmp)	11.15	1688	33.19	48.90	74.81	25.40	30.96
FD(cmp)	6.00		33.35	32.52	49.91	19.73	27.01
VE-SL	1.85	2468	**4.07**	4.12	**7.28**	4.30	3.09
VE-CL-F	3.23	11160	4.08	4.07	9.48	5.47	4.08
VE-CL-R	2.54	9800	4.16	**4.06**	9.66	5.20	3.78
VE-SL-DFT	1.85	2468	4.85	4.86	9.96	**3.78**	**2.60**
VE-CL-F-DFT	3.23	11160	4.75	4.83	8.48	4.36	3.11
VE-CL-R-DFT	2.54	9800	4.74	4.82	9.31	4.36	2.87

then for consistency we apply ADIFOR and TAMC to the function coding after loop-unrolling. In practice, this made little difference to CPU times but does lead to a large number of lines of code as seen in Table IV. The application of ADIFOR with the SPARSLINC library for sparse storage of gradients was tried on the ALPHA and Ultra10 platforms, but results were disappointing being approximately 10 times slower than compressed finite differencing.

The row labelled VE-SL refers to vertex elimination AD with differentiation at the statement level. Since there are no intermediate variables for statement-level differentiation of this problem, as explained above, the partial derivatives of each statement correspond to entries in the Jacobian. With reference to (16), the subblocks **B**, **L** and **T** of the extended Jacobian are empty and subblock **R** is the required function Jacobian. No linear solve is required, $W(\text{linear solve}) = 0$, and the computed statement partial derivatives are inserted directly into the Jacobian. The equality of "Ratios of flops" for Hand-coded and VE-SL indicates their equivalence. This is not true for conventional AD. For example, in forward mode AD the coefficients **R** multiply the length n (or length q for compression) vectors of the $\nabla x_i,\ i = 1, \ldots, n$.

The reverse pre-elimination of Section 3.7 is not necessary on this problem since it will produce the same elimination sequence as the reverse ordering [Naumann 1999, p. 55]. No cross-country elimination sequences were used for this problem since use of reverse pre-elimination alone would allow for Jacobian evaluation.

6.2.4 *FIC—Flow in Channel.* For this problem the supplied subroutine uses $p = 680$ intermediate variables and its code-list $p = 1328$. The resulting 32×32 Jacobian is sparse with a maximum of 9 nonzeros per row and per column. Table V gives the nominal flop and CPU ratios as before. For the established techniques, we again use the DSM software to enable row compression of the Jacobian calculation using $q = 9$ Jacobian-vector products and the function is unrolled prior to differentiation. An interesting feature of the FIC problem is that each of its intermediate variables is used in only one other statement. Consequently, reverse pre-elimination alone eliminates all intermediate variables and is equivalent to the reverse orderings VE-SL-R and VE-CL-R. Because of

Table V. Ratios of Jacobian to Function Flop Counts, Numbers of Lines of Code (l.o.c.) and Ratios of Jacobian to Function CPU Times for Flow in Channel

Technique	Ratios of flops $W(\nabla\mathbf{f})/W(\mathbf{f})$	l.o.c.	Ratios of CPU times ALPHA	SGI	Ultra10	PIII	AMD
Hand-coded	1.91	1786	2.44	2.72	8.33	3.34	2.58
ADIFOR(cmp)	10.44	2989	15.50	57.93	38.48	47.68	59.52
TAMC-F(cmp)	10.85	3508	15.54	56.66	38.57	46.84	57.95
FD(cmp)	9.20		15.06	17.61	18.59	30.24	17.42
VE-SL-F	3.49	6300	2.21	3.08	3.82	**4.18**	2.78
VE-SL-R	2.25	4420	2.14	3.10	4.05	5.12	3.66
VE-CL-F	**4.44**	9982	2.33	3.26	4.94	4.87	3.35
VE-CL-R	2.75	7411	2.12	3.09	**3.41**	5.17	3.33
VE-SL-R-DFT	2.25	4420	**2.10**	**2.97**	4.89	4.72	2.67
VE-CL-R-DFT	2.75	7411	2.12	3.02	6.80	4.91	**2.56**

this, no pre-eliminated results are shown. We also give results for the DFT code-reordering applied to these two cases. No cross-country elimination results are shown since they could not improve on pre-elimination.

