Jacobian Code Generated by Source Transformation and Vertex Elimination is as Efficient as Hand Coding

Shaun A. Forth[1,2], Mohamed Tadjouddine[1,2], John D. Pryce[1,2] and John K. Reid[3]

ABSTRACT

This paper 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. We describe the source transformation implementation of our tool ELIAD and present results from 5 test cases, 4 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 5 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.

---

[1] RMCS Shrivenham.

[2] Work funded by the UK's EPSRC and MOD under grant GR/R21882.

[3] JKR Associates and Computational Science and Engineering Department, RAL.

Computational Science and Engineering Department
Atlas Centre
Rutherford Appleton Laboratory
Oxfordshire OX11 0QX

December 13, 2002.

GLOSSARY

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

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. 1998; Giering 1997; 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] 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 **S** with m rows.

Alternatively, the *operator overloading* approach [Griewank 2000, Section 5.1-5.6] utilises the object-oriented features of modern 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 for example [Tadjouddine et al. 2001; 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 paper 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 and Section 7, we present an extended set of results from ELIAD, and we discuss these results in Section 8. 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 9 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 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:

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

independent variables: $\mathbf{x} = (x_i, i = 1, \ldots, n)$ whose values must be supplied, and

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

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. We find it convenient to assume that no expression for a dependent variable involves another dependent variable, that is, if $i > n + p$ and $j \prec i$, then $j \leq n + p$. If need be, this may be ensured by making a copy of any dependent variable used to calculate another dependent variable. We also assume that, apart from the dependent variables, the value of every active variable is referenced at least once in later statements. If such a variable is not referenced, it may be removed completely from the calculation.

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 $\mathbf{C} = \{c_{i,j}\}_{1 \leq i,j \leq N}$ 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

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

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 $N_\mathbf{c}$ to be the number of entries in $\mathbf{C}$.

EXAMPLE 2.3 EXTENDED JACOBIAN SYSTEM. *For our example code, the extended Jacobian system is given by*

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

with nonzero off-diagonal entries in the extended Jacobian given by

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

We see that $N_{\mathbf{c}} = 12$.

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

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

we can express the solution of equation (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 equation (9) as $\nabla\mathbf{f} = \left[(-\mathbf{Q}^T)(\mathbf{C} - \mathbf{I}_N)^{-1}\right]\mathbf{P}$, 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.

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). Consequently, conventional AD tools cannot readily exploit the sparsity of the first n rows of $\mathbf{P}$ or last m rows of $\mathbf{Q}$ and the associated linear solves of (5) or (10) assume that $\mathbf{D}$ or $\mathbf{A}$ is a dense matrix.

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* $12 \times 3 = 36$ *multiplications and* $7 \times 3 = 21$ *additions/subtractions, that is, 57 floating-point operations (flops).*

EXAMPLE 2.5 REVERSE/ADJOINT MODE AD. *For Example 2.3 the adjoint system is solved using* $12 \times 2 = 24$ *multiplications and* $5 \times 2 = 10$ *additions/subtractions, that is, 34 flops.*

For our simple example, reverse mode AD uses fewer arithmetic operations than forward mode.

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 large scale problems with many independent variables ($n \gg 1$) and few constraints, we see that reverse mode is to be preferred.

For sparse Jacobians whose sparsity pattern is known, there are established techniques, collectively termed *Jacobian compression*, for reducing the number of Jacobian-vector products needed to calculate all nonzero entries of $\nabla\mathbf{f}(\mathbf{x})$. It is often possible to find a set $\mathbf{s}_i$, $i = 1, \ldots, q$, $q \ll n$, of n-vectors such that, on calculating all the Jacobian-vector products $\nabla\mathbf{f}(\mathbf{x})\mathbf{s}_i$, it is possible to recover all the nonzero entries of $\nabla\mathbf{f}(\mathbf{x})$. When the $\nabla\mathbf{f}(\mathbf{x})\mathbf{s}_i$ are approximated by finite differences, this technique is usually known as *sparse finite differencing*. In a similar manner, but without the truncation errors associated with finite differences, we may use forward-mode AD to calculate the $\nabla\mathbf{f}(\mathbf{x})\mathbf{s}_i$ and recover the Jacobian with time complexity $O(q) \times W(\mathbf{f})$. If the sparse Jacobian has one or more full or nearly full rows, the value of q will be n or near n and the technique will fail to achieve the desired reduction in run time. In such cases, [Coleman and Verma 1996; 1998b] showed that it may be possible to find a set $\mathbf{s}_j^T$, $j = 1, \ldots, q, q \ll m$, of m-vectors that allow $\nabla\mathbf{f}(\mathbf{x})$ to be recovered from the set of vector-Jacobian products $\mathbf{s}_j^T\nabla\mathbf{f}(\mathbf{x})$ calculated by reverse mode AD. The time complexity is now $O(q) \times W(\mathbf{f})$. Note that this approach is impossible to apply via finite differencing. Coleman and Verma showed that if neither technique succeeds in producing a good compression, it may be possible to combine the forward and reverse approaches to calculate all nonzero entries efficiently.

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

$$c_{ij} = c_{ij} - c_{ik}c_{kj} \tag{11}$$

for which c_{ik} and c_{kj} are both entries. No code is generated where is it known *a priori* that either c_{ik} or c_{kj} is always zero.

The techniques that we have 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. 1996; Bischof et al. 1998], use of sparse options in AD01 [Pryce and Reid 1998], and exploitation of the sparse matrix class of Matlab [Coleman and Verma 1998a; Forth 2001]. The formal operations count for such an approach are low, but the overhead of manipulating the sparse storage typically makes them uncompetitive [Griewank 2000, p.156].

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

$$W(\nabla\mathbf{f}) = W(\mathbf{f}) + W(\mathbf{C}) + W(\text{linear solve}), \tag{12}$$

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_{ij}.

— W(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(linear solve) since we can have little influence on the other two costs.

