SIAM J. SCI. COMPUT. © 2008 Society for Industrial and Applied Mathematics
Vol. 30, No. 4, pp. 1659–1677

FAST AUTOMATIC DIFFERENTIATION JACOBIANS BY COMPACT LU FACTORIZATION[*]

JOHN D. PRYCE[†] AND EMMANUEL M. TADJOUDDINE[‡]

Abstract. For a vector function coded without branches or loops, a code for the Jacobian is generated by interpreting Griewank and Reese's vertex elimination as Gaussian elimination and implementing this as compact LU factorization. Tests on several platforms show such a code is typically 4 to 20 times faster than that produced by tools such as ADIFOR, TAMC, or TAPENADE, on average significantly faster than vertex elimination code produced by the ELIAD tool [Tadjouddine et al., in Proceedings of ICCS (2), Lecture Notes in Comput. Sci. 2330, Springer, New York, 2002] and can outperform a hand-coded Jacobian. The LU approach is promising, e.g., for CFD flux functions that are central to assembling Jacobians in finite element or finite volume calculations and, in general, for any inner-loop basic block whose Jacobian is crucial to an overall computation involving derivatives.

Key words. automatic differentiation, vertex elimination, source transformation, abstract computational graph, sparse matrix, LU factorization

AMS subject classifications. 65K05, 65Y20, 65F50, 15A24, 90C90

DOI. 10.1137/050644847

1. Introduction.

Automatic differentiation (AD) is now a standard technology for computing derivatives of a (vector) function $\mathbf{f}(\mathbf{x})$ defined by a computer code, such as sensitivities with respect to design parameters, Jacobians for use in Newton and related iterations, Taylor series generation, etc. This article is concerned with first derivatives, i.e., with the Jacobian matrix $J = J(\mathbf{x}) = \mathbf{f}'(\mathbf{x})$ of $\mathbf{f}$.

Especially for large industrial problems, the preferred approach is *source-to-source translation*, where the code for $\mathbf{f}$ is preprocessed to produce Jacobian code, i.e., code for $\mathbf{f}'$. With current compilers, such a code generally runs an order of magnitude faster than the alternative of generating derivatives by operator overloading.

For functions defined by straight-line code, this paper shows that the *vertex elimination* (VE) method of Griewank and Reese [9] is a form of Gaussian elimination and sets it in the framework of sparse matrix factorization as described, e.g., in Duff, Erisman, and Reid [5]. In the experiments presented here we have implemented it as a *compact LU factorization* (usually named after Crout or Doolittle). Standard VE implementations, e.g., [6], produce a code with many short statements, while the LU style produces fewer, longer statements of inner-product form. Each affected entry in the extended Jacobian is changed just once, with a statement $c_{ij} = c_{ij} + \sum_{k\in K} c_{ik}c_{kj}$ if updating an original local derivative or $c_{ij} = \sum_{k\in K} c_{ik}c_{kj}$ if creating fill-in. The structure of a LU-style Jacobian code makes it easy to do certain optimizations locally that standard VE cannot do without a more global code analysis.

In the graph viewpoint, an entry c_{ij} labels the edge from vertex j to vertex i, K is the set of vertices k such that there is a 2-edge path $j \to k \to i$, and each affected edge $j \to i$ is updated/created by collapsing such paths into it. Researchers into vertex

[*]Received by the editors November 10, 2005; accepted for publication (in revised form) December 18, 2007; published electronically April 18, 2008.
http://www.siam.org/journals/sisc/30-4/64484.html

[†]CISE, Department of Informatics and Simulation, Cranfield University (RMCS Shrivenham), Swindon SN1 8LA, United Kingdom (j.d.pryce@cranfield.ac.uk).

[‡]Department of Computing Science, University of Aberdeen, King's College, Aberdeen AB24 3UA, United Kingdom (etadjoud@csd.abdn.ac.uk).

elimination AD have mainly thought in terms of manipulating the computational graph (Griewank [8] and Naumann [11]). They acknowledge the linear algebra but see it as secondary. Our approach is hard to devise without taking a linear algebra view.

As with other sparse factorizations, the pivot order is important and is chosen with the aim of controlling fill-in. Following Naumann's work on VE, e.g., [12], we have used in our tests the classic Markowitz local algorithm, Naumann's vertex lowest relative (VLR) variant of Markowitz, simple forward and reverse ordering, and "preelimination" versions of these where all intermediate vertices in the computational graph (CG) that have a single successor are used first. Preelimination was recommended by Naumann in [12], and in earlier tests of VE with the EliAD tool [16, 17] it gave promising results.

The functions that originally interested us in this work were flux "kernels" for finite element or finite volume calculations in CFD. These are at most a few hundred lines of code, and typically a resulting Jacobian routine will be called hundreds of thousands of times, or more, in a CFD calculation while assembling a large sparse Jacobian from pieces. In such applications, algorithms for choosing the pivot order are a negligible part of the overall cost, and we have not paid any attention to their efficiency; our Markowitz-style methods are slow for large systems. From the sparse matrix mainstream, work such as Amestoy, Li, and Ng [1], and references therein, presents asymptotically faster Markowitz-like algorithms that would be useful if our present work proves relevant to Jacobians of lengthy functions.

Our performance tests show that a code produced by LU methods usually ran faster than other VE-style code produced by the EliAD tool; it was generally comparable with hand coding and much faster than code from other tools such as Tapenade.

The material is structured as follows. Section 2 describes the standard approach to VE, and the linear algebra viewpoint leading to the LU approach. Section 3 outlines our code generation method. Section 4 describes our performance comparisons between Jacobian codes produced by various methods for a number of functions and a number of platforms. Section 5 draws conclusions.

2. Vertex and edge elimination.

2.1. Code lists.

The calculation of $\mathbf{f}$ will be described by a code list [8], equivalent to static single-assignment form [4]. It is a sequence of equations

$$v_i = \varphi_i(\text{relevant previous } v_j) \tag{2.1}$$

for $i = 1, \ldots, p+m$. The φ_i are given elementary functions and "relevant previous v_j" denotes those variables v_j that are the actual arguments of φ_i—necessarily all having $j < i$. Here, $v_{1-n}, \ldots, v_0$ are aliases for $\mathbf{f}$'s *input variables* $x_1, \ldots, x_n$, while $v_{p+1}, \ldots, v_{p+m}$ are aliases for $\mathbf{f}$'s *output variables* $y_1, \ldots, y_m$, and $v_1, \ldots, v_p$ are *intermediate variables.* That is, there are n inputs, p intermediates, and m outputs.

A code list describes the values calculated by a single execution-trace through the program code of $\mathbf{f}$. This paper does not study how a Jacobian code for functions, whose code contains branches and loops, may be generated. For standard VE this has been implemented in EliAD; see [16].

2.2. A simple AD example.

The sparse linear algebra view of computing a Jacobian by AD will be illustrated by a simple function $\mathbf{y} = \mathbf{f}(\mathbf{x})$ with three inputs and two outputs. The left column of (2.2) below shows a code list for $\mathbf{f}$, written in Matlab-like notation. On the right the code list variables, written in normal mathematical notation, are shown as functions of the inputs x_1, x_2, and x_3.

(2.2)

`function [y1,y2] = f(x1,x2,x3)`	
`v1 = x1*x2`	$v_1 = x_1x_2$
`v2 = sin(v1)`	$v_2 = \sin(x_1x_2)$
`v3 = 2*v2`	$v_3 = 2\sin(x_1x_2)$
`v4 = v3-x1`	$v_4 = 2\sin(x_1x_2) - x_1$
`y1 = x3*v4`	$y_1 = x_3\,(2\sin(x_1x_2) - x_1)$
`y2 = 3*v4`	$y_2 = 3\,(2\sin(x_1x_2) - x_1)$.

The inputs are `x1`, `x2`, and `x3`; the dependents are `y1` and `y2`; and the intermediates are `v1`, `v2`, `v3`, and `v4`. We wish to generate a code to calculate $J = \partial(y_1, y_2)/\partial(x_1, x_2, x_3)$, comprising $\partial y_1/\partial x_1$, $\partial y_1/\partial x_2$, etc.

The basic linear relations of AD are obtained by differentiating line by line:

(2.3)

`v1 = x1*x2`	$\mathrm{d}v_1 = x_2\,\mathrm{d}x_1 + x_1\,\mathrm{d}x_2$
`v2 = sin(v1)`	$\mathrm{d}v_2 = \cos(v_1)\,\mathrm{d}v_1$
`v3 = 2*v2`	$\mathrm{d}v_3 = 2\,\mathrm{d}v_2$
`v4 = v3-x1`	$\mathrm{d}v_4 = \mathrm{d}v_3 - \mathrm{d}x_1$
`y1 = x3*v4`	$\mathrm{d}y_1 = x_3\,\mathrm{d}v_4 + v_4\,\mathrm{d}x_3$
`y2 = 3*v4`	$\mathrm{d}y_2 = 3\,\mathrm{d}v_4$.