6.3 Discussion of the Results

We now discuss the results of Tables I to V.

6.3.1 *Forward and Reverse Vertex Elimination.* In Tables I to V, the ratio of the nominal operations count $W(\nabla(\mathbf{f}))/W(\mathbf{f})$ for rows VE-SL-F to VE-CL-R indicates that forward and reverse vertex elimination should give much improved performance compared to the established techniques and this is confirmed in the run-time ratios. Indeed on the ALPHA and Ultra10 platforms, the vertex elimination techniques almost always out-perform the hand-coded Jacobian code, where available, and the best always does so.

6.3.2 *Forward and Reverse Vertex Elimination with Reverse Pre-Elimination.* The difference in nominal flops between the code-list and statement-level differentiated Jacobian codes may be greatly reduced by using reverse pre-elimination (Section 3.7). For example, compare VE-CL-F/VE-CL-R and VE-CLP-F/VE-CLP-R in Tables I to II. The reduction is particularly noteworthy for the forward orderings (VE-CL-R, VE-CLP-F) of these tables. Pre-elimination may also improve a statement-level differentiated code in cases where an active intermediate variable of the original function code is only used once. See, for example, Table I.

In terms of run-time, the application of pre-elimination usually improves efficiency. Sometimes the improvement is very substantial (see Table I, Ultra10 and PIII) and sometimes there is hardly any change despite a worthwhile reduction in the number of operations (see Table I, ALPHA). We believe this is associated with how well the compiler optimizes the code. We see that reverse pre-elimination rarely increases run time. We conclude that this is a very worthwhile strategy to use.

6.3.3 *Depth-First Traversal (DFT) Statement Reordering.* The results with depth-first traversal (DFT) statement re-ordering (Section 6.1.4) were very

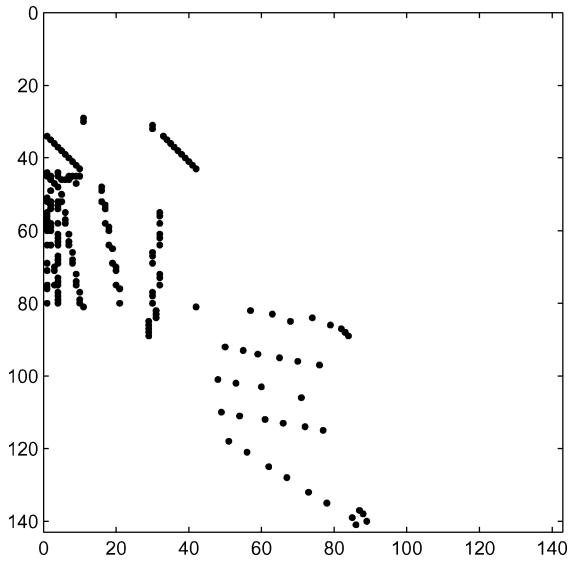

Fig. 2. Variable usage in the CPF Jacobian code VE-SLP-F. Each row corresponds to a statement and its entries indicate the statements that calculated the values of the variables involved.

mixed. Significant gains were sometimes made: ALPHA and AMD of Table III, PIII and AMD of Table IV, PIII and AMD of Table V. Sometimes significant losses were made: ALPHA, SGI of Table IV, Ultra10 of Table V. When successful, DFT appears to act as desired by reducing the number of memory operations performed. For example, in Table V the application of DFT re-ordering to the VE-SL-R subroutine to give VE-SL-R-DFT results in an 8% speed-up on the PIII platform. On examining the associated assembler we find that this is most likely to be due to a 14% drop in the number of loads and 8% drop in the number of stores. Conversely, on the Ultra10 platform we get a 21% reduction in efficiency on applying DFT and this is found to be associated with an undesired 27% increase in the number of stores.

