Proceedings of the 9th WSEAS International Conference on Applied Mathematics, Istanbul, Turkey, May 27-29, 2006 (pp536-541)

On Symbolic Jacobian Accumulation

EBADOLLAH VARNIK and UWE NAUMANN
RWTH Aachen University, Department of Computer Science
Seffenter Weg 23, D-52056 Aachen, Germany

Abstract: Derivatives are essential ingredients of a wide range of numerical algorithms. We focus on the accumulation of Jacobian matrices by Gaussian elimination on a sparse implementation of the extended Jacobian. A symbolic algorithm is proposed to determine the fill-in. Its runtime undercuts that of the original accumulation algorithm by a factor of ten. On the given computer architecture we are able to handle problems with roughly four times the original size.

Key–Words: Jacobian Accumulation, Extended Jacobian, Symbolic Elimination.

1 Introduction

The context of this paper is *automatic differentiation* [1, 3, 2] of numerical programs. We consider vector functions

$$F : I\!R^n \supseteq D \rightarrow I\!R^m, \quad \mathbf{y} = F(\mathbf{x}) \quad , \tag{1}$$

that map a vector $\mathbf{x} \equiv (x_i)_{i=1,\ldots,n}$ of *independent* variables onto a vector $\mathbf{y} \equiv (y_j)_{j=1,\ldots,m}$ of *dependent* variables. We assume that F has been implemented as a computer program. Hence, it can be decomposed into a sequence of p single assignments of the value of scalar *elemental* functions φ_i to unique *intermediate* variables v_j. This *code list* of F is given as

$$(I\!R \ni)\ v_j = \varphi_j(v_i)_{i \prec j} \quad , \tag{2}$$

where $j = n+1, \ldots, q$ and $q = n + p + m$. The binary relation $i \prec j$ denotes a direct dependence of v_j on v_i. So, $P_j = \{i : i \prec j\}$ is the index set of the arguments of φ_j. Similarly, $S_j = \{i : j \prec i\}$ is the index set of the elemental functions that have v_j as an argument. The variables $\mathbf{v} = (v_i)_{i=1,\ldots,q}$ are partitioned into the sets X containing the *independent* variables $(v_i)_{i=1,\ldots,n}$, Y containing the *dependent* variables $(v_i)_{i=n+p+1,\ldots,q}$, and Z containing the intermediate variables $(v_i)_{i=n+1,\ldots,n+p}$. The code list of F can be represented as a directed acyclic *computational graph* $G = G(F) = (V, E)$ with integer vertices $V = \{i : i \in \{1, \ldots, q\}\}$ and edges $(i, j) \in E$ if and only if $i \prec j$. Moreover, $V = X \cup Z \cup Y$, where $X = \{1, \ldots, n\}$, $Z = \{n+1, \ldots, n+p\}$, and $Y = \{n+p+1, \ldots, q\}$. Hence, X, Y, and Z are mutually disjoint. We distinguish between *independent* ($i \in X$), *intermediate* ($i \in Z$), and *dependent* ($i \in Y$) vertices. Under the assumption that all elemental functions are continuously differentiable in some neighborhood of their arguments all edges (i, j) can be labeled with the partial derivatives $c_{j,i} \equiv \frac{\partial v_j}{\partial v_i}$ of v_j w.r.t. v_i. This labeling yields the *linearized* computational graph G of F. From now on we use the notation G to refer to the linearized computational graph.