The d's mean *derivatives with respect to whatever parameters in which we are interested*. Eliminating intermediate $\mathrm{d}v_k$ to get the $\mathrm{d}y_i$ as linear combinations of the $\mathrm{d}x_j$,

$$\mathrm{d}y_i = \sum_j J_{ij}\,\mathrm{d}x_j,$$

and one obtains $J = [J_{ij}]$, the desired Jacobian matrix.

One way of computing J is by *classical forward AD*. Here, d means *gradient with respect to the input variables*. In our example, $\mathrm{d} = (\partial/\partial x_1, \partial/\partial x_2, \partial/\partial x_3)$. The process is shown in (2.4) below.

$$\boxed{\begin{array}{lllllll}
\text{Initialize with} & & & & & & \\
\quad \mathrm{d}x_1 & = & & & (1 & 0 & 0) \\
\quad \mathrm{d}x_2 & = & & & (0 & 1 & 0) \\
\quad \mathrm{d}x_3 & = & & & (0 & 0 & 1) \\
\text{and continue} & & & & & & \\
\quad \mathrm{d}v_1 & = & x_2\,\mathrm{d}x_1 + x_1\,\mathrm{d}x_2 & = & (x_2 & x_1 & 0) \\
\quad \mathrm{d}v_2 & = & \cos(v_1)\,\mathrm{d}v_1 & = & (\cos(v_1)\,x_2 & \cos(v_1)\,x_1 & 0) \\
\quad \mathrm{d}v_3 & = & 2\,\mathrm{d}v_2 & = & & \dots & \\
\quad \mathrm{d}v_4 & = & \mathrm{d}v_3 - \mathrm{d}x_1 & = & & \dots & \\
\text{ending with} & & & & & & \\
\quad \mathrm{d}y_1 & = & x_3\,\mathrm{d}v_4 + v_4\,\mathrm{d}x_3 & = & & \dots & \\
\quad \mathrm{d}y_2 & = & 3\,\mathrm{d}v_4 & = & & \dots &
\end{array}} \tag{2.4}$$

(formulas for entries in the last four rows omitted). This of course is done numerically at run time, not in the symbolic way suggested in (2.4). The process amounts to eliminating the $\mathrm{d}v_k$ by *forward substitution.*

2.3. A linear algebra viewpoint.

The equations in matrix form. The relations in (2.3) form a sparse linear system

(2.5)
$$\begin{array}{c|ccc|cccc|cc|}
 & & n=3 & & & p=4 & & & m=2 & \\
\hline
 & x_2 & x_1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
p{=}4 & 0 & 0 & 0 & \cos(v_1) & -1 & 0 & 0 & 0 & 0 \\
 & 0 & 0 & 0 & 0 & 2 & -1 & 0 & 0 & 0 \\
 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\
\hline
m{=}2 & 0 & 0 & v_4 & 0 & 0 & 0 & x_3 & -1 & 0 \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 & -1
\end{array}
\begin{bmatrix} \mathrm{d}x_1 \\ \mathrm{d}x_2 \\ \mathrm{d}x_3 \\ \hline \mathrm{d}v_1 \\ \mathrm{d}v_2 \\ \mathrm{d}v_3 \\ \mathrm{d}v_4 \\ \hline \mathrm{d}y_1 \\ \mathrm{d}y_2 \end{bmatrix} = \mathbf{0}.$$

The system illustrated by (2.5) may be written in general as

(2.6)
$$\begin{array}{cccc} & n & p & m \\ p & B & L-I & 0 \\ m & R & T & -I \end{array}
\begin{bmatrix} \mathrm{d}\mathbf{x} \\ \mathrm{d}\mathbf{v} \\ \mathrm{d}\mathbf{y} \end{bmatrix} = \mathbf{0}.$$

This is part of the *extended Jacobian* [8, p. 22] of Griewank, namely, $C_0 - I$, where C_0 is the square matrix

$$C_0 = \begin{array}{cccc} & n & p & m \\ n & 0 & 0 & 0 \\ p & B & L & 0 \\ m & R & T & 0 \end{array}.$$

The nonzeros of C_0 are the *local derivatives* c_{ij} of the code list. With rows and columns indexed $(1{-}n)$, $(2{-}n)$, ..., $(p{+}m)$—to match the aliased names for the input and output variables—we have

(2.7)
$$c_{ij} = \frac{\partial \varphi_i}{\partial v_j}, \quad 1 \le i \le p{+}m, \quad 1{-}n \le j \le p{+}m,$$

where φ_i is regarded as a function of *all* the v_j, taking c_{ij} as zero if v_j is not one of the variables on which φ_i depends.

The bottom right block of the matrix is shown as $-I$ in (2.6). This is not so when some output y_k is used in the definition of another output y_i. To eliminate such cases, introduce an extra intermediate that is a copy of y_k. We assume that this has been done. Then all required data are contained in the $(p{+}m) \times (n{+}p)$ matrix

(2.8)
$$C = \begin{array}{ccc} & n & p \\ p & B & \widetilde{L} \\ m & R & T \end{array},$$

where $\widetilde{L} = L - I$. Thus B holds the (local) derivatives of the intermediates w.r.t. the inputs, and so on. Following C_0's numbering, the rows are numbered $1, \dots, p{+}m$ and the columns $1{-}n, \dots, p$, so the -1's on the diagonal of $\widetilde{L}$ are in positions (k,k), $k = 1, \dots, p$. Submatrix L is strictly lower triangular because each variable depends only on previous ones. Thus $\widetilde{L}$ is nonsingular, being triangular with a nonzero diagonal.

The Jacobian J is a function purely of the entries of C. Solving (2.6) we find $\mathrm{d}\mathbf{y} = (R - T\widetilde{L}^{-1}B)\,\mathrm{d}\mathbf{x}$, and thus J is given by

$$J = J(C) = R - T\widetilde{L}^{-1}B, \tag{2.9}$$

the Schur complement of $\widetilde{L}$ in C. This was noted in Griewank [8].

Vertex and edge elimination. Griewank and Reese's presentation of *vertex elimination* [9] and Naumann's of *forward and backward edge elimination* [11, 12] are in terms of the computational graph. A key feature of them is that they transform the graph, equivalently C, by successive steps that keep $J(C)$ invariant. This can be seen as "obvious" because they transform the underlying linear equations in a way that does not change the relation between the input and output variables. However, it seems useful to give a matrix proof of it with Lemma 2.2 below.

From the linear algebra viewpoint, vertex and edge elimination are sequences of row or column operations on C as follows. The *simple row operation* on C whereby α_i times row k is added to row i, followed by multiplying row k by $\alpha_k \neq 0$, amounts to premultiplying C by the $(p{+}m) \times (p{+}m)$ matrix that is the identity with its kth column replaced by $(\alpha_1, \dots, \alpha_n)^T$. A *simple column operation* on C postmultiplies C by an $(n{+}p) \times (n{+}p)$ matrix with the transposed shape.

A forward (resp., backward) *edge elimination* step is such a row (column) operation that pivots in an intermediate row (column) with $\alpha_k = 1$ and only one other α_i (resp., α_j) nonzero, its value chosen to zero the (i,k) (resp., (k,j)) entry of C.

A VE step can be described symmetrically by either a row or a column operation. As a row operation it removes multiples of an intermediate row k $(k = 1, \dots, p)$ of C from other rows, to zero all of column k below the pivot. It is a standard step of Gaussian elimination (GE) where the update $c_{ij} = c_{ij} - c_{ik}c_{kj}/c_{kk}$ can be simplified because always $c_{kk} = -1$:

$$\left.\begin{array}{ll} c_{ij} = c_{ij} + c_{ik}c_{kj} & \text{for } i \text{ with } c_{ik} \neq 0 \text{ and } j \text{ with } c_{kj} \neq 0; \\ c_{ik} = 0 & \text{for } i > k; \\ \text{all other entries of } C \text{ unchanged.} & \end{array}\right\} \tag{2.10}$$

Regarded as a column operation, it does the same except that the second line reads $c_{kj} = 0$ for $j < k$. The k row and column are irrelevant to subsequent elimination steps and may be ignored—or deleted, which makes the row and column versions have identical effects.

A *complete vertex elimination* (CVE) consists of p steps (2.10), using all pivots $c_{kk} = -1\,(k = 1, \dots, p)$, in some order specified by the *pivot order*, a permutation $\pi = (\pi_1, \dots, \pi_n)$ of $1, \dots, p$.