To understand how DFT reordering can reduce reads and writes to and from cache, consider the CPF problem with results of Table III. With only four exceptions, application of DFT reordering improves performance. Figures 2 and 3 show within which statements of the *Jacobian codes* VE-SLP-R and VE-SLP-R-DFT the variables are used. The first empty 29 rows in both plots correspond to the 11 independent variables and 18 constants (which are not calculated). Entries in columns 1 to 29 therefore denote uses of these variables and constants. In Figure 2, rows 30 to 62 correspond to the calculation of the function and local derivatives in the VE-SLP-R code; rows 63 to 87 correspond to the vertex elimination of intermediates from the function's extended Jacobian; and rows 88 to 141 represent the assignment of the calculated scalar Jacobian entries to their matrix storage. Note that several columns are empty and correspond to variables unused in the right-hand side of any other statement, for example, assignments to the Jacobian. We see that the statements using a variable's value are often widely separated from the statement that calculated that value.

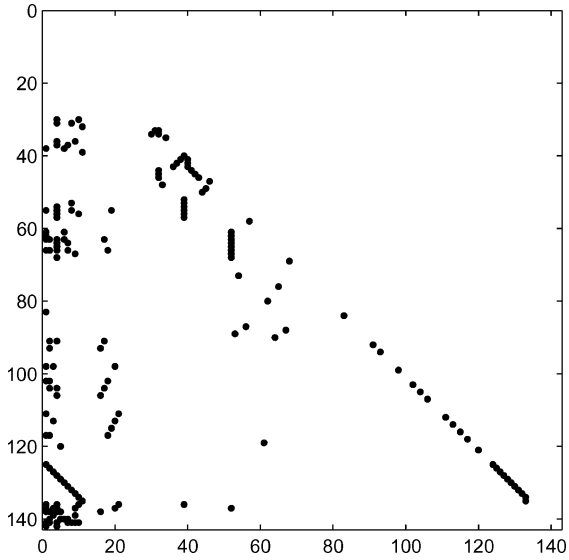

Fig. 3. Variable usage in the the DFT reordered CPF Jacobian code `VE-SLP-F-DFT`. Each row corresponds to a statement and its entries indicate the statements that calculated the values of the variables involved.

For example, statement 141 uses the result of statement 86—separated by 55 other statements. Unless the compiler itself performs significant reordering of the arithmetic instructions of the Jacobian code, then to keep all variables in registers between their calculation and last use will require a large number of registers. In Figure 3, we see that, with only a handful of exceptions, DFT reordering ensures entries of rows 30 to 141 lie either in the first 22 columns, corresponding to uses of independent variables and constants, or are clustered close to the diagonal corresponding to use of a value shortly after its calculation. Consequently if the independents are first read from cache and, together with the problem constants, kept in 22 registers throughout, then only a small number of other registers will be needed to avoid reads and writes to cache other than those necessary to store the function values and Jacobian entries.

In a significant number of cases our best result (highlighted in bold) was obtained with a DFT code, which means that the technique should not be dismissed despite its mixed performance. The technique clearly needs improvement. Its most apparent weakness is that it fails to account for multiple uses of the independent variables and other constants. If, as in Figure 3, such uses are distributed throughout the code, and the numbers of such variables and constants is close to or exceeds the number of registers available, then performance will be impaired by the many loads and stores required.

6.3.4 *Cross-Country Vertex Elimination.* Only Table I shows operation count gains from cross-country vertex elimination that are other than negligible. In Table I, across all platforms, the best results were obtained by cross-country vertex elimination, sometimes aided by DFT statement re-ordering.

Table VI. Size and Number of Entries in Blocks **B**, **L**, **R** and **T** of the Extended Jacobians (Eq. (17)) for Test Cases Differentiated at Statement Level. Figures in Brackets are After Reverse Pre-Elimination

	Size			Number of entries in block			
Problem	n	m	p	**B**	**L**	**R**	**T**
Roe	10	5	62 (36)	39 (39)	140 (88)	5 (5)	14 (37)
HHD	8	8	18 (18)	8 (8)	16 (16)	0 (0)	68 (68)
CPF	11	11	12 (2)	12 (1)	10 (1)	38 (49)	6 (5)
CTS	134	252	0 (0)	0 (0)	0 (0)	882 (882)	0 (0)
FIC	32	32	582 (0)	582 (0)	486 (0)	16 (246)	96 (0)