[Griewank 2000, Chapter 3] takes account of the time required for data transfers from and to the floating-point registers under rather conservative conditions. However, the non-sequential way our optimized derivative code accesses memory and 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, Chapter 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

$$W(\text{linear solve (forward mode)}) \leq n(2N_{\mathbf{c}} - p - m). \tag{13}$$

For each entry, we incur n multiplications when multiplying a row by a $c_{i,j}$ and n additions 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 allow the Jacobian $\nabla\mathbf{f}(\mathbf{x})$ to be recovered from q Jacobian-vector products $\nabla\mathbf{f}(\mathbf{x})\mathbf{s}_i$, $i = 1, \ldots, q$. Clearly, the cost of calculating this is given by (13), but with n replaced by q.

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

$$W(\text{linear solve (reverse mode)}) \leq m(2N_{\mathbf{c}} - p - n). \tag{14}$$

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 (14) with m replaced by q.

2.5 Code-Lists and Statement-Level Differentiation

So far in this paper, 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 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:*

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

in which we note that 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 (q=0), the second line of equation (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 take the view that the extended Jacobian description is more accessible to the scientific computing community.

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. We then discard the pivot row and the pivot column, since they are no longer relevant. 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

$$\left[\begin{array}{ccc|ccc|c} -1 & & & & & & \\ & -1 & & & & & \\ & & -1 & & & & \\ \hline c_{4,1} & c_{4,2} & & -1 & & & \\ & c_{5,2} & c_{5,3} & & -1 & & \\ & c_{6,2} & c_{6,3} & c_{6,4} & & -1 & \\ \hline & c_{7,2} & & c_{7,4} & c_{7,5} & & -1 \\ & & & & c_{8,5} & c_{8,6} & \end{array}\right] \left[\begin{array}{c} \nabla u_1 \\ \nabla u_2 \\ \nabla u_3 \\ \hline \nabla u_4 \\ \nabla u_5 \\ \nabla u_6 \\ \hline \nabla u_7 \\ \nabla u_8 \end{array}\right] = \left[\begin{array}{ccc} -1 & & \\ & -1 & \\ & & -1 \\ \hline & & \\ & & \\ \hline & & \end{array}\right]. \tag{16}$$

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, $c_{6,2} = c_{6,2} + c_{6,4} * c_{4,2}$, $c_{7,2} = c_{7,2} + c_{7,4} * c_{4,2}$. Row and column 4 are now discarded to give

$$\left[\begin{array}{ccc|ccc|c} -1 & & & & & & \\ & -1 & & & & & \\ & & -1 & & & & \\ \hline & c_{5,2} & c_{5,3} & -1 & & & \\ c_{6,1} & c_{6,2} & c_{6,3} & & -1 & & \\ \hline c_{7,1} & c_{7,2} & & c_{7,5} & & -1 & \\ & & & c_{8,5} & c_{8,6} & & -1 \end{array}\right] \left[\begin{array}{c} \nabla u_1 \\ \nabla u_2 \\ \nabla u_3 \\ \hline \nabla u_5 \\ \nabla u_6 \\ \hline \nabla u_7 \\ \nabla u_8 \end{array}\right] = \left[\begin{array}{ccc} -1 & & \\ & -1 & \\ & & -1 \\ \hline & & \\ & & \\ \hline & & \end{array}\right]. \tag{17}$$

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

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

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 the Jacobian has been calculated with a total cost of 16 floating-point operations (flops). This is a substantial saving over the forward mode of Example 2.4 (57 flops) and reverse mode of Example 2.5 (34 flops).

There is a very close relationship between this forward elimination algorithm and the solution of 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, p22],

$$\mathbf{C}-\mathbf{I}=\begin{bmatrix}-\mathbf{I}_n & \mathbf{0} & \mathbf{0}\\ \mathbf{B} & \mathbf{L}-\mathbf{I}_p & \mathbf{0}\\ \mathbf{R} & \mathbf{T} & -\mathbf{I}_m\end{bmatrix}, \tag{18}$$

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

$$\begin{bmatrix}-\mathbf{I}_n & \mathbf{0} & \mathbf{0}\\ \mathbf{B} & \mathbf{L}-\mathbf{I}_p & \mathbf{0}\\ \mathbf{R} & \mathbf{T} & -\mathbf{I}_m\end{bmatrix}\begin{bmatrix}\nabla\mathbf{x}\\ \nabla\mathbf{w}\\ \nabla\mathbf{y}\end{bmatrix}=\begin{bmatrix}-\mathbf{I}_n\\ \mathbf{0}\\ \mathbf{0}\end{bmatrix}. \tag{19}$$

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 $\mathbf{L}$ block and all entries in the $\mathbf{T}$ block from (19),

$$\begin{bmatrix}-\mathbf{I}_n & \mathbf{0} & \mathbf{0}\\ -(\mathbf{L}-\mathbf{I}_p)^{-1}\mathbf{B} & -\mathbf{I}_p & \mathbf{0}\\ \mathbf{R}-\mathbf{T}(\mathbf{L}-\mathbf{I}_p)^{-1}\mathbf{B} & \mathbf{0} & -\mathbf{I}_m\end{bmatrix}\begin{bmatrix}\nabla\mathbf{x}\\ \nabla\mathbf{w}\\ \nabla\mathbf{y}\end{bmatrix}=\begin{bmatrix}-\mathbf{I}_n\\ \mathbf{0}\\ \mathbf{0}\end{bmatrix}, \tag{20}$$

to leave the Jacobian in the lower-left block. We see the two approaches are algebraically equivalent but 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.

An alternative is to pivot on the diagonal entries of the extended Jacobian, but now in reverse order. This confines fill-in to the $\mathbf{R}$ and $\mathbf{T}$ blocks. It is equivalent to solving the transposed system (10) by back-substitution. For our example, this technique also requires 16 flops to calculate the Jacobian.

If full advantage of sparsity is taken, then [Griewank and Reese 1991] showed that these forward and reverse 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.