To justify that CVE thus defined is indeed equivalent to GE, whatever the pivot order, we need to show that (i) no VE step alters any pivots from the value -1, and (ii) the lower triangular structure is preserved. These follow because $\widetilde{L}$ is lower triangular, so line 1 of (2.10) only affects the block to the left of and below (k,k), i.e., $i > k$ and $j < k$. The foregoing implies the following.

PROPOSITION 2.1. *CVE is equivalent to doing a symmetric permutation of the intermediate rows and columns into pivot order (thus keeping the -1's on the diagonal of $\widetilde{L}$) and then eliminating the intermediate variables by GE without interchanges.*

Equivalence is in the sense that GE and CVE do identical arithmetic operations. In inexact arithmetic, even the roundoff errors are the same.

2.4. An example elimination. For our example (2.2), the following traces the effect of elimination in pivot order (3 2 1 4). Nonzeros with a known constant value are shown by their value in a circle. Other nonzeros are marked $\bullet$, except that a value just created is marked $\ast$ if it updates an existing entry (thus requiring one multiplication and one addition to compute) or $\times$ if it fills in a previously zero position (requiring one multiplication only). The about-to-be-used pivot is shown in bold **-1**. The used pivot rows and columns are blanked out.

$$\begin{bmatrix} B & \widetilde{L} \\ R & T \end{bmatrix} = \begin{array}{c|ccc|cccc|} & x_1 & x_2 & x_3 & v_1 & v_2 & v_3 & v_4 \\ v_1 & \bullet & \bullet & 0 & -1 & 0 & 0 & 0 \\ v_2 & 0 & 0 & 0 & \bullet & -1 & 0 & 0 \\ v_3 & 0 & 0 & 0 & 0 & \textcircled{2} & \mathbf{-1} & 0 \\ v_4 & \textcircled{-1} & 0 & 0 & 0 & 0 & \textcircled{1} & -1 \\ \hline y_1 & 0 & 0 & \bullet & 0 & 0 & 0 & \bullet \\ y_2 & 0 & 0 & 0 & 0 & 0 & 0 & \textcircled{3} \end{array}.$$

Step 1: Subtract multiples of v_3 row from rows below.

$$\begin{array}{c|ccc|cccc|} & x_1 & x_2 & x_3 & v_1 & v_2 & v_3 & v_4 \\ v_1 & \bullet & \bullet & 0 & -1 & 0 & & 0 \\ v_2 & 0 & 0 & 0 & \bullet & \mathbf{-1} & & 0 \\ v_3 & & & & & & & \\ v_4 & \textcircled{-1} & 0 & 0 & 0 & \textcircled{2} & & -1 \\ \hline y_1 & 0 & 0 & \bullet & 0 & 0 & & \bullet \\ y_2 & 0 & 0 & 0 & 0 & 0 & & \textcircled{3} \end{array}.$$

Step 2: Subtract multiples of v_2 row from rows below.

$$\begin{array}{c|ccc|cccc|} & x_1 & x_2 & x_3 & v_1 & v_2 & v_3 & v_4 \\ v_1 & \bullet & \bullet & 0 & \mathbf{-1} & & & 0 \\ v_2 & & & & & & & \\ v_3 & & & & & & & \\ v_4 & \textcircled{-1} & 0 & 0 & \times & & & -1 \\ \hline y_1 & 0 & 0 & \bullet & 0 & & & \bullet \\ y_2 & 0 & 0 & 0 & 0 & & & \textcircled{3} \end{array}.$$

Step 3: Subtract multiples of v_1 row from rows below.

$$\begin{array}{c|ccc|cccc|} & x_1 & x_2 & x_3 & v_1 & v_2 & v_3 & v_4 \\ v_1 & & & & & & & \\ v_2 & & & & & & & \\ v_3 & & & & & & & \\ v_4 & \ast & \times & 0 & & & & \mathbf{-1} \\ \hline y_1 & 0 & 0 & \bullet & & & & \bullet \\ y_2 & 0 & 0 & 0 & & & & \textcircled{3} \end{array}.$$

Step 4: Subtract multiples of v_4 row from rows below.

$$\begin{array}{c|ccc|cccc|} & x_1 & x_2 & x_3 & v_1 & v_2 & v_3 & v_4 \\ v_1 & & & & & & & \\ v_2 & & & & & & & \\ v_3 & & & & & & & \\ v_4 & & & & & & & \\ \hline y_1 & \times & \times & \bullet & & & & \\ y_2 & \times & \times & 0 & & & & \end{array}.$$

At the end J occupies the R block. The above sequence requires 1 addition and 7 multiplications: compare 9 additions and 21 multiplications by forward AD if done as in (2.4) treating the $\mathrm{d}v_i$ as full 3-vectors. This may seem an unfair comparison, but even such a modern AD tool as TAPENADE uses full rather than sparse calculation in its "vector" mode, which is its most natural way to compute Jacobians.

2.5. VE preserves the Jacobian. Any product of simple row (resp., column) operations that pivot in a row (column) belonging to an intermediate variable has a matrix of the form

$$U = \begin{array}{cc} & \begin{array}{cc} p & m \end{array} \\ \begin{array}{c} p \\ m \end{array} & \begin{bmatrix} W & 0 \\ X & I \end{bmatrix} \end{array}, \left(\text{resp., } V = \begin{array}{cc} & \begin{array}{cc} n & p \end{array} \\ \begin{array}{c} n \\ p \end{array} & \begin{bmatrix} I & 0 \\ Y & Z \end{bmatrix} \end{array} \right), \tag{2.11}$$

where W and Z are nonsingular. Direct calculation verifies the following, which is actually a general property of Schur complements; see (2.9).

LEMMA 2.2.

(i) *Given C in* (2.8)*, the Jacobian $J(C)$ is unchanged by simple row (column) operations on C that pivot in an intermediate row (column), i.e., by any transformation $C = UCV$ with U and V as in* (2.11).

(ii) *Hence, any sequence of such operations that reduces $T\widetilde{L}^{-1}B$ to zero leaves $J(C)$ in the lower left block (the R block) of C.*

A VE step, whether as a row or a column operation, satisfies the conditions of Lemma 2.2. Any CVE reduces the T block to zero. Hence, by the lemma, at the end J occupies the R block of C.

2.6. The LU approach. In the graph view, a VE step amounts to creating, or updating, edges from one of vertex k's predecessors to one of its successors and then deleting vertex k. CVE is a sequence of atomic VE steps, each of which, by the lemma, keeps J invariant. The LU approach cannot be interpreted as steps that keep J invariant because the operations, though identical at the level of individual updates $c_{ij} = c_{ij} + c_{ik}c_{kj}$, are in a different order. This may be why it has not been considered by AD workers who are more comfortable with the graph view.

For what follows, we permute the intermediate variables to pivot order, swap the first and second column blocks, and index the right-hand column block as is then natural: $p+1, \ldots, p+n$ instead of $1-n, \ldots, 0$. This replaces C in (2.8) by

$$C^* = (c^*_{ij}) = \begin{array}{cc} & \begin{array}{cc} p & n \end{array} \\ \begin{array}{c} p \\ m \end{array} & \begin{bmatrix} (PLP^T - I) & PB \\ TP^T & R \end{bmatrix} \end{array} = \begin{array}{cc} & \begin{array}{cc} p & n \end{array} \\ \begin{array}{c} p \\ m \end{array} & \begin{bmatrix} \mathcal{L} & \mathcal{B} \\ \mathcal{T} & R \end{bmatrix} \end{array}, \tag{2.12}$$

say. Here P is the permutation matrix representing π. Matrix $\mathcal{L}$ is not lower triangular in general, but the pivots remain on its main diagonal: $c^*_{kk} = -1$ $(k = 1, \ldots, p)$. The Jacobian is still the Schur complement of the $p \times p$ block, $J = R - \mathcal{T}\mathcal{L}^{-1}\mathcal{B}$.

Since CVE is a partial GE, it is equivalent to an LU factorization. This equivalence, and various approaches to LU for sparse matrices, are discussed in [5, sections 3.5–7]. The discussion there is for square matrices but holds for the present case, too. In the following algorithm the "partial" behavior, in the sense that elimination is taken only as far as rows and columns 1 to p, is implemented by the upper bound of the sum being $\min\{i-1, j-1, p\}$ instead of the usual $\min\{i-1, j-1\}$.