Why it is only for the Roe case that cross-country elimination produces benefits demands some explanation. Table VI shows the number of entries in each of the blocks of the extended Jacobian as defined by Eq. (17) before and after pre-elimination. For the two sparse cases CTS and FIC of Sections 6.2.3 and 6.2.4, pre-elimination alone obtains the Jacobian in block **R** with other blocks then empty. For the CPF problem of Section 6.2.2, pre-elimination eliminates all but 2 intermediates, so there can only be two subsequent, distinct elimination sequences for the statement-level differentiated code, namely forward and reverse orderings. This explains why use of the Markowitz and VLR heuristics leads to no improvement in the number of flops. After pre-elimination, the HHD problem of Section 6.2.1 still has 18 intermediates and so there is scope for the Markowitz and VLR heuristics to improve efficiency. A small reduction in nominal flops is obtained, though the two sequences are equivalent and performance is not significantly improved.

However, for the Roe problem of Section 6.1, 36 intermediates and 88 entries in block **L** remain after pre-elimination. This greater complexity gives scope for the Markowitz and VLR heuristics to have a greater impact on the number of flops. Hence only for this case do the heuristics result in the most efficient Jacobian code, even so the improvement is small.

6.3.5 *Code-List versus Statement-Level Differentiation.* In Tables I–V we see that the pre-eliminated statement-level differentiated vertex-elimination codes always use fewer nominal flops and, in 38 of the 55 distinct problem/platform combinations (excluding DFT reorderings), out-perform the corresponding code-list differentiated codes.

6.4 Overall Performance of Vertex-Elimination AD

Given the many variations in Jacobian code produced by ELIAD (statement-level or code-list differentiation, pre-elimination, elimination orderings and code re-orderings) it is useful to summarise the performance of the best ELIAD generated Jacobian code with respect to the other calculation methods.

Table VII gives the speed-up obtained in moving from 1-sided finite differencing (using row compression where applicable) to our best vertex elimination method for each problem and each platform. The inherent truncation error associated with finite differencing and the superior efficiency of factors between 1.7 to 11 for the ELIAD generated code appear to justify use of elimination AD

Table VII. Speed-Up of Best Vertex Elimination AD over 1-Sided Finite Differencing

	Platform				
Problem	ALPHA	SGI	Ultra10	PIII	AMD
Roe	3.0	2.6	1.7	2.8	2.5
HHD	4.7	3.4	4.2	3.4	3.5
CPF	11.0	9.2	7.8	7.3	6.4
CTS	8.2	8.0	6.9	4.6	10.4
FIC	7.2	6.2	5.5	7.2	6.9

Table VIII. Speed-Up of Best Vertex Elimination over Best Conventional AD

	Platform				
Problem	ALPHA	SGI	Ultra10	PIII	AMD
Roe	2.0	2.3	1.9	2.1	1.9
HHD	5.9	5.7	5.8	4.5	4.0
CPF	4.5	4.4	5.9	3.2	3.9
CTS	8.2	12.0	10.3	5.9	11.6
FIC	7.4	20.0	11.3	11.2	22.8

Table IX. Speed-Up of Best Vertex Elimination AD over Hand-Coded Jacobian Code

	Platform				
Problem	ALPHA	SGI	Ultra10	PIII	AMD
HHD	1.13	1.05	1.50	0.69	0.76
CPF	1.49	1.62	2.02	1.22	1.26
CTS	1.14	0.93	1.32	1.12	1.22
FIC	1.16	0.96	2.44	0.80	1.02

over finite differencing, though the efficiency of finite differencing would be improved by allowing the compiler to inline function calls.

Table VIII gives the speed-up obtained in moving from the best conventional AD technique to our best vertex elimination method for each problem and each platform. Speed-ups of between 1.9 to 22.8 demonstrate the superiority of vertex-elimination AD over conventional forward and reverse mode AD for Jacobian calculation.

Table IX gives the speed-up obtained in moving from hand-coded Jacobian code to our best vertex elimination method for the 4 problems for which we have hand-coded Jacobians. Only for 5 of the 20 problem/platform combinations is hand-coding superior, and for two of these the discrepancy is within 7%.