Equation (2) can be written as a system of nonlinear equation $C(\mathbf{v})$ [4] as follows:

$$\varphi_j(v_i)_{i \prec j} - v_j = 0 \quad \text{for } j = n+1, \ldots, q \quad . \tag{3}$$

Proceedings of the 9th WSEAS International Conference on Applied Mathematics, Istanbul, Turkey, May 27-29, 2006 (pp536-541)

Differentiation with respect to $\mathbf{v}$ leads to

$$C' = C'(\mathbf{v}) \equiv (c'_{j,i})_{i,j=1,\ldots,q} = \begin{cases} c_{j,i} & \text{if } i \prec j \\ -1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} . \quad (4)$$

The *extended Jacobian* C' is lower triangular. Its rows and columns are enumerated as $j, i = 1, \ldots, q$. Row j of C' corresponds to vertex j of G and contains the partial derivatives $c_{j,k}$ of vertex j w.r.t. all of its predecessors $k \in P_j$. In the following we refer to a row i as *independent* for $i \in \{1, \ldots, n\}$, as *intermediate* for $i \in \{n+1, \ldots, n+p\}$, and as *dependent* if $i \in \{n+p+1, \ldots, q\}$.

The focus of this paper is on finding *fill-in* generated during the Jacobian accumulation by *Gaussian* elimination on C'. The structure of the paper is as follows: In Section 2 we introduce a *symbolic* algorithm that uses a sparse bit pattern to detect fill-in. Section 3 presents runtime and memory analysis.

1.1 Elimination Techniques

The *Jacobian matrix* (or simply *Jacobian*) of F as defined in Equation (1) at point $\mathbf{x}_0$ is defined as follows:

$$(I\!R^{m\times n} \ni)\ F' = F'(\mathbf{x}_0) \equiv \left(\frac{\partial y_i}{\partial x_j}(\mathbf{x}_0)\right)_{j=1,\ldots,n}^{i=1,\ldots,m} .$$

F' can be obtained by eliminating all intermediate vertices $j \in Z$ from G as introduced in [5]. Each predecessor $i \in P_j$ of j is connected with all successors $k \in S_j$. If $(i,k) \notin E$, then it has to be generated and labeled with $c_{k,i} := c_{k,j} \cdot c_{j,i}$. Otherwise the value of $c_{k,i}$ is updated as $c_{k,i} := c_{k,i} + c_{k,j} \cdot c_{j,i}$. In the former case we say that fill-in is generated whereas *absorption* takes place in the latter. The elimination of vertex j can be understood as some sort of Gaussian elimination of all non-zero entries in row/column j of C'. Therefore one has to find all those rows k with $j \prec k$. In order to eliminate row/column j we perform the following transformation on C'.

Definition 1 (Row/Column Elimination in C')

$$c_{k,i} := c_{k,i} + c_{k,j} \cdot c_{j,i} \quad \forall i \prec j \wedge \forall k : j \prec k \quad (5)$$

$$c_{j,i} := 0 \quad \forall i \prec j \quad (6)$$

$$c_{k,j} := 0 \quad \forall k : j \prec k \quad (7)$$

$$c_{j,j} := 0 \quad . \quad (8)$$

Note that $c_{k,i} = 0$ if $i \not\prec k$. The new partial derivatives of v_k, $j \prec k$, with respect to v_i, $i \prec j$, are computed by applying the chain rule in Equation (5). Hence, any sensitivities of the v_k on v_j as well as of v_j on any of the v_i are removed in Equation (6) and Equation (7), respectively. *Fill-out* is generated. Setting the diagonal entry $c_{j,j}$ to zero in Equation (8) leads to the removal of the j-th row and column in C'. If $c_{k,i} = 0$ then Equation (5) leads to fill-in, otherwise it yields absorption.

1.2 Example

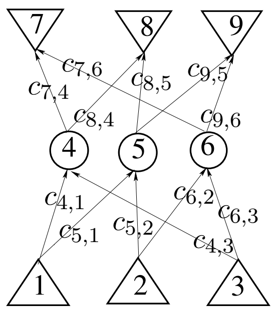
Consider the vector function $F : R^3 \to R^3$ whose code list is given in Figure 1(a). The corresponding G and C' are shown in Figure 1 (b) and (c), respectively. The symbols $\triangle$ represent independent, $\bigtriangledown$ dependent, and $\bigcirc$ intermediate vertices. Consider row 5 in Figure 1 (c) containing $c_{5,1}$ and $c_{5,2}$. These are labels of incoming edges $(1,5)$ and $(2,5)$ of vertex 5 in Figure 1 (b). Column 5 contains the partial derivatives $c_{8,5}$ and $c_{9,5}$ that are the labels of outgoing edges $(5,8)$ and $(5,9)$ of vertex 5. In the context of symbolic elimination we are merely interested in the sparsity structure of C'. Hence, $\times$ represents fill-in, $\bigcirc$ represents fill-out, and blanks represent zeros in C'.