ALGORITHM 2.1 (LU ON C^*).

```
For $i = 2, \dots, p+m$, $j = 2, \dots, p+n$, and $i \neq j$, in a suitable order
    set $c^*_{ij} = c^*_{ij} + \sum_{k=1}^{\min\{i-1, j-1, p\}} c^*_{ik} c^*_{kj}$.
At the end, $J$ is in the bottom right block of $C^*$.
```

The first row and column are unchanged. An order is "suitable" provided at each stage those quantities required on the right-hand side are already computed. Alternative orders and their memory access patterns are described in [5, section 3.7]. In our tests we used a *Crout* order, which computes the (not already found) entries of the second row and column, then of the third row and column, and so on.

The computational schemes we have tested are based on Algorithm 2.1. For a given pivot order, the floating point operation counts ("op counts") of the CVE, GE, and LU methods are identical, given they exploit sparsity equally. When one outperforms another it will be mainly due to the code making better use of registers and cache. LU also allows some useful optimizations at code-generation time.

3. Tools used in the experiments.

This section describes the tool used to generate Jacobian code for the experiments in section 4 and also a tool for generating "random functions."

3.1. Jacobian code generation tool.

Summary. First, an example illustrates the kind of Jacobian code that is generated. The entries of C are represented by simple variables rather than array elements. Our convention is that, e.g., the (v_{13} row, x_5 column) entry is the variable `cv13x5`.

A complete procedure to calculate J at a given input $\mathbf{x}$ consists of three parts. The first is the code of $\mathbf{f}$, augmented by statements to compute the local derivatives by AD. For instance, a statement `v17 = v15*x6` in the code list, i.e., $v_{17} = v_{15}x_6$, would be augmented by `cv17v15 = x6` and `cv17x6 = v15` representing $\partial v_{17}/\partial v_{15} = x_6$ and $\partial v_{17}/\partial x_6 = v_{15}$. For statement-level mode (described in subsection 4.1) the local derivatives were produced by AD reverse mode on each individual statement.

The second part of J is the elimination code, which is the topic of this paper. The final part packs up the variables that comprise J. For instance, in our simple example J is the collection

$$J = \begin{bmatrix} \texttt{cy1x1} & \texttt{cy1x2} & \texttt{cy1x3} \\ \texttt{cy2x1} & \texttt{cy2x2} & \texttt{0} \end{bmatrix}.$$

One must take care of structural zeros in J, such as J_{23} here.

We have to generate code for a sequence of assignments of the form

$$c_{ij} = c_{ij} + \sum_{k \in K} c_{ik} c_{kj}, \tag{3.1}$$

where we only include terms $c_{ik}c_{kj}$ for which both factors are (potentially) nonzero. In AD applications, a large proportion of the original entries are constants, independent of run-time problem data. For example, the local derivatives for $x \pm y$ are $1, \pm 1$; that for ax, where a is a constant, is a. If one propagates knowledge of constant c_{ij} values as the elimination proceeds, the Jacobian code can be much shortened.

GE-style elimination comprises assignments of the form $c_{ij} = c_{ik}c_{kj}$ (fill-in) or $c_{ij} = c_{ij} + c_{ik}c_{kj}$ (update), possibly altering a c_{ij} several times. A possible pseudocode

```
//Input:  local derivatives
//cv1x1,cv1x2,cv2v1,cv3v2,cv4x1,
//cv4v3, cy1x3, cy1v4, cy2v4
//eliminate v3
cv4v2  = cv4v3*cv3v2
//eliminate v2
cv4v1  = cv4v2*cv2v1
//eliminate v1
cv4x1 = cv4x1 + cv4v1*cv1x1
cv4x2  = cv4v1*cv1x2
//eliminate v4
cy1x1  = cy1v4*cv4x1
cy2x1  = cy2v4*cv4x1
cy1x2  = cy1v4*cv4x2
cy2x2  = cy2v4*cv4x2
```

```
//Input:  non-constant elem derivs
// cv1x1, cv1x2, cv2v1, cy1x3, cy1v4.
//Use cv3v2=2, cv4x1=-1, cv4v3=1, cy2v4=3
cv4v1 = 2*cv2v1
cv4x1 = -1 + cv4v1*cv1x1
cv4x2 = cv4v1*cv1x2
cy1x1 = cy1v4*cv4x1
cy2x1 = 3*cv4x1
cy1x2 = cy1v4*cv4x2
cy2x2 = 3*cv4x2
```

FIG. 3.1. *Generation of Jacobian code for the simple example. On the left: GE-style, not exploiting known constants. On the right: LU-style, exploiting constants.*

for the simple example (2.2), *not* exploiting constant values, is shown on the left side of Figure 3.1. It follows the steps shown graphically in subsection 2.4.

LU-style code generation for this example, with the entries generated in the order suggested earlier, and exploiting constants, yields the code on the right side of Figure 3.1. The problem is too simple to show the "fewer, longer statements" feature. Each assignment has the form (3.1) where c_{ij} on the right is omitted if it is currently zero. $K = K(i,j)$ is the set of indices k in the unreordered C that correspond to $k = 1, \dots, \min\{i-1, j-1, p\}$ in the reordered matrix C^* of (2.12).

Implementation outline. The LU-style Jacobian code ("Jcode") generation tool was written in MATLAB, which proved convenient because of its object-oriented programming features, high-level syntax, and support for sparse matrices. The tool takes as input (a) a file specifying the CG of the function **f**, (b) a specification of a heuristic (such as "Markowitz") for choosing the pivot sequence, and (c) a specification of the overall approach such as "LU" or "GE." It produces the same Jcode that a production tool might produce (only more slowly).

Separately, code to compute **f**'s local derivatives ("LDcode") is generated. A complete Jacobian subroutine is made of the LDcode followed by the Jcode, embedded in the template code of subroutine header and closing statements. For **f** given by preexisting Fortran code the CG file, LDcode, and template code were generated by processing **f**'s Fortran file with the ELIAD tool. For **f** generated by the "random function" tool of subsection 3.2, that tool produced the CG file, LDcode, and template code.

One can use a simple data structure that avoids the complexity of general compiler techniques. A combined numerical/symbolic elimination is done that can be summarized as follows:

Repeatedly, do (3.1) for a suitable sequence of triples i, j, and K.

The sequence of i, j, K's is determined during elimination by the overall elimination strategy (pivot order, GE or LU style, Crout, or some other order of operations) and the resulting fill-in. This is a version of the "loop-free code" approach to sparse matrix factorization described in [5, section 9.6].

The operation representing (3.1) acts on a "partially eliminated sparse matrix" object that knows which of its entries are compile-time constants and which are run-time values. Each call to the operation updates this matrix and may output a statement of Jacobian code. The following scheme works well in MATLAB. Numerical elimination is done on a sparse array C that initially holds the local derivatives c_{ij} in (2.7). At each stage, c_{ij} either holds a finite numeric value, meaning that constant value, or holds not a number (NaN), meaning a value unknown until run time. The rules of operations on NaN, plus care to avoid the operation $\mathrm{NaN} \times 0$, imply that this remains true throughout.

A code is generated for each occurrence of (3.1) that has least one NaN multiplied by a nonzero value, doing obvious simplifications. For instance, suppose $i = 12$, $j = 3$, and $K = \{4, 7, 8, 11\}$, representing the assignment

$$c_{12,3} = \overset{=-1}{c_{12,3}} + \overset{=-1}{c_{12,4}} * c_{4,3} + \overset{=1}{c_{12,7}} * \overset{=1}{c_{7,3}} + c_{12,8} * c_{8,3} + c_{12,11} * \overset{=3.75}{c_{11,3}},$$

where known values are marked above the relevant c_{rs}. Suppose, say, indices 3, 4, 7, 8, and 11 are intermediate variables, and 12 is an output variable y_1. Then when simplified using the known values it gives the line of code

```
cy1v3 = -cv4v3 + cy1v8*cv8v3 + cy1v11*3.75.
```

If the right side simplifies to the existing value of c_{ij} or to a constant, then (except for an entry in the final Jacobian) no code is generated.

Aliasing. Tests on our examples showed that a large proportion of the statements generated by this process were simple copies $c_{ij} = c_{rs}$ or $c_{ij} = -c_{rs}$. To eliminate these, a table records equivalence classes of the c_{ij} under the relation $c_{ij} = \sigma c_{rs}$, where $\sigma = \pm 1$. One member of the equivalence class is chosen as the "alias" of each other member and replaces it everywhere. If, in the above example, the table shows that $c_{8,3}$ has $-c_{4,3}$ as its alias, then the generated code changes to

```
cy1v3 = -cv4v3 - cy1v8*cv4v3 + cy1v11*3.75.
```

On the flow in channel (FIC) problem reported below, aliasing removed over 500 statements to leave just 244 statements—one per nonzero entry in J. For the random functions in our experiments it typically removed over 20% of statements.

Comments. This implementation was intended for modest-sized functions where most or all of the variables used fit within the cache of current machines. In particular storing the Jacobian entries as simple variables gives no control over how they are laid out in main memory.