7. CONCLUSIONS AND PLANS FOR FUTURE WORK

In this paper, we have presented the first extended set of results from ELIAD, a source-transformation implementation of the vertex-elimination AD approach to Jacobian calculation of functions defined by Fortran code. Careful timings demonstrate an efficiency equal to that obtained by hand-coding and between 2 to 20 times superior to conventional AD techniques. This superiority is due

to ELIAD's fuller exploitation of the sparsity of the extended Jacobian system that governs the relationship between the derivatives of all variables in the function code. Further, this exploitation of sparsity is performed at the Jacobian code generation stage, with statements generated only for those arithmetic operations involving nonzero entries in the extended Jacobian.

Vertex elimination requires an ordering, or pivot sequence, in which to eliminate the derivatives of intermediate variables from the extended Jacobian. Monotonic increasing and decreasing orderings correspond to sparsity-exploiting variants of conventional forward and reverse mode AD [Griewank and Reese 1991]. Consequently, vertex-elimination AD never uses more floating-point operations than conventional AD. The use of cross-country (i.e., nonmonotonic) orderings gives further scope for reducing the number of floating-point operations. We have found that employing a pre-elimination strategy of eliminating any intermediate variable used only once, followed by a forward or reverse ordered elimination of all other intermediates, is very successful, both in reducing the number of floating-point operations and improving run-times. In the future, we wish to improve this strategy in the light of recent work [Naumann 2003]. More involved ordering heuristics, such as the Markowitz and VLR strategies, were only worthwhile for one test case studied to date. Recently, elimination AD has been generalized to edge [Naumann 1999] and face [Naumann 2001, 2004] eliminations, which may further reduce the number of flops required for Jacobian calculation. We are currently assessing these techniques for inclusion into ELIAD.

Interestingly, we found that reordering a Jacobian code's statements frequently affected its performance. This appears to be due to changes in the number of loads and stores from cache to registers in the assembler of the reordered code. We are currently performing an in-depth study of the assembler produced for all the Jacobian codes of our study in order to get a better understanding of this issue and consequently improve both elimination heuristics and our code re-ordering strategy.

For the sparse Jacobian cases (FIC and CTS), the ELIAD generated Jacobian code was between 6 to 20 times more efficient than conventional AD using Jacobian compression. So, vertex-elimination AD appears excellently suited to least squares optimisation problems and numerical PDE solvers, where efficient Jacobian calculation enables fast solution via Newton-like solvers. To be more generally applicable for such problems requires removal of ELIAD's present restriction to loop-free code. This will greatly complicate ELIAD's activity analysis which will have to handle array indices as index ranges and not fixed values as at present. The techniques of Tadjouddine et al. [1998] and Tadjouddine [1999] will be modified to enable this. The ability to store such Jacobians in a suitable sparse matrix format will also be necessary.

At the time of writing, ELIAD's functionality is being extended to include subprograms. A hierarchical approach utilising a conservative activity analysis of all variables in all possible branches and subroutines is adopted. This approach automates and generalizes that of Bischof and Haghighat [1996].

There are still many open questions related to the optimal calculation of Jacobians by elimination, both theoretical and implementational. Despite this,

Table X. Platforms

	a) Processors			
Platform	Processor	CPU	L1-Cache	L2-Cache
ALPHA	EV6	667 MHz	128 KB	8 MB
SGI	R12000	300 MHz	64 KB	8 MB
Ultra10	SUN Ultra10	440 MHz	32 KB	2 MB
PIII	Pentium 3	700 MHz	32 KB	256 KB
AMD	Athlon XP1800+	1533 MHz	128 KB	256 KB

	b) Compilers	
Platform	Compiler	Options
ALPHA	Compaq f95 5.4	-O5 -fast -arch host -tune host
SGI	f90 MIPSPro 7.3	-Ofast -IPA:inline=OFF -INLINE:none
Ultra10	Workshop f90 6.0	-fast
PIII/AMD	Compaq Visual Fortran	/architecture:host /assume:noaccuracy_sensitive /inline:manual /math_library:fast /tune:host /opt:fast

Table XI. Summary Statistics for Test Problem Functions

Problem	n	m	p-SL	p-CL
Roe	10	5	62	208
HHD	8	8	20	84
CPF	11	11	13	57
CTS	134	252	0	1386
FIC	32	32	680	1328

we feel certain that, with such issues now being studied in depth, the days of scientists and engineers painstakingly hand-coding Jacobian code to ensure efficiency are numbered.