3.2 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 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 $\mathbf{B}$ and $\mathbf{R}$ or $\mathbf{R}$ and $\mathbf{T}$. Another pivot ordering is likely to create fill-ins in block $\mathbf{L}$. This block is usually far larger than $\mathbf{B}$, $\mathbf{R}$ or $\mathbf{T}$, since there are usually far more intermediate variables than input or output variables. There is therfore 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, Erisman, and Reid 1989]. One of the earliest pivot ordering strategies is due to [Markowitz 1957]. The Markowitz strategy involves, at each elimination step, choosing the 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 number of multiplications required to eliminate that pivot entry (noting that pivots are all -1) and the associated fill-in. Although the Markowitz

strategy is often very successful in reducing the operations count, there are simple examples [Griewank 2000, p179] for which it is found to be sub-optimal. 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, p179]. 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 either the first pivot with minimum cost or the last [Tadjouddine et al. 2001]. In Section 6 and Section 7 we only present results obtained from one of these tie-breaking strategies, that with the lesser number of floating-point operations.

3.3 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 to differentiating each statement is to perform a symbolic differentiation. This approach is adopted by TAMC [Giering 1997]. 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 natural to differentiate the local code list using reverse mode AD. This strategy is used by ADIFOR [Bischof et al. 1998] within an overall forward mode AD approach.

3.4 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. 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.

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]. It uses source-transformation to convert Fortran code