When this works well, standard sparse matrix considerations of optimizing data transfer between CPU and main memory are not very important. Below, we also test what happens as the functions get "larger" and data transfer begins to dominate.

3.2. Random function tool. To enable rapid and extensive testing, we wrote another MATLAB tool that allows us to generate a "random function" $\mathbf{f}$ by selecting its numbers n, m, and p of inputs, outputs, and intermediate variables, respectively, plus a random number seed for reproducibility purposes. Each code list statement does one arithmetic operation $+, -, \times$, or $\div$, selected using the MATLAB random number generator with probabilities reflecting a typical numerical code. The tool outputs (a) the CG for use by the other tool, (b) a Fortran file that computes just $\mathbf{f}$, and (c) a Fortran file that computes the local derivatives, into which a code such as produced by our tool is inserted to make a complete subroutine.

TABLE 4.1
Platforms.

(a) Processors

Platform	Processor	CPU speed	L1-Cache	L2-Cache
AMD	AMD Opteron	2.2 GHz	32KB	1 MB
Intel	Pentium D	2.99 GHz	32KB	2 MB
DELL	Pentium M	1.7 GHz	64KB	512 KB

(b) Compilers

Platform	OS	Compiler	Options
AMD G95	Linux/Centos	G95 version 0.9	`-O3 -r8`
Intel G95	Windows XP	G95 version 0.9	`-O3 -r8`
DELL G95	Windows XP	G95-MinGW version 0.9	`-O3 -r8`
DELL Salford	Windows XP	Salford FTN95	`/optimise/p6/dreal` /silent/fpp
DELL Absoft	Windows XP	Absoft f95 Pro 10	`-w -O2 -N113`

4. Experiments. We compared the speed of a Jacobian code produced by the LU approach with that produced by (a) other vertex elimination techniques, (b) conventional AD tools ADIFOR, TAMC, or TAPENADE, (c) hand-coded Jacobian where available, and (d) finite-differencing (FD). This is carried out for several test problems on five different platforms (processor and compiler) described in Table 4.1.

We include FD because this is often used in applications where an exact J is not essential, such as in a Newton iteration to solve equations. FD can compete if it computes J faster than does AD (sufficiently to outweigh a slower overall convergence). This is almost never the case in the results we present, however.

We first present in detail performance data on two test cases for eight LU-style codes, preceded by results for other methods, cited from Forth et al. [6]. Then, we draw some observations which we further investigate using several examples, most of which were randomly generated.

In the tables $\mathbf{f}'$ denotes (the code for) the Jacobian of $\mathbf{f}$. For each technique, we give (i) $W(\mathbf{f}')/W(\mathbf{f})$, the ratio of the nominal number of floating-point operations within the generated Jacobian code to those in the function code, and (ii) the corresponding ratios $\mathrm{CPU}(\mathbf{f}')/\mathrm{CPU}(\mathbf{f})$ of CPU times for each platform. The $W(\mathbf{f}')$ and $W(\mathbf{f})$ counts were obtained using a Perl script to count the number of operations in executable statements. Each of `*`, `+` , `-`, and `/` counts as one operation. They are inaccurate estimates, since on the one hand, each elementary function such as `sqrt` is counted as one operation, as does raising to a power `**`. On the other hand, they do not take account of optimizations performed by the compiler: at the chosen optimization levels, the compilers perform constant value propagation and evaluation [7, p. 32] to avoid unnecessary operations, though the timings suggest that this is not always well done.

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 and by the ADIFOR or TAPENADE routine otherwise. Except for the FD results, discrepancies were always at the round-off level.

To improve consistency of timings, a number N_d (problem-dependent but typically several hundred) of sets of input data was generated for a given problem using the Fortran intrinsic routine `random_number`. The code for each technique was run N_d times, once for each data set (if the same data were used repeatedly, the cache might not be flushed, and a clever compiler might notice the reuse and "optimize out" evaluations after the first). The total time was divided by N_d to give the CPU time estimate for that code. The process was repeated several times and results computed for each repetition. The results for each repetition were nearly identical, showing that the rankings are not due to vagaries of measurement in a multitasking system.

4.1. Initial tests. The first tests present data for the same compiler on three machines, for the same machine with three compilers, and for a fairly large number of ways of generating Jacobian code. This illustrates that the performance varies strongly between compilers and between machines, as well as between codes generated by the same overall algorithm with slightly different heuristics. We use the two following examples. The distinction between statement-level (SL) and code-list (CL) is explained in the last paragraph of this page.

ROE. The 5×10 dense Jacobian of the Roe flux function [14], consisting of around 180 lines of Fortran 77 source code. A SL differentiation of the input code yields a sparse 67×72 matrix C with 198 nonzeros, while its CL differentiation leads to a sparse 213×218 matrix C with 344 nonzeros. Results are in Table 4.2.

FIC. The 32×32 Jacobian of the FIC problem from the MINPACK test set [3]. This is sparse, with 244 out of a possible 1024 nonzeros. A SL differentiation of the code yields a sparse 712×712 matrix C with 1276 nonzeros, while its CL differentiation leads to a sparse 1360×1360 matrix C with 1924 nonzeros. Results are in Table 4.3.

The FIC function uses loops heavily (nesting four deep), and this is reflected in some of the Jacobian codes. To apply the VE/LU methods, the FIC function code was loop-unrolled into straight-line form. We give figures for the code (marked "unrolled") that result from unrolling some of the Jacobian code produced by other tools. In addition to the op-count and CPU-time ratios described above, we give the Jacobian object code file size in kilobytes (Kb) (on AMD/g95). For FIC this shows, for instance, that the hand-coded, Tapenade, and Tamc-R Jacobians, coded using loops, are short but slow. The unrolled hand-coded version is the fastest, or close to it, on all the platforms, but now of comparable length to the (loop-free) VE and LU codes. Even for loop-free code, shorter is not always faster.

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.

The first rows are for the established tools Adifor, Tamc, and Tapenade, for a hand-coded Jacobian in the FIC case, and for approximation by one-sided finite differences. Various versions of VE follow, produced by the EliAD tool, and finally the LU methods. For VE and LU the mnemonics F, R, Mark, and VLR denote ways of choosing the pivot order: *forward* order, *reverse* order, the order given by the *Markowitz* algorithm, and Naumann's [12] *VLR* variant on Markowitz. P stands for preelimination: all vertices that have only one successor are eliminated first (working from the output end of the graph to the input), and then F, R, Mark, or VLR is applied to the remaining vertices.

SL means that the computational graph to which methods are applied is that of the original statements of the function code; CL means it is the graph resulting from breaking each statement into constituent elementary operations. For instance, the statement `c = sqrt(a*a + b*b)` generates one vertex of the SL graph[1] but four of the CL graph. Depth-first traversal (DFT) of the Jacobian code [17] is a postprocessing

[1]The local derivatives of a statement are computed by local reverse mode, as in Adifor.

TABLE 4.2
Tests on Roe flux.