APPENDIXES

A. PLATFORMS

We ran test cases on COMPAQ ALPHA, Silicon Graphics and SUN UNIX machines with processor/compiler combinations denoted ALPHA, SGI and Ultra10. We also ran on two PC platforms with Pentium 3 and AMD Athlon processors. Relevant hardware data and the compiler information is given in Table X.

B. TEST PROBLEM FUNCTION STATISTICS, CPU TIMES AND OPERATIONS COUNTS

Table XI gives summary statistics for all 5 test problems. Columns p-SL and p-CL refer to the number of (active) intermediate variables in each function's original statements and in its code-list respectively.

Table XII gives N_{evals}, the number of distinct Jacobians calculated, and N_{repet} the number of times these calculations were repeated, in order to obtain reliable Jacobian timings.

Table XII. Values of N_{evals} and N_{repet} Used for Each Problem and Platform

	N_{evals}		N_{repet}			
Problem	ALPHA, SGI & Ultra10	PIII & AMD	ALPHA	SGI & Ultra10	PIII	AMD
Roe	500	60	2000	2000	16667	16667
HHD	2500	312	400	400	3200	12800
CPF	1000	125	2000	500	4000	16000
CTS	1	1	12000	12000	12000	120000
FIC	200	25	100	100	6400	6400

Table XIII. Nominal Floating-Point Operations Counts $W(\mathbf{f})$, Lines of Code (l.o.c.) and CPU Times for Test Problem Functions

	$W(\mathbf{f})$		Function CPU time(μs)				
Problem	(flops)	l.o.c.	ALPHA	SGI	Ultra10	PIII	AMD
Roe	222	139	0.475	0.951	0.806	0.880	0.336
HHD	84	68	0.121	0.224	0.206	0.231	0.0943
CPF	68	45	0.351	0.870	0.503	0.495	0.162
CTS	1638	535	1.18	3.16	2.25	2.49	0.981
FIC	1266	759	1.53	2.93	2.99	1.94	0.878

The second column of Table XIII gives the nominal computational cost $W(\mathbf{f}(\mathbf{x}))$ of calculating the test problems described in Section 6. It is determined by counting the number of floating operations within the Fortran code using a simple PERL script. Table XIII also gives the number of noncomment lines of code (l.o.c.) and the average CPU times required to calculate the test problem functions $\mathbf{f}(\mathbf{x})$, as measured using the `CPU_TIME` intrinsic function of the Fortran 95 programming language.

ACKNOWLEDGMENTS

We thank Neil Stringfellow and Venkat Sastry for their help in developing the PERL scripts used in this article and the three anonymous referees for the great care with which they read our article and for their many suggestions.

REFERENCES

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1995. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass.

AVERICK, B. M., CARTER, R. G., MORÉ, J. J., AND XUE, G.-L. 1992. The MINPACK-2 test problem collection. Preprint MCS–P153–0692, ANL/MCS–TM–150, Rev. 1, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill. See `ftp://info.mcs.anl.gov/pub/MINPACK-2/tprobs/P153.ps.Z`.

BENDTSEN, C. AND STAUNING, O. 1996. FADBAD, A flexible C++ package for automatic differentiation. Tech. Rep. IMM-REP-1996-17, Technical University of Denmark, IMM, Departement of Mathematical Modeling, Lyngby.

BERZ, M., BISCHOF, C., CORLISS, G., AND GRIEWANK, A., EDS. 1996. *Computational Differentiation: Techniques, Applications, and Tools*. SIAM, Philadelphia, Pa.

BISCHOF, C., BÜCKER, H., LANG, B., RASCH, A., AND VEHRESCHILD, A. 2002. Combining source transformation and operator overloading techniques to compute derivatives for MATLAB programs.

In *Proceedings of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002)*. IEEE Computer Society Press, Los Alamitos, Calif., pp. 65–72.

BISCHOF, C. H. 1991. Issues in parallel automatic differentiation. In *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, A. Griewank and G. F. Corliss, Eds. SIAM, Philadelphia, Pa., 100–113.