for a function **f** into Fortran code for **f** and its Jacobian. ELIAD 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, e.g. $c_{7,1}$ 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, e.g. 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. This is handled by combining the `true` and `false` branches of an `IF` construct into a single block whose input variables are all possible input variables for the block and whose output variables are all possible output variables for the block. No such test problems are considered in this paper, though preliminary results for one test problem may be found in [Forth and Tadjouddine 2002]. 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** 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 and Section 7, 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 x_6, x_5 and x_4 in that order – from the five assignments of Example 2.1 could be built up as follows:*

```
! Calculate the values of the variables and local derivatives
c_4_1 = 1/x_1;   c_4_2 = 1/x_2
x_4 = log(x_1*x_2)
c_5_2 = x_3**2;   c_5_3 = 2*x_2*x_3
x_5 = x_2*x_3**2-a
c_6_2=1/x_3;  c_6_3=-x_2/x_3**2;  c_6_4=b
x_6 = b*x_4+x_2/x_3
c_7_2=-1;   c_7_4=2*x_4;   c_7_5=1
x_7 = x_4**2+x_5-x_2
c_8_5=-1;  c_8_6=1/(2*sqrt(x_6))
x_8 = sqrt(x_6)-x_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 6.3.

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. This must be done for several test problems on several processor/compiler *platforms*.

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 optimising 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 7.3, Section 7.4) have large, 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, p77], i.e. 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 ($N_{repeats}$) of times. Values of N_{evals} and $N_{repeats}$ used for each platform and each of the test problems of Section 6 and Section 7 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.

We now present in detail the first of our test cases; the remainder are discussed in Section 7.

6. THE HHD TEST-PROBLEM AND ALGORITHM ENHANCEMENTS

In this section, we describe the Human Heart Dipole (HHD) problem from the MINPACK-2 optimization test suite [Averick et al. 1992], and the application of vertex elimination with forward and reverse orderings to both the statement-level and code-list source codes (Section 2.5). We introduce statement re-orderings that aim to improve the effectiveness of optimising compilers by keeping statements using the same variables close together in the Jacobian code. We present comparative timings for all the strategies that we have discussed, but defer discussion of these timings until Section 8 in order to take account of our other test problems.

6.1 Human Heart Dipole Problem

The HHD problem has $n = 8$ independent variables, $m = 8$ dependent variables and corresponds to the formulation

$$
\begin{aligned}
y_1(\mathbf{x}) &= x_1 + x_2 - \sigma_{mx}, \\
y_2(\mathbf{x}) &= x_3 + x_4 - \sigma_{my}, \\
y_3(\mathbf{x}) &= x_5 x_1 + x_6 x_2 - x_7 x_3 - x_8 x_4 - \sigma_A, \\
y_4(\mathbf{x}) &= x_7 x_1 + x_8 x_2 + x_5 x_3 + x_6 x_4 - \sigma_B, \\
y_5(\mathbf{x}) &= x_1(x_5^2 - x_7^2) - 2x_3 x_5 x_7 + x_2(x_6^2 - x_8^2) - 2x_4 x_6 x_8 - \sigma_C, \\
y_6(\mathbf{x}) &= x_3(x_5^2 - x_7^2) + 2x_1 x_5 x_7 + x_4(x_6^2 - x_8^2) + 2x_2 x_6 x_8 - \sigma_D, \\
y_7(\mathbf{x}) &= x_1 x_5(x_5^2 - 3x_7^2) + x_3 x_7(x_7^2 - 3x_5^2) + x_2 x_6(x_6^2 - 3x_8^2) + x_4 x_8(x_8^2 - 3x_6^2) - \sigma_E, \\
y_8(\mathbf{x}) &= x_3 x_5(x_5^2 - 3x_7^2) - x_1 x_7(x_7^2 - 3x_5^2) + x_4 x_6(x_6^2 - 3x_8^2) - x_2 x_8(x_8^2 - 3x_6^2) - \sigma_F,
\end{aligned}
$$

with constants σ_{mx}, σ_{my}, σ_A, σ_B, σ_C, σ_D, σ_E and σ_F supplied. It is coded via p=20 intermediate variables of which 8 are simple copies of the independent variables. The code-list uses $p = 84$ intermediate variables. The first row of data in Table XIII of Appendix B presents data associated with calculating the HHD function itself.

Table I, presents performance data for several techniques applied to calculating the Jacobian $\nabla\mathbf{f}(\mathbf{x})$

Table I. Ratios of Jacobian to function flop counts and CPU times for Human Heart Dipole problem.

Technique	Ratios of flops $W(\nabla\mathbf{f})/W(\mathbf{f})$	Ratios of CPU times ALPHA	SGI	Ultra10	PIII	AMD
Hand-coded	2.00	2.97	3.88	4.79	2.22	2.64
ADIFOR	13.88	15.30	21.26	18.61	14.59	13.80
TAMC-F	18.31	16.93	24.27	24.85	16.62	14.90
TAMC-R	20.79	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	3.05	4.28	3.75	3.32	3.82
VE-SL-R	3.00	3.26	3.92	3.83	3.35	3.79
VE-CL-F	5.00	2.92	4.50	4.01	3.83	4.26
VE-CL-R	3.95	2.79	4.01	3.52	3.74	4.33
VE-SLP-F	3.05	2.75	3.83	3.50	**3.21**	3.73
VE-SLP-R	3.05	**2.61**	3.76	3.43	3.29	3.70
VE-CLP-F	3.90	2.79	3.90	3.80	3.61	4.24
VE-CLP-R	3.90	2.79	3.74	**3.19**	3.63	4.09
VE-SLP-F-DFT	3.05	2.99	3.80	3.54	3.22	3.52
VE-SLP-R-DFT	3.05	2.62	**3.71**	3.88	3.28	3.53
VE-CLP-F-DFT	3.90	3.19	3.88	3.86	3.56	3.80
VE-CLP-R-DFT	3.90	3.25	3.93	**3.19**	3.43	**3.46**
VE-SLP-Mark[1]	3.00	2.70	3.85	3.36	3.26	3.69
VE-SLP-VLR[1]	3.00	2.74	3.83	3.36	3.30	3.73
VE-CLP-Mark[2]	3.86	2.82	3.91	4.02	3.83	4.21
VE-CLP-VLR[2]	3.86	2.80	3.90	4.03	3.77	4.19
VE-SLP-Mark-DFT[3]	3.00	2.99	3.81	3.53	3.29	3.54
VE-SLP-VLR-DFT[3]	3.00	2.99	3.82	3.55	3.31	3.51
VE-CLP-Mark-DFT[4]	3.86	3.17	3.88	3.86	3.43	3.77
VE-CLP-VLR-DFT[4]	3.86	3.19	3.86	3.86	3.42	3.73

Notes (see also Section 6.4):
[1] These codes differ only in statement order.
[2] These codes differ only in statement order.
[3] These codes identical and differ from [1] only in statement order.
[4] These codes identical and differ from [2] only in statement order.

of the HHD function. 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 our PERL script) and the corresponding ratios of CPU times. 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.

In Table I, the first row of data corresponds to the hand-coded Jacobian code supplied by [Averick et al. 1992], but with all branching removed. Getting a smaller ratio than this with an AD technique is the aim of our work.

The next four rows 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`.

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 with statement-level (SL) differentiation and the second pair by differentiating the code-list (CL). In each of these two pairs, the first uses the forward (F) elimination ordering and the second the reverse (R) ordering.

Note that 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, p32] to avoid such unnecessary arithmetic operations.

6.2 Reverse Pre-Elimination

In Table I, rows VE-SLP-F and VE-SLP-R correspond to a reverse pre-elimination (Section 3.4) followed by forward- and reverse-ordered eliminations of all remaining vertices. Rows VE-CLP-F and VE-CLP-R are the code-list differentiated equivalents.

6.3 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 perturbed the order of statements without altering the data dependencies within the code. We reordered the assignment statements with the aim of using each assigned value soon after its assignment. This was done by a modified version of the depth-first traversal 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 depth-first traversal (DFT) strategy to the codes of rows VE-SLP-F to VE-CLP-R.