Technique	$\frac{W(\mathbf{f}')}{W(\mathbf{f})}$	$\mathbf{f}'$ obj Kb(AMD)	CPU($\mathbf{f}'$)/CPU($\mathbf{f}$) by platform				
			AMD G95	Intel G95	Dell G95	Dell Salford	Dell Absoft
ADIFOR	15.95	17.0	17.01	15.76	9.37	6.96	13.29
TAPENADE	16.50	8.0	13.58	14.93	11.95	8.56	12.23
TAMC-F	21.18	10.1	22.06	25.26	13.58	12.96	15.94
TAMC-R	12.69	13.9	17.80	30.10	9.26	9.96	13.53
FD	12.14	–	12.83	15.28	14.53	10.52	12.41
VE-SL-F	8.89	30.9	45.19	44.68	26.16	7.87	13.23
VE-SL-R	7.32	25.6	42.27	43.21	24.79	7.30	11.23
VE-CL-F	12.85	27.5	41.35	42.08	25.32	16.48	13.77
VE-CL-R	9.50	26.1	42.26	45.33	24.47	13.52	11.29
VE-SL-P-F	7.85	27.3	6.77	9.56	4.47	4.61	5.41
VE-SL-P-R	6.78	22.7	5.78	9.21	$\underline{4.05}$	4.22	5.00
VE-CL-P-F	8.35	26.8	6.36	10.27	4.32	5.57	5.24
VE-CL-P-R	7.28	23.2	$\underline{5.64}$	9.09	$\underline{4.05}$	5.09	$\underline{4.29}$
VE-SL-P-F-DFT	7.85	26.0	7.20	10.57	5.00	5.26	5.59
VE-SL-P-R-DFT	6.78	21.6	6.22	8.08	4.63	4.96	4.65
VE-CL-P-F-DFT	8.35	25.4	6.42	10.98	4.58	8.61	5.24
VE-CL-P-R-DFT	7.28	21.0	5.72	8.56	4.58	8.39	4.41
VE-SL-P-Mark	7.35	25.8	6.44	11.27	4.47	4.39	5.00
VE-SL-P-VLR	6.60	22.9	5.92	8.68	4.21	4.13	4.35
VE-CL-P-Mark	7.86	26.1	6.08	11.27	4.53	5.17	5.12
VE-CL-P-VLR	7.11	22.9	$\underline{5.62}$	9.21	4.26	4.96	$\underline{4.29}$
VE-SL-P-Mark-DFT	7.35	23.9	6.37	8.97	5.16	4.87	5.18
VE-SL-P-VLR-DFT	6.60	20.7	5.99	7.79	4.53	4.70	4.59
VE-CL-P-Mark-DFT	7.86	23.5	6.30	9.26	4.53	8.35	5.06
VE-CL-P-VLR-DFT	7.11	19.8	**5.34**	7.85	**3.95**	8.22	**4.12**
SL-F-LU	4.25	23.8	7.30	6.85	5.11	4.35	5.71
SL-R-LU	3.66	21.2	6.13	6.85	4.37	$\underline{3.78}$	5.24
SL-Mark-LU	3.56	21.1	6.28	6.67	4.37	4.04	5.06
SL-VLR-LU	3.43	19.1	5.80	**5.61**	4.53	**3.74**	4.35
SL-P-F-LU	3.78	22.0	6.49	6.55	4.89	4.17	5.35
SL-P-R-LU	3.44	19.9	5.79	$\underline{6.14}$	4.32	**3.74**	4.88
SL-P-Mark-LU	3.54	20.9	6.24	7.03	4.47	4.00	5.59
SL-P-VLR-LU	3.36	18.7	**5.34**	6.02	$\underline{4.05}$	**3.74**	$\underline{4.29}$
Mean $\mathbf{f}$ evaluation time (μs)			.19	.25	.48	.53	.43

phase devised by Reid, where the statements are reordered aiming to place each assignment "$v = \ldots$" close to the statements that use v.

For each platform we give the mean time for one $\mathbf{f}$ evaluation, CPU($\mathbf{f}$), at the end of the table. For the Roe case, the variation in CPU($\mathbf{f}'$)/CPU($\mathbf{f}$) ratios is due only slightly to differences in function time but mainly to differences in Jacobian time. For FIC, the CPU($\mathbf{f}$) times for the three Dell compilers are in the wide ratio of G95 : Absoft : Salford $\approx$ 1 : 2 : 3, which distorts the figures—what look like very good ratios for Salford are due to its having a comparatively slow function.

The difference between platforms is striking. Those between different compilers on one machine suggest that they use very different methods to optimize code.[2] Note, for instance, on the FIC problem, comparing the Dell results, VE-CL-P-R-DFT is 4.4 times faster with Absoft than Salford, while ADIFOR-unrolled is 3.5 times faster

[2]Absoft support told us optimization level -O2 was suitable: higher levels made no appreciable speed difference to this kind of code while increasing the length of the binary. For G95, -O3 seemed best for similar reasons. Salford only has one optimization level; i.e., it is either "off" or "on."

TABLE 4.3
Tests on FIC problem.

Technique	$\frac{W(\mathbf{f}')}{W(\mathbf{f})}$	$\mathbf{f}'$ obj Kb(AMD)	CPU($\mathbf{f}'$)/CPU($\mathbf{f}$) by platform				
			AMD G95	Intel G95	Dell G95	Dell Salford	Dell Absoft
Hand-coded	1.91	1.4	6.29	6.72	5.92	2.83	4.22
Hand-coded unrolled	1.91	31.0	$\underline{2.52}$	5.47	**2.65**	**1.51**	3.26
ADIFOR unrolled	34.42	51.5	95.07	107.67	94.90	19.43	67.95
TAPENADE	35.99	2.1	106.36	102.13	94.49	31.51	68.98
TAMC-F unrolled	35.99	50.4	88.79	98.01	84.49	20.38	68.98
TAMC-R	44.64	2.5	106.47	110.21	90.75	33.59	73.19
FD unrolled	33.73	–	133.93	111.40	99.80	47.74	76.04
VE-SL-F	3.49	34.3	2.76	4.77	2.99	4.15	3.26
VE-SL-R	2.25	34.5	2.70	4.61	2.92	2.83	3.29
VE-CL-F	4.44	38.0	3.21	10.01	4.49	6.60	16.20
VE-CL-R	2.75	33.2	2.66	5.65	3.27	4.34	2.81
VE-SL-P-F	2.25	34.5	2.67	4.62	2.92	2.83	3.07
VE-SL-P-R	2.25	34.5	2.65	4.62	2.92	2.83	3.10
VE-CL-P-F	2.75	33.2	2.65	5.90	3.27	4.53	2.78
VE-CL-P-R	2.75	33.2	2.66	5.80	3.20	4.34	2.81
VE-SL-P-F-DFT	2.25	28.8	3.12	**3.52**	$\underline{2.72}$	3.96	2.36
VE-SL-P-R-DFT	2.25	28.8	3.12	$\underline{3.56}$	$\underline{2.72}$	4.15	2.40
VE-CL-P-F-DFT	2.75	27.4	2.89	4.66	$\underline{2.72}$	7.92	$\underline{1.82}$
VE-CL-P-R-DFT	2.75	27.4	2.91	4.62	$\underline{2.72}$	7.74	**1.76**
VE-SL-P-Mark	2.25	34.5	2.65	4.60	2.92	2.83	3.10
VE-SL-P-VLR	2.25	34.5	2.65	4.59	2.92	2.83	3.16
VE-CL-P-Mark	2.75	33.2	2.66	5.87	3.20	4.34	2.81
VE-CL-P-VLR	2.75	33.2	2.66	5.88	3.27	4.15	2.72
VE-SL-P-Mark-DFT	2.25	28.8	3.12	$\underline{3.57}$	$\underline{2.72}$	4.15	2.40
VE-SL-P-VLR-DFT	2.25	28.8	3.11	$\underline{3.56}$	**2.65**	4.15	2.36
VE-CL-P-Mark-DFT	2.75	27.4	2.91	4.67	$\underline{2.72}$	7.74	$\underline{1.79}$
VE-CL-P-VLR-DFT	2.75	27.4	2.89	4.69	2.79	7.74	$\underline{1.79}$
SL-F-LU	1.72	30.8	$\underline{2.47}$	5.57	2.99	$\underline{2.26}$	3.10
SL-R-LU	1.72	30.8	$\underline{2.45}$	5.73	2.99	$\underline{2.26}$	3.07
SL-Mark-LU	1.72	30.9	$\underline{2.45}$	5.45	2.99	$\underline{2.26}$	3.10
SL-VLR-LU	1.72	30.8	$\underline{2.48}$	5.48	2.99	$\underline{2.26}$	3.13
SL-P-F-LU	1.72	30.8	$\underline{2.46}$	5.64	2.92	$\underline{2.08}$	3.07
SL-P-R-LU	1.72	30.8	**2.44**	5.61	2.92	$\underline{2.26}$	3.10
SL-P-Mark-LU	1.72	30.8	$\underline{2.45}$	5.53	2.99	$\underline{2.26}$	3.13
SL-P-VLR-LU	1.72	30.8	$\underline{2.45}$	5.50	2.99	$\underline{2.26}$	3.10
Mean $\mathbf{f}$ evaluation time (μs)			0.61	0.81	0.92	3.3	2.0

with Salford than Absoft, a "discrepancy ratio" of $4.4 \times 3.5 = 15.4$. This is the most extreme of many similar discrepancies.

We believe that the variation between different machines with the same compiler is mainly due to how they manage the memory hierarchy. The processors we used have a cache-based memory hierarchy with the time to load data into the arithmetic registers increasing from the level-1 cache through the level-2 cache to the main memory. Arithmetic registers are limited, typically 32 or 64. Optimizing compilers usually seek to maximize performance by minimizing redundant calculations as well as the memory traffic between registers and cache. *Stalls* arise when an instruction calls for a value not yet loaded into registers causing the processor to wait. Some of the processors such as the AMD or Intel in 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. More details on such issues may be found in [7]. We do not at present have any way to predict the performance of one of our VE/LU techniques, though some work in this direction is presented in [15]. This included the study of assembler code and register usage. Because we were unable, so far, to find clear patterns emerging, we do not present such data here.