BISCHOF, C. H., CARLE, A., HOVLAND, P. D., KHADEMI, P., AND MAUER, A. 1998. ADIFOR 2.0 user's guide (Revision D). Tech. Rep., Mathematics and Computer Science Division Technical Memorandum no. 192 and Center for Research on Parallel Computation Technical Report CRPC-95516-S. See `www.mcs.anl.gov/adifor`.

BISCHOF, C. H., CARLE, A., KHADEMI, P., AND MAUER, A. 1996a. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computat. Sci. Eng. 3*, 3, 18–32.

BISCHOF, C. H. AND HAGHIGHAT, M. R. 1996. Hierarchical approaches to automatic differentiation. In *Computational Differentiation: Techniques, Applications, and Tools*, M. Berz, C. Bischof, G. Corliss, and A. Griewank, Eds. SIAM, Philadelphia, Pa., 83–94.

BISCHOF, C. H., KHADEMI, P. M., BOUARICHA, A., AND CARLE, A. 1996b. Efficient computations of gradients and Jacobians by dynamic exploitation of sparsity in automatic differentiation. *Optimiza. Meth. Softw. 7*, 1–39.

BISCHOF, C. H., ROH, L., AND MAUER, A. 1997. ADIC—An extensible automatic differentiation tool for ANSI-C. *Softw.—Pract. Exp. 27*, 12, 1427–1456. See `www-fp.mcs.anl.gov/division/software`.

COLEMAN, T. F., GARBOW, B. S., AND MORÉ, J. J. 1984. Software for estimating sparse Jacobian matrices. *ACM Trans. Math. Softw. 10*, 3, 329–345.

COLEMAN, T. F. AND VERMA, A. 1998. ADMAT: An automatic differentiation toolbox for MATLAB. Tech. Rep., Computer Science Department, Cornell University.

CORLISS, G., FAURE, C., GRIEWANK, A., HASCOËT, L., AND NAUMANN, U., Eds. 2001. *Automatic Differentiation: From Simulation to Optimization*. Computer and Information Science. Springer, New York.

DUFF, I., ERISMAN, A. M., AND REID, J. 1989. *Direct methods for sparse matrices*. Monographs on numerical analysis. Oxford University Press.

FAURE, C. AND PAPEGAY, Y. 1998. Odyssée User's Guide. Version 1.7. Rapport technique RT–0224, INRIA, Sophia-Antipolis, France. Sept. See `www.inria.fr/RRRT/RT-0224.html`, and `www.inria.fr/safir/SAM/Odyssee/odyssee.html`.

FORTH, S. A. 2001. User guide for MAD—A Matlab automatic differentiation toolbox. Applied Mathematics and Operational Research Report AMOR 2001/5, Cranfield University (RMCS Shrivenham), Swindon, SN6 8LA, UK. June.

FORTH, S. A. AND TADJOUDDINE, M. 2003. CFD Newton solvers with EliAD, an elimination automatic differentiation tool. In *Computational Fluid Dynamics 2002: Proceedings of the 2nd International Conference on Computational Fluid Dynamics, ICCFD* (Sydney, Australia). S. Armfield, P. Morgan, and K. Srinivas, Eds., Springer-Verlag, New York, 134–139.

GIERING, R. AND KAMINSKI, T. 1998. Recipies for adjoint code construction. *ACM Trans. Math. Softw. 24*, 4, 437–474. (Also appeared as Max-Planck Institut für Meteorologie Hamburg, Technical Report No. 212, 1996.)

GOEDECKER, S. AND HOISIE, A. 2001. *Performance Optimisation of Numerically Intensive Codes*. Software, Environments, Tools. SIAM, Philadelphia, Pa., ISBN 0-89871-482-3.

GRIEWANK, A. 2000. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, Pa.

GRIEWANK, A. AND CORLISS, G. F., Eds. 1991. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. SIAM, Philadelphia, Pa.

GRIEWANK, A., JUEDES, D., AND UTKE, J. 1996. ADOL–C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Softw. 22*, 2, 131–167.

GRIEWANK, A. AND REESE, S. 1991. On the calculation of Jacobian matrices by the Markowitz rule. In *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, A. Griewank and G. F. Corliss, Eds. SIAM, Philadelphia, Pa., 126–135.