6.4 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 3.2 to order the vertex elimination, after the reverse

pre-elimination of Section 3.4. Note that the identical flops ratio for the Markowitz and VLR orderings when applied to either the statement-level (VE-SLP-Mark and VE-SLP-VLR) or code-list (VE-CLP-Mark and VE-CLP-VLR) differentiation is more than coincidence. As noted in Table I, the respective elimination sequences and hence the ELIAD generated Jacobian codes do differ, but are found to be identical after DFT reordering. The small reduction in the number of floating-point operations compared to pre-accumulated forward orderings is insufficient to give a performance improvement.

We see that comparison of the data of row VE-SLP-Mark-DFT with VE-SLP-VLR-DFT and row VE-CLP-Mark-DFT with VE-CLP-VLR-DFT indicates that the run time ratios of Table I are consistent to within 0.04 for the AMD and within 0.02 for the others. We might therefore rank the two underlined

entries on the ALPHA and PIII platforms as also having equivalent best timings to within measurement error on those platforms.

6.5 Summary for HHD Test Case

For this test case, vertex elimination Jacobian code generated by ELIAD requires between 0.67 (Ultra10) to 1.45 (PIII) times the CPU time of hand-coded Jacobian code, is 3.4 (SGI,PIII) to 4.7 (ALPHA) times faster than 1-sided finite differencing, and 4.0 (AMD) to 5.9 (ALPHA) times faster than the best conventional AD generated code. In particular, we note that on the three UNIX platforms (ALPHA, SGI, Ultra10) the ELIAD generated code is more efficient than the hand-coded Jacobian code. Encouraged by these results, we now consider four further test cases.

7. FURTHER TEST CASES

In Section 7.1, we consider another full Jacobian case from [Averick et al. 1992]. In Section 7.2, we consider the Roe flux CFD problem [Roe 1981] which also has a full Jacobian. We have considered this problem previously [Tadjouddine et al. 2002], though not in conjunction with the pre-elimination technique. We then move on to two sparse test problems. Section 7.3 concerns the Coating Thickness Standardisation problem [Averick et al. 1992] which, as coded and differentiated at statement level has no intermediate variables. Both this, and the Flow in Channel case of Section 7.4 [Averick et al. 1992] are amenable to Jacobian compression techniques (Section 2.3) and so their Jacobians may be efficiently calculated by present AD techniques.

7.1 Combustion of Propane (Full Formulation)

The Combustion of Propane (CPF) problem [Averick et al. 1992] has $n = 11$ independent variables, $m = 11$ dependent variables and corresponds to the formulation,

$$\begin{aligned}
y_1(\mathbf{x}) &= x_1 + x_4 - 3 \\
y_2(\mathbf{x}) &= 2x_1 + x_2 + x_4 + x_7 + x_8 + x_9 + 2x_{10} - R \\
y_3(\mathbf{x}) &= 2x_2 + 2x_5 + x_6 + x_7 - 8 \\
y_4(\mathbf{x}) &= 2x_3 + x_9 - 4R \\
y_5(\mathbf{x}) &= K_5 x_2 x_4 - x_1 x_5 \\
y_6(\mathbf{x}) &= K_6 x_2^{1/2} x_4^{1/2} - x_1^{1/2} x_6 \left(P/x_{11}\right)^{1/2} \\
y_7(\mathbf{x}) &= K_7 x_1^{1/2} x_2^{1/2} - x_4^{1/2} x_7 \left(P/x_{11}\right)^{1/2} \\
y_8(\mathbf{x}) &= K_8 x_1 - x_4 x_8 \left(P/x_{11}\right) \\
y_9(\mathbf{x}) &= K_9 x_1 x_3^{1/2} - x_4 x_9 \left(P/x_{11}\right)^{1/2} \\
y_{10}(\mathbf{x}) &= K_{10} x_1^2 - x_4^2 x_{10} \left(P/x_{11}\right)^{1/2} \\
y_{11}(\mathbf{x}) &= x_{11} - \sum_{j=1}^{10} x_j
\end{aligned}$$

with data $P, R, K_5, \ldots, K_{10}$ supplied. The function is coded with $p = 13$ intermediate variables and the corresponding code-list has $p = 57$ intermediates. The nominal flop count $W(\mathbf{f})$ and (unrolled) function run times on our 5 platforms are given in Table XIII of Appendix B.

Combustion of Propane (Full Table II gives ratios for the CPF problem in exactly the same way as Table I does for the HHD problem in Section 6. For statement-level differentiation of this problem, we find that the forward and Markowitz orderings are identical, and consequently so is their Jacobian code. Further, the VLR ordering is equivalent and so all three are identical after DFT statement reordering. For the code-list vertex eliminations, the forward, Markowitz and VLR orderings and hence their Jacobian codes are identical and, though changed, remain identical after DFT statement reordering.

By comparing the timing data of identical cases in Table II we find consistency between all such cases on the ALPHA platform, to within 0.02 on the SGI, Ultra10 and PIII platforms and to within 0.04 on the

Table II. Ratio of Jacobian to function CPU times for Combustion of Propane (full formulation)

Technique	Ratios of flops $W(\nabla\mathbf{f})/W(\mathbf{f})$	Ratios of CPU times ALPHA	SGI	Ultra10	PIII	AMD
Hand-coded	2.24	1.97	2.42	3.66	2.23	2.86
ADIFOR	14.44	5.89	6.63	10.69	5.76	8.81
TAMC-F	24.76	6.60	7.41	11.52	6.95	12.92
TAMC-R	27.03	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	1.77	1.91	2.53	2.15	3.12
VE-SL-R	2.41	1.82	1.94	2.60	2.23	3.14
VE-CL-F	3.81	1.77	1.91	2.55	2.23	3.24
VE-CL-R	2.78	1.68	1.89	2.35	2.18	3.11
VE-SLP-F[1]	2.37	1.74	**1.49**	1.93	2.02	2.94
VE-SLP-R	2.40	1.66	1.64	1.84	2.04	2.93
VE-CLP-F[4]	2.75	1.62	1.50	1.87	1.97	2.86
VE-CLP-R	2.78	1.64	1.60	1.85	1.98	2.86
VE-SLP-F-DFT[2]	2.37	1.37	1.53	1.85	1.85	**2.27**
VE-SLP-R-DFT	2.40	1.34	1.52	1.95	1.88	**2.27**
VE-CLP-F-DFT[5]	2.75	**1.32**	1.72	1.89	1.84	2.32
VE-CLP-R-DFT	2.78	1.35	1.54	**1.81**	1.84	2.31
VE-SLP-Mark[1]	2.37	1.74	**1.49**	1.92	2.04	2.93
VE-SLP-VLR[3]	2.37	1.74	**1.49**	1.92	2.03	2.94
VE-CLP-Mark[4]	2.75	1.62	1.51	1.86	1.97	2.89
VE-CLP-VLR[4]	2.75	1.62	1.51	1.86	1.97	2.93
VE-SLP-Mark-DFT[2]	2.37	1.37	1.52	1.85	1.84	**2.27**
VE-SLP-VLR-DFT[2]	2.37	1.37	1.53	1.85	1.84	**2.27**
VE-CLP-Mark-DFT[5]	2.75	**1.32**	1.72	1.89	**1.83**	2.33
VE-CLP-VLR-DFT[5]	2.75	**1.32**	1.73	2.00	1.84	2.33

Notes:
[1] These codes identical and differ from [2] and [3] only in statement order.
[2] These codes identical and differ from [1] and [3] only in statement order.
[3] These code differs from [1] and [2] only in statement order.
[4] These codes identical and differ from [5] only in statement order.
[5] These codes identical and differ from [4] only in statement order.

AMD platform. The underlined entries in Table II could therefore be ranked as having equivalent best timings on their platform.

7.2 Roe Flux