For both Roe and FIC problems, on all five platforms, all the VE/LU-style methods vastly outperform the standard tools and can even outperform "hand-coded unrolled." For the Roe problem, one of the LU methods gives best or nearly best performance on all platforms. For FIC, it is remarkable that 244 entries of $\mathbf{f}'$ are computed in such a small multiple of the time to compute only $\mathbf{f}$. This must be because FIC's $\mathbf{f}$ is in fact a linear function. For FIC, the LU methods are strikingly better than other VE methods on the Dell/Salford platform (hand-coded unrolled is even better), significantly worse than VE methods postprocessed by DFT on Intel/G95, and comparable with other VE methods on the other platforms. We have not tried postprocessing an LU-style code by DFT, but the results suggest it could be useful.

What is clear is that the LU results are far less sensitive to the pivot heuristic used than are the other VE results. In all our experiments, on all five platforms, the LU methods' performance varied only modestly over these eight pivot heuristics. Assuming this is true more generally it is a significant reason to prefer LU to other VE methods.

4.2. Further tests.

Subsection 2.4 mentioned that for the widely used AD tool Tapenade the simplest way to generate Jacobian code is *vector mode*, which does not exploit sparsity unless one uses separate methods such as Jacobian compression [8, Chapter 7]. One should thus expect the LU approach to outperform it, at least provided the problem is not so large that the LU code is dominated by data transfer. To investigate this further, the following examples compare it to Tapenade and to finite-differencing, on a number of problems of increasing size until the LU approach clearly runs into difficulties.

First we use the problems:

- *PLEI.* The 28×28 Jacobian of the Pleiades problem from the initial value problem test set [10]. This is sparse, with 210 out of a possible 784 nonzeros. Using for example, the CL differentiation, the corresponding C matrix is of size 630×630 with 994 nonzeros (sparsity 0.25%).
- *RC1.* The 300×300 Jacobian of a function randomly generated by our Matlab tool by setting n=300, m=300, and p=1450. This has 4635 nonzero entries out of a possible $90,000$. This is already in CL form, and its corresponding C matrix is of size 1750×1750 with 994 nonzeros.
- *RC2.* The 122×255 Jacobian of a function randomly generated by our Matlab tool by setting n=255, m=122, and p=2650. This has 7366 nonzero entries out of a possible $31,110$. This is already in CL form, and its corresponding C matrix is of size 2672×2905 with 5544 nonzeros.
- *RC3.* The 300×222 Jacobian of a function randomly generated by our Matlab tool by setting n=222, m=300, and p=3250. This has 14,569 nonzero entries out of a possible $66,600$. This is already in CL form and its corresponding C matrix is of size 3550×3472 with 7099 nonzeros.

For completeness, Table 4.4 gives nominal flop-counts, lines of code and function-time statistics for the above test cases (but not those used for Figure 4.1).

The Tapenade vector mode treats the jth column of the Jacobian J as the directional derivative $(\mathbf{e}_j \cdot \nabla)\,\mathbf{f}$, where $\mathbf{e}_j$ is the jth unit vector, and computes these

TABLE 4.4
Nominal floating-point operations counts $W(\mathbf{f})$, lines of code (l.o.c.), and CPU times for test problem functions.

Problem	$W(\mathbf{f})$ (Flops)	L.o.c.	Function CPU time(μs) AMD G95	Intel G95	Dell G95	Dell Salford	Dell Absoft
ROE	222	139	.19	.25	.48	.53	.43
FIC	1266	759	.61	.81	.92	3.3	2.0
PLEI	735	238	.70	.86	13.	3.9	9.3
RC1	1750	1750	3.0	5.2	7.5	9.0	10.
RC2	2772	2772	4.8	7.5	13.	50.	13.
RC3	3550	3550	6.1	8.8	–	21.	20.

TABLE 4.5
Tests on PLEI and random problems. For reasons of space, the names of the LU methods are abbreviated, e.g., P-VLR has the meaning of SL-P-VLR-LU in the previous tables. A runnable program for RC3 could not be created on Dell/G95.

Technique	$\frac{W(\mathbf{f}')}{W(\mathbf{f})}$	CPU($\mathbf{f}'$)/CPU($\mathbf{f}$) by platform AMD G95	Intel G95	Dell G95	Dell Salford	Dell Absoft
PLEI problem						
HAND-CODED	1.25	1.40	1.57	2.29	1.50	1.36
TAPENADE	12.3	8.55	10.2	19.0	8.87	7.82
FD	5.29	29.5	29.9	31.6	30.0	26.3
LU best	2.83	2.23	2.78	4.92	2.70	1.85
Achieved by		P-Mark	R	P-VLR	F	P-VLR
RC1 problem, $(n, m, p) = (300, 300, 1450)$ with 4635 nonzeros						
TAPENADE	331.	7760.	3980.	2810.	2240.	589.
FD	301.	483.	517.	460.	362.	272.
LU best	4.12	488.	332.	426.	341.	147.
Achieved by		P-VLR	VLR	P-Mark	P-Mark	P-VLR
RC2 problem, $(n, m, p) = (255, 122, 2650)$ with 7366 nonzeros						
TAPENADE	636.	7280.	4310.	3440.	3990.	450.
FD	256.	284.	203.	298.	94.5	297.
LU best	6.86	111.	51.	52.8	10.0	51.2
Achieved by		P-Mark	VLR	F	F	F
RC3 problem, $(n, m, p) = (222, 300, 3250)$ with 14,569 nonzeros						
TAPENADE	554.	6550.	6310.	–	1240.	1030.
FD	223.	293.	271.	–	259.	190.
LU best	7.68	326.	310.	–	99.2	104.
Achieved by		VLR	VLR		P-Mark	P-R

in a loop one column at a time. The unit vectors are the columns of the $n \times n$ identity matrix, which is input to the process as a "seed matrix." (For a general $n \times q$ seed matrix S, the output is JS: this can be exploited, e.g., by Jacobian compression methods.) The vector mode also uses optimization techniques such as common subexpression elimination and some code motion. However, it may perform useless calculations such as multiplying by or adding to zero.

Results are shown in Table 4.5. For the random functions, the flop-counts of LU-style codes are dramatically less than those of the corresponding TAPENADE code. This is because these represent sparse computations and the LU approach can exploit the sparsity. The LU-style codes run roughly 4 to 20 times as fast as the TAPENADE ones. However, FD (though of course inaccurate) provides run times similar to and occasionally faster than the LU-based approach for the random functions.

To investigate this observation, we applied these techniques on more codes generated by our random function tool. Figure 4.1 shows the behavior with increasing

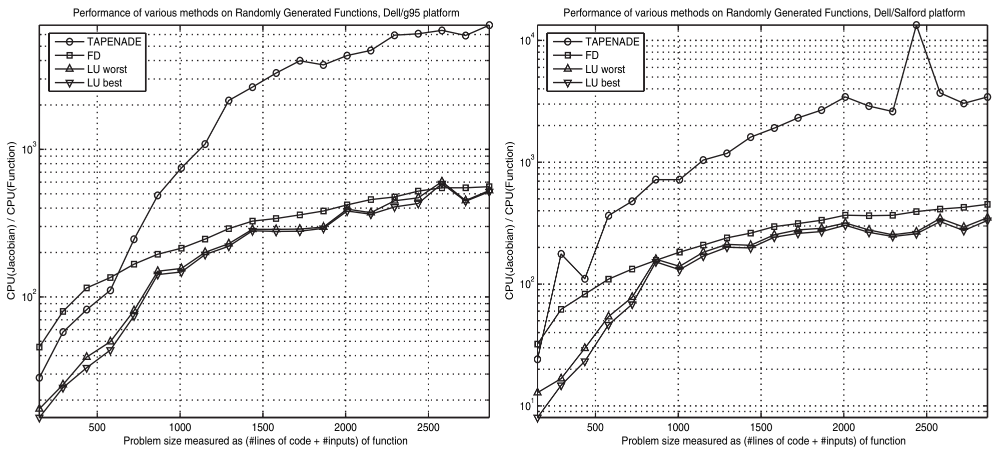

FIG. 4.1. *Performance on random functions of sizes* $(n, m, p) = (15i+1, 15i+6, 113i)$ *for* $i = 1, 2, \ldots, 20$ *(largest size in figure is* $(301, 306, 2260)$*). The number of statements (each an elementary operation) in* $\mathbf{f}$ *is* $m + p$*, the size of the Jacobian is* $n \times m$*, and FD calls the function* $n + 1$ *times to compute the Jacobian. The sparsity of the Jacobian decreases fairly uniformly from* 58% *for the smallest to* 10.3% *for the largest.*

size, graphically. Random functions of sizes $(n, m, p) = (15i + 1, 15i + 6, 113i)$ for $i = 1, 2, \ldots, 20$ were generated. The figure shows the $\text{CPU}(\mathbf{f}')/\text{CPU}(\mathbf{f})$ ratios of TAPENADE, FD, and the best and worst of the eight LU methods. In this test, clearly the latter are quite insensitive to the choice of heuristic; as problem size increases TAPENADE is progressively slower than them, while FD is catching up. The anomalous peak and trough in the TAPENADE results with Salford are genuine: repeated runs gave almost identical results.