HASCOET, L., NAUMANN, U., AND PASCUAL, V. 2003. TBR analysis in reverse mode automatic differentiation. In *Future Generation Computer Systems*. Elsevier, Amsterdam, The Netherlands.

HOVLAND, P. D. AND MCINNES, L. C. 2001. Parallel. simulation of compressible flow using automatic differentiation and PETSc. *Parall. Comput. 27*, 4 (Mar.), 503–519.

KNUTH, D. E. 1997. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Adison-Wesley, Reading, Mass.

MARKOWITZ, H. 1957. The elimination form of the inverse and its application. *Manage. Sci. 3*, 257–269.

NAUMANN, U. 1999. Efficient calculation of Jacobian matrices by optimized application of the chain rule to computational graphs. Ph.D. thesis, Technical University of Dresden.

NAUMANN, U. 2001. Elimination techniques for cheap Jacobians. In *Automatic Differentiation: From Simulation to Optimization*, G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, Eds. Springer-Verlag, New York, Chapter 29, 241–246.

NAUMANN, U. 2003. Statement-level optimality of tangent-linear and adjoint models. Argonne National Laboratory, preprint, ANL/MCS-P-1066, June.

NAUMANN, U. 2004. Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph. *Math. Prog. 99*, 3 (Apr.), 399–421.

PARR, T., LILLY, J., WELLS, P., KLAREN, R., ILLOUZ, M., MITCHELL, J., STANCHFIELD, S., COKER, J., ZUKOWSKI, M., AND FLACK, C. 2000. ANTLR Reference Manual. Tech. Rep., MageLang Institute's jGuru.com. January. See `www.antlr.org/doc/index.html`.

PRYCE, J. D. AND REID, J. K. 1998. ADO1, A Fortran 90 code for automatic differentiation. Tech. Rep. RAL-TR-1998-057, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire, OX11 OQX, England. See `ftp://matisa.cc.rl.ac.uk/pub/reports/prRAL98057.ps.gz`.

REID, J. 2003. On the stability of automatic differentiation. In preparation.

ROE, P. L. 1981. Approximate Riemann solvers, parameter vectors, and difference schemes. *J. Computat. Phys. 43*, 357–372.

TADJOUDDINE, M. 1999. La différentiation automatique. Ph.D. dissertation, Université de Nice, Sophia Antipolis, France.

TADJOUDDINE, M., EYSETTE, F., AND FAURE, C. 1998. Sparse Jacobian computation in automatic differentiation by static program analysis. In *Proceedings of the 5th International Static Analysis Symposium*. Number 1503 in Lecture Notes in Computer Science. Springer-Verlag, New York, 311–326.

TADJOUDDINE, M., FORTH, S. A., AND PRYCE, J. D. 2001. AD tools and prospects for optimal AD in CFD flux Jacobian calculations. In *Automatic Differentiation: From Simulation to Optimization*, G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, Eds. Springer-Verlag, New York, Chapter 30, 247–252.

TADJOUDDINE, M., FORTH, S. A., AND PRYCE, J. D. 2003. Hierarchical automatic differentiation by vertex elimination and source transformation. In *Proceedings of Computational Science and Its Applications—ICCSA 2003*. Lecture Notes in Computer Science, vol. 2. Springer-Verlag, New York, 115–124.

TADJOUDDINE, M., FORTH, S. A., PRYCE, J. D., AND REID, J. K. 2002. Performance issues for vertex elimination methods in computing Jacobians using automatic differentiation. In *Proceedings of the 2nd International Conference on Computational Science*, P. M. Sloot, Ed. Lecture Notes in Computer Science, vol. 2. Springer-Verlag, New York, 1077–1086.

VERMA, A. 1998. ADMAT: Automatic differentiation in MATLAB using object oriented methods. In *SIAM Interdisciplinary Workshop on Object Oriented Methods for Interoperability*. SIAM, National Science Foundation, Yorktown Heights, N.Y., 174–183.

WILKINSON, J. 1965. *The Algebraic Eigenvalue Problem*. Oxford University Press.

Received December 2002; revised October 2003 and February 2004; accepted March 2004