This test case corresponds to the evaluation of fluxes of mass, energy and 3 components of momentum between two cells of a CFD finite-volume flow solver using Roe's flux difference splitting scheme [Roe 1981]. For this problem, there are $n = 10$ independent variables in two vectors $\mathbf{q}_l = [p_l, \rho_l, u_l, v_l, w_l]^T$ and $\mathbf{q}_r = [p_r, \rho_r, u_r, v_r, w_r]^T$ corresponding to the pressure, density and 3 components of velocity to the left and right of a mesh cell interface. A numerical flux $\mathbf{y} = \mathbf{f}(\mathbf{q}_l, \mathbf{q}_r), \mathbf{y} \in I\!R^5$ is calculated in an involved manner involving a suitable averaging of the $\mathbf{q}_{l,r}$ to interface values $\hat{\mathbf{q}}$, finding eigenvalue and eigenvectors associated with a matrix constructed from $\hat{\mathbf{q}}$ and using the eigenvalues and eigenvectors to *upwind* the flux function $\mathbf{y} = \mathbf{f}(\mathbf{q}_l, \mathbf{q}_r)$. For further details see [Roe 1981]. For our purposes, we note that the associated Fortran subroutine uses $p = 62$ intermediate variables to calculate the $\mathbf{y}$, the corresponding code-list uses $p = 233$ intermediates, and the required 5×10 Jacobian is dense.

Function nominal flops $W(\mathbf{f})$ and measured CPU times may be found in Table XIII of Appendix B. Table III gives the nominal flop and CPU ratios as for our previous examples, except that we do not possess hand-coded Jacobians in this case. Note that, unlike in the previous two case, all the cross-country (Markowitz and VLR) orderings are different.

Table III. Ratios of Jacobian to function flop counts and CPU times for Roe Flux

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

7.3 Coating Thickness Standardization

The Coating Thickness Standardization (CTS) problem [Averick et al. 1992] corresponds to the least-squares data-fitting of two bilinear functions,

$$\begin{aligned} Z_1(\zeta,\eta) &= x_1 + x_2\zeta + x_3\eta + x_4\zeta\eta, \\ Z_2(\zeta,\eta) &= x_5 + x_6\zeta + x_7\eta + x_8\zeta\eta, \end{aligned}$$

with supplied coordinates $(\zeta_i,\eta_i), i = 1,\ldots,n_0$, to supplied data $z_i, i = 1,\ldots,n_0$ and $z_i, i = n_0+1,\ldots,2n_0$ respectively. It is assumed that the coordinates ζ_i,η_i are subject to errors x_{8+i}, x_{8+i+n_0} to give data-fit residuals coded as single statements,

$$\left.\begin{aligned} y_i(\mathbf{x}) &= x_1 + x_2(\zeta_i + x_{8+i}) + x_3(\eta_i + x_{8+i+n_0}) \\ &\quad + x_4(\zeta_i + x_{8+i})(\eta_i + x_{8+i+n_0}) - z_i \\ y_{i+n_0}(\mathbf{x}) &= x_5 + x_6(\zeta_i + x_{8+i}) + x_7(\eta_i + x_{8+i+n_0}) \\ &\quad + x_8(\zeta_i + x_{8+i})(\eta_i + x_{8+i+n_0}) - z_{i+n_0} \end{aligned}\right\}, \tag{22}$$

for $i = 1,\ldots,n_0$. Additional residuals,

$$y_{i+2n_0}(\mathbf{x}) = w_i x_{8+i}, i = 1,\ldots,2n_0, \tag{23}$$

for given weights w_i, are also calculated. We see that rows 1 to $2n_0$ of the resulting Jacobian have 6 nonzero entries and rows $2n_0+1$ to $4n_0$ have just one nonzero. Also, since rows 1 to n_0 depend on $x_1,\ldots,x_4$, and rows n_0+1 to $2n_0$ depend on $x_5,\ldots,x_8$, we see that there are columns with n_0 entries. For the data supplied, $n_0 = 63$ so that $n = 134$, $m = 252$ and there are columns with 63 entries. Coding of (22),(23) ensures that for a statement-level differentiation there are no intermediate variables, that is, $p = 0$. In contrast, the code-list uses $p = 1386$ intermediates.

Table XIII of Appendix B gives the nominal number of flops needed to calculate the function $\mathbf{f}$ for this test problem and the associated CPU time on each platform studied. Table IV gives the nominal flop and CPU ratios as for our previous examples, except that compression is used for the conventional

Table IV. Ratios of Jacobian to function flop counts and CPU times for Coating Thickness Standardization

Technique	Ratios of flops $W(\nabla\mathbf{f})/W(\mathbf{f})$	Ratios of CPU times ALPHA	SGI	Ultra10	PIII	AMD
Hand-coded	1.85	4.62	3.79	9.63	3.89	3.18
ADIFOR(cmp)	6.38	37.42	48.84	75.80	25.40	30.23
TAMC-F(cmp)	11.15	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	**4.07**	4.12	**7.28**	4.30	3.09
VE-CL-F	3.23	4.08	4.07	9.48	5.47	4.08
VE-CL-R	2.54	4.16	**4.06**	9.66	5.20	3.78
VE-SL-DFT	1.85	4.85	4.86	9.96	**3.78**	**2.60**
VE-CL-F-DFT	3.23	4.75	4.83	8.48	4.36	3.11
VE-CL-R-DFT	2.54	4.74	4.82	9.31	4.36	2.87

methods: ADIFOR(cmp),TAMC-F(cmp) and FD(cmp). Since the Jacobian is sparse, with a maximum of 6 nonzeros per row, we use the DSM software [Coleman et al. 1984] to obtain a column compression for the supplied sparsity pattern. In this case, we required 6 Jacobian-vector products from which the Jacobian may be reconstructed. Note that a row compression would require at least 63 vector-Jacobian products and so would be much less efficient; we therefore did not run this version. As before, the best ELIAD generated entry for each column is shown in bold and underline the entries that are very nearly as good. 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, then the partial derivatives of each statement correspond to entries in the Jacobian. With reference to (18), the sub-blocks $\mathbf{B}$, $\mathbf{L}$ and $\mathbf{T}$ of the extended Jacobian are empty and sub-block $\mathbf{R}$ 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. Indeed for this example, the operations count indicates equivalence of the statement-level vertex elimination with the hand-coded Jacobian. This is not true for conventional AD. For example, in forward mode AD the coefficients $\mathbf{R}$ multiply the length-n (or length q for compression) vectors of the $\nabla x_i, i = 1, \ldots, n$.

Lines VE-CL-F and VE-CL-R correspond to differentiation of the code list followed by forward and reverse eliminations. We would expect the nominal cost $W(\nabla\mathbf{f})$ for the reverse ordering to be close to that of the statement-level differentiation. The appreciable difference once again is due to redundant multiplications by one in the code-list differentiated code. The reverse pre-elimination of Section 3.4 is not necessary on this problem. This is because it will eliminate derivatives of all variables corresponding to intermediate results in the statements starting with the last statement. Since the only other variables' derivatives remaining are those of the dependent variables we see that the reverse pre-elimination will produce the same elimination sequence as the reverse ordering.

No cross-country elimination sequences were used for this problem since reverse pre-elimination alone allows for Jacobian evaluation.

7.4 Flow in Channel

The Flow in Channel (FIC) test case [Averick et al. 1992] corresponds to solution of the 4^{th} order nonlinear boundary value problem,

$$u'''' = R\,[u'u'' - uu'''], \quad 0 \le t \le 1, \quad u(0) = u'(0) = 0,\ u(1) = 1,\ u'(1) = 0,$$

via a 4-stage collocation method on a mesh with 4 internal mesh points. The Reynolds number R is taken as 10. Discretization of the collocation equations results in a nonlinear system of 32 equations, $y_i = f_i(\mathbf{x}), i = 1, \ldots, 32$ in 32 unknowns $\mathbf{x}$. The supplied subroutine uses $p = 680$ intermediate variables

and its code-list $p = 1328$. The resulting $n \times m = 32 \times 32$ Jacobian is sparse with a maximum of 9 nonzeros per row and 9 nonzeros per column. An interesting feature of this subroutine is that all its intermediate variables are used in only one other statement.

Table XIII of Appendix B gives the nominal number of flops needed to calculate the function $\mathbf{f}$ for this test problem and the associated CPU time on each platform studied. Table V gives the nominal flop and