It seems that the key factor is not the length of code but the size, or probably the number of nonzero entries, of the Jacobian. On the AMD and Intel platforms, above a certain size, FD is faster than the LU techniques—probably because the Jacobian entries cannot fit into the cache and data transfer is dominating the computation. The level-1 cache holds about 4,000 double-precision values on the AMD and Intel machines, and about 8,000 on the Dell.

How important is the pivot order? In the Roe example, VE with simple forward and reverse order, both at the statement and at the code-list level, was far worse than the other VE/LU-style methods on all the platforms, but combining them with pre-elimination gave orders competitive with the other methods. In the FIC case the different orders for VE and LU, with and without DFT postprocessing, were far more similar to each other with the exception of "outliers" such as the bad time for VE-CL-F on Intel/G95 and the fact that VE-CL-P-R-DFT was the best on Dell/Absoft but nearly the worst VE/LU-style method on Dell/Salford. For the random functions, we only have different LU-style methods to compare: again, the pivot order does not make a dramatic difference in the tests shown.

We have also tried randomly generated pivot orders. They always gave performance far *worse* than the ones presented in this paper. Thus, the order matters, but at present we have no way to tell which order will be best for a given platform.

5. Conclusions.

The VE approach to calculating the Jacobian J of a vector function $\mathbf{f}$ is usually described in terms of the computational graph. Earlier work, e.g., [6], using the EliAD tool, showed the potential speedup of VE-style methods compared with other AD tools for generating Jacobian code. We have shown how VE is equivalent to a (partial) Gaussian elimination or LU factorization of a sparse matrix. This view offers different ways to sequence the operations that are not obvious in the graph approach.

For **f** with a straight-line code, we have used source transformation to generate a straight-line Jacobian code using an order of operations derived from a Crout compact LU factorization. For the previous VE work with EliAD, elimination pivot orders had been generated by Markowitz-style and other heuristics. For each such order a corresponding LU code was produced. On all platforms tried, the best LU version was always nearly as fast as the best VE version and often significantly faster; the LU codes were less sensitive to the choice of pivot heuristic. Both VE and LU are usually many times faster than codes generated by tools such as Adifor, Tamc, and the more recent Tapenade. For the FIC problem an LU version was significantly faster than a hand-coded Jacobian on three out of the five platforms used.

These initial problems were small enough for all the values in the computation to fit in the cache of current machines. The large performance variation between platforms, and between different VE/LU methods on the same platform, is therefore mainly due to differences in pipeline, register, and cache scheduling on different systems. We believe that a deeper understanding of these issues will lead to significant further speedups by reordering the operations involved in factorization and that, for problems of this size, data transfer from and to main memory is less important.

Cache usage will be improved by minimizing the *active set* of computed values at any stage that need to be used in the future. The DFT approach (subsection 4.1) tried to do this with uncertain success; many sparse matrix methods as in Duff, Erisman, and Reid [5, Chapters 7–10] describe approaches whose matrix access patterns are in a sense intermediate between Gaussian elimination and compact LU factorization and might be used to "tune" the active set.

Implementing AD directly within the compiler, as the CompAD project of Naumann et al. [13] is doing with the numerical algorithms group (NAG) compiler, is a new development that is likely to lead to just such improvements in low-level optimization of AD.

The pivot order matters, but at present we have no way to tell which order will be best for a given platform. To get near-best performance of VE/LU-style Jacobians, one may follow the methods of the Automatically Tuned Linear Algebra Software (ATLAS) project [2] to generate and run alternative codes automatically and choose the one that runs fastest on a given platform.

We then examined "larger" functions, using randomly generated functions of increasing size, to see what happens when there are too many values to fit into cache, with the LU codes eventually becoming slower than FD. We believe that the number of nonzeros in the final Jacobian has more effect than the number of intermediate values in the computation. For these larger problems, main memory traffic is obviously key, and the approach of storing matrix entries as simple variables should be replaced by one that gives control over memory arrangement and data structure. Standard sparse factorization techniques [5] then become far more relevant.

In summary, our results indicate that a Jacobian code based on LU factorization should be used in such cases as:

- a heavily used function of moderate length that can be coded up without loops, such as the Roe flux and similar flux functions whose Jacobians are needed repeatedly for assembling global Jacobians in CFD, or

- a basic block within a larger section of code, in cases where a compiler can determine, or be told, that this basic block is crucial to the overall performance of the program.

Acknowledgments. We wish to thank the anonymous referees, as well as Shaun Forth and John Reid, for their careful reading of the text and constructive comments.

REFERENCES

[1] P. R. Amestoy, X. S. Li, and E. G. Ng, *Diagonal Markowitz Scheme with Local Symmetrization*, SIAM J. Matrix Anal. Appl., 29 (2007), pp. 228–244.

[2] J. Dongarra, A. Petitet, R. C. Whaley, et al., *ATLAS (Automatically Tuned Linear Algebra Software) Project*, http://math-atlas.sourceforge.net.

[3] B. M. Averick, R. G. Carter, J. J. Moré, and G.-L. Xue, *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, IL, 1992. See ftp://info.mcs.anl.gov/pub/MINPACK-2/tprobs/P153.ps.Z.

[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, *Efficiently computing static single assignment form and the control dependence graph*, ACM Trans. Program. Lang. Syst., 13 (1991), pp. 451–490.

[5] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices*, Oxford Science Publications, London, 1986.

[6] S. A. Forth, M. Tadjouddine, J. D. Pryce, and J. K. Reid, *Jacobian code generated by source transformation and vertex elimination can be as efficient as hand-coding*, ACM Trans. Math. Software, 30 (2004), pp. 266–299.

[7] S. Goedecker and A. Hoisie, *Performance Optimization of Numerically Intensive Codes*, SIAM, Philadelphia, 2001.

[8] A. Griewank, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, Frontiers Appl. Math. 19, SIAM, Philadelphia, 2000.

[9] A. Griewank and S. Reese, *On the calculation of Jacobian matrices by the Markowitz rule*, in Automatic Differentiation of Algorithms: Theory, Implementation, and Application, A. Griewank and G. F. Corliss, eds., SIAM, Philadelphia, 1991, pp. 126–135.

[10] F. Mazzia and F. Iavernaro, *Test Set for Initial Value Problem Solvers*, Technical report, Department of Mathematics, University of Bari, 2003. See www.dm.unibs.it/~testset.

[11] U. Naumann, *Efficient Calculation of Jacobian Matrices by Optimized Application of the Chain Rule to Computational Graphs*, Ph.D. thesis, Technical University of Dresden, 1999.

[12] U. Naumann, *Elimination techniques for cheap Jacobians*, in Automatic Differentiation: From Simulation to Optimization, Computer and Information Science, G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, eds., Springer, New York, 2001, pp. 241–246.

[13] U. Naumann, B. Christianson, J. Riehme, and D. Gendler, *Differentiation Enabled Fortran Compiler Technology*, University of Aachen, Germany, University of Hertfordshire, UK, and Numerical Algorithms Group Ltd., UK. See http://www.nag.co.uk/nagware/research/ad_overview.asp.

[14] P. L. Roe, *Approximate Riemann solvers, parameter vectors, and difference schemes*, J. Comput. Phys., 43 (1981), pp. 357–372.

[15] M. Tadjouddine, F. Bodman, J. D. Pryce, and S. A. Forth, *Improving the performance of the vertex elimination algorithm for derivative calculation*, in AD2004: Proceedings of the 4th International Conference on Automatic Differentiation, Lect. Notes Comput. Sci. Eng. 50, M. Bücker et al., ed., Springer, New York, 2005, pp. 111–120.

[16] M. Tadjouddine, S. A. Forth, and J. D. Pryce, *Hierarchical automatic differentiation by vertex elimination and source transformation*, in Proceedings of ICCSA (2), Lecture Notes in Comput. Sci. 2668, V. Kumar et al., ed., Springer, New York, 2003, pp. 115–124.

[17] M. Tadjouddine, S. A. Forth, J. D. Pryce, and J. K. Reid, *Performance issues for vertex elimination methods in computing Jacobians using automatic differentiation*, in Proceedings of ICCS (2), Lecture Notes in Comput. Sci. 2330, P. M. A. Sloot et al., ed., Springer, New York, 2002, pp. 1077–1086.