Table V. Ratios of Jacobian to function flop counts and CPU times for Flow in Channel

Technique	Ratios of flops $W(\nabla\mathbf{f})/W(\mathbf{f})$	Ratios of CPU times ALPHA	SGI	Ultra10	PIII	AMD
Hand-coded	1.91	2.44	2.72	8.33	3.34	2.58
ADIFOR(cmp)	10.44	15.50	57.93	38.48	47.68	59.52
TAMC-F(cmp)	10.85	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	2.21	3.08	3.82	**4.18**	2.78
VE-SL-R	2.25	2.14	3.10	4.05	5.12	3.66
VE-CL-F	4.44	2.33	3.26	4.94	4.87	3.35
VE-CL-R	2.75	2.12	3.09	**3.41**	5.17	3.33
VE-SL-R-DFT	2.25	**2.10**	**2.97**	4.89	4.72	2.67
VE-CL-R-DFT	2.75	2.12	3.02	6.80	4.91	**2.56**

CPU ratios in a similar manner as for our previous examples. Again the most efficient vertex elimination case in each row is shown in bold and any underlined entries are as fast to within measurement error. As in Section 7.3, we use the DSM software to enable row compression of the Jacobian calculation using 9 Jacobian-vector products. We could have used a column compression via 9 vector-Jacobian products calculated using TAMC in adjoint mode. We did not since, given possible recomputation and more involved control flow for the reverse mode, we would expect resulting run times to be slightly worse than the row compression. Since, as noted above, each intermediate variable in the subroutine is used just once then reverse pre-elimination alone would eliminate all intermediate variables. Pre-elimination is thus equivalent to the reverse orderings VE-SL-R and VE-CL-R and so 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 pre-elimination alone would produce the Jacobian.

8. DISCUSSION OF THE RESULTS

We now discuss the results shown in Tables I to V.

8.1 Conventional AD Techniques: ADIFOR and TAMC

In Tables I to III, we note both a lower nominal flops ratio $W(\nabla\mathbf{f})/W(\mathbf{f})$ and CPU time ratio for ADIFOR compared to TAMC-F despite both using forward mode AD to calculate the Jacobian. This is because ADIFOR uses a reverse-mode AD differentiation of each statement to calculate its partial derivatives whereas TAMC-F uses a statement-level symbolic differentiation of each statement and relies on compiler optimization to remove common sub-expressions. Here, the compiler optimizations are not performing as well as the statement-level reverse differentiation of ADIFOR. In our two cases where the Jacobian is sparse and compression is used (Tables IV and V), the performance of ADIFOR and TAMC is very similar on all platforms despite a near double nominal flop count in Table IV for TAMC. Note that all the statements in the source code for the FIC problem of Table V are multiply-adds which both ADIFOR and TAMC differentiate for no cost in FLOPs, and for the CTS problem the most complicated statements (22) are sufficiently involved that TAMC's symbolic differentiation involves more nominal floating point operations than ADIFOR's statement-level reverse strategy and yet the statements are sufficiently simple that the optimizing compiler can generate equally efficient assembler code.

In Tables I and II, we see that reverse mode AD, specifically TAMC-R, uses both more nominal flops and is less efficient than forward mode. Since the same statement-level differentiation is performed in both TAMC modes and $n = m$, we expected forward and reverse modes to have similar costs (see (13) and (14))

and hence similar runtime ratios $W(\nabla \mathbf{f})/(W(\mathbf{f})$. We speculate that the extra time is caused by the reverse method needing to store more data.

Across all problems and platforms, the best of the conventional AD techniques were between 2.6 (Table II, PIII) to 22.5 (Table V, AMD) times more expensive than the hand coding.

8.2 One-Sided Finite Differencing

The 1-sided finite-difference approximation was between 2.2 (Table V, Ultra10) to 9.1 (Table V, PIII) times more expensive than the hand-coded Jacobians, where available. Of course, it was also far less accurate. Note that, since inlining of subroutine calls has been prevented in our study (c.f. Section. 5), the efficiency of finite differencing could be further improved.

8.3 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. This is confirmed in the run-time ratios. The best of these four vertex elimination methods gave a speed-up over the best of the conventional AD techniques of 1.5 (Table III, Ultra10) to 8.9 (Table IV, AMD) times.

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.

8.4 Code-List versus Statement-Level Differentiation

In Tables I-V we see that the statement-level differentiated vertex-elimination codes always use fewer nominal flops and, in 34 of the 50 problem/platform combinations, out-perform the corresponding code-list differentiated codes.

8.5 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 3.4) which can dramatically reduce the number of operations for forward and reverse vertex elimination applied to the code-list. This is seen in the reduction in nominal flops when moving from VE-CL-F/VE-CL-R to VE-CLP-F/VE-CLP-R in Tables I to III. The reduction is particularly noteworthy for the forward orderings (VE-CL-R, VE-CLP-F) of these tables. Pre-elimination may also affect 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 III.

In terms of run-time, the application of pre-elimination usually improves efficiency. Sometimes the improvement is very substantial (see Table III, Ultra10 and PIII) and sometimes there is hardly any change despite a worthwhile reduction in the number of operations (see Table III, ALPHA). We believe that 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.

8.6 Depth-First Traversal (DFT) Statement Re-Ordering

The results with depth-first traversal (DFT) statement re-ordering (Section 6.3) were very mixed. Significant gains were sometimes made: ALPHA and AMD of Table II, 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. Our aim of applying DFT reordering was to reduce the number of loads of data from cache to registers and stores of data from registers to cache. Sometimes this approach is successful. For example, consider the FIC test case of Table V and in particular the application of DFT re-ordering to the VE-SL-R subroutine to give VE-SL-R-DFT. On the PIII platform we obtain an associated 8% speed-up and on examining the associated assembler we find that this is most likely to be associated with 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 a 27% increase in the number of stores.

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.

8.7 Cross-Country Vertex Elimination

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

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

Table VI. Size and number of entries in blocks **B**, **L**, **R** and **T** of the extended Jacobians (equation (19)) 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**
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)
Roe	10	5	62(36)	39(39)	140(88)	5(5)	14(37)
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)

by equation (19) before and after pre-elimination. For the two sparse cases CTS and FIC of Sections 7.3 and 7.4, pre-elimination alone obtains the Jacobian in block **R** with other blocks then empty. For the CPF problem of Section 7.1, 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 6.1 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 7.2, 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.

8.8 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

Table VII. Speed-up of best vertex elimination AD over 1-sided finite differencing

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

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 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 more AD for Jacobian calculation.

Table VIII. Speed-up of best vertex elimination over best conventional AD

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

Table IX 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%.

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

9. 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. non-monotonic) 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 2002b]. 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; 2002a] 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 large 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; 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 branching and subprograms. In both cases, a hierarchical approach utilising a conservative activity analysis of all variables in all possible branches and subroutines is adopted. This approach automates and generalises that of [Bischof and Haghighat 1996].

We feel certain that, with the theoretical and implementation issues of Automatic Differentiation for Jacobian calculation now being studied in depth, the days of scientists and engineers painstakingly hand-coding Jacobian code to ensure efficiency are numbered.

APPENDICES

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.

Table X. Platforms

a) Processors

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

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

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 XI. Summary statistics for test problem functions

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

Table XII gives N_{evals}, the number of distinct Jacobians calculated, and $N_{repeats}$ the number of times

Table XII. Values of N_{evals} and $N_{repeats}$ used for each problem and platform

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

these calculations were repeated, in order to obtain reliable Jacobian timings.

The second column of Table XIII gives the nominal computational cost $W(\mathbf{f}(\mathbf{x}))$ of calculating the

Table XIII. Nominal floating-point operations counts $W(\mathbf{f})$ and CPU times for test problem functions

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

test problems described in Section 6 and Section 7. It is determined by counting the number of floating operations within the Fortran code using a simple PERL script. Table XIII also gives 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.

Acknowledgements

We thank Neil Stringfellow and Venkat Sastry for their help in developing the PERL scripts used in this paper.

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. Technical Report 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, Penn.

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. and Haghighat, M. R. 1996. Hierarchical approaches to automatic differentiation. See Berz et al. [1996], 83–94.

Bischof, C. H., Khademi, P. M., Bouaricha, A., and Carle, A. 1996. Efficient computations of gradients and Jacobians by dynamic exploitation of sparsity in automatic differentiation. *Optimization Methods and Software 7*, 1–39.

Bischof, C. H., Roh, L., and Mauer, A. 1997. ADIC — An extensible automatic differentiation tool for ANSI-C. *Software–Practice and Experience 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. Software 10,* 3, 329–345.

Coleman, T. F. and Verma, A. 1996. Structure and efficient Jacobian calculation. See Berz et al. [1996], 149–159.

Coleman, T. F. and Verma, A. 1998a. ADMAT: An automatic differentiation toolbox for MATLAB. Tech. rep., Computer Science Department, Cornell University.

Coleman, T. F. and Verma, A. 1998b. The efficient computation of sparse Jacobian matrices using automatic differentiation. *SIAM J. Sci. Comput. 19,* 4, 1210–1233.

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. S., Erisman, A. M., and Reid, J. K. *Direct methods for sparse matrices.* 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. 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. and Tadjouddine, M. 2002. CFD Nwton solvers with EliAD, an elimination automatic differentiation tool. In *Second International Conference on Computational Fluid Dynamics.*

Giering, R. 1997. *Tangent Linear and Adjoint Model Compiler, Users Manual.* Center for Global Change Sciences, Department of Earth, Atmospheric, and Planetary Science, MIT, Cambridge, MA. Unpublished. Available at `puddle.mit.edu/~ralf/tamc`.

Goedecker, S. and Hoisie, A. 2001. *Performance Optimisation of Numerically Intensive Codes.* Software, Environments, Tools. SIAM, Philidelphia. 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, Penn.

Griewank, A. and Corliss, G. F., Eds. 1991. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application.* SIAM, Philadelphia, Penn.

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. Software 22,* 2, 131–167.

Griewank, A. and Reese, S. 1991. On the calculation of Jacobian matrices by the Markowitz rule. See Griewank and Corliss [1991], 126–135.

Hovland, P. D. and McInnes, L. C. 2001. Parallel simulation of compressible flow using automatic differentiation and PETc. *Parallel Computing 27,* 4 (March), 503–519.

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

Markowitz, H. 1957. The elimination form of the inverse and its application. *Management Science 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. See Corliss et al. [2001], Chapter 29, 241–246.

Naumann, U. 2002a Elimination methods in computational graphs. on the optimal accumulation of jacobian matrices-conceptual framework. Submitted to *Mathematical Programming.*

Naumann, U. 2002b On optimal Jacobian accumulation for single expression use programs. Submitted to *Mathematical Programming.*

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`.

Roe, P. L. 1981. Approximate Riemann solvers, parameter vectors, and difference schemes. *Journal of Computational Physics 43*, 357–372.

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

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

TADJOUDDINE, M., FORTH, S. A., AND PRYCE, J. D. 2001. AD tools and prospects for optimal AD in CFD flux Jacobian calculations. See Corliss et al. [2001], Chapter 30, 247–252.

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 Second International Conference on Computational Science*, P. M. Sloot, Ed. Lecture Notes in Computer Science, vol. 2. Springer-Verlag, Amsterdam, 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, New York, 174–183.