Eliminating $c_{5,1}$ is equivalent to *front-elimination* [6] of $(1,5)$ as shown in Figure 2 (a). Fill-in is generated as $c_{8,1}$ $[(1,8)]$ and $c_{9,1}$ $[(1,9)]$ since rows [vertices] 8 and 9 have non-zeros [incoming edges] in [from] column [vertex] 5.

The elimination of the row/column [vertex] 5 in C' [G] can be done by elimination [front-elimination] of all non-zeros [incoming edges] in [to] row/column [vertex] 5. The resulting fill-in, namely $c_{8,1}$,

$$
\begin{aligned}
v_1 &= x_1\\
v_2 &= x_2\\
v_3 &= x_3\\
v_4 &= v_1 * v_3\\
v_5 &= v_1 * v_2\\
v_6 &= v_2 * v_3\\
v_7 &= v_4 * v_6\\
v_8 &= v_4 * v_5\\
v_9 &= v_5 * v_6
\end{aligned}
$$

(a)

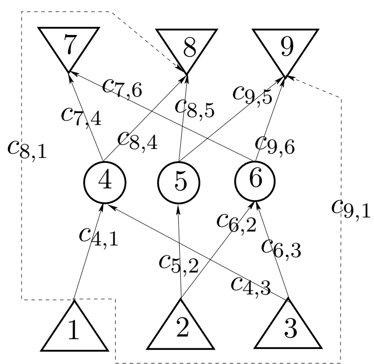


(b)

$$
\begin{bmatrix}
-1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0\\
c_{4,1} & 0 & c_{4,3} & -1 & 0 & 0 & 0 & 0 & 0\\
c_{5,1} & c_{5,2} & 0 & 0 & -1 & 0 & 0 & 0 & 0\\
0 & c_{6,2} & c_{6,3} & 0 & 0 & -1 & 0 & 0 & 0\\
0 & 0 & 0 & c_{7,4} & 0 & c_{7,6} & -1 & 0 & 0\\
0 & 0 & 0 & c_{8,4} & c_{8,5} & 0 & 0 & -1 & 0\\
0 & 0 & 0 & 0 & c_{9,5} & c_{9,6} & 0 & 0 & -1
\end{bmatrix}
$$

(c)

Figure 1: Code list (a); linearized computational graph G (b); C' (c) of F.



(a)

$$
\begin{bmatrix}
-1 & & & & & & & & \\
 & -1 & & & & & & & \\
 & & -1 & & & & & & \\
c_{4,1} & & c_{4,3} & -1 & & & & & \\
\bigcirc & c_{5,2} & & & -1 & & & & \\
 & c_{6,2} & c_{6,3} & & & -1 & & & \\
 & & & c_{7,4} & & c_{7,6} & -1 & & \\
\times & & & c_{8,4} & c_{8,5} & & & -1 & \\
\times & & & & c_{9,5} & c_{9,6} & & & -1
\end{bmatrix}
$$

(b)

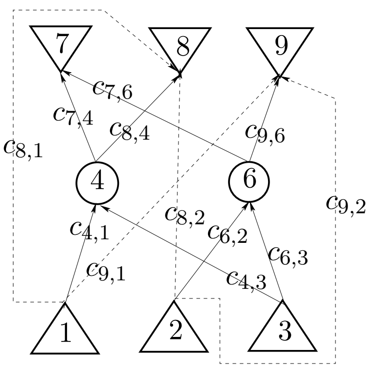
Figure 2: G [C'] after front-elimination [elimination] of $(1, 5)$ [$c_{5,1}$] (a) [(b)].

$c_{8,2}$, $c_{9,1}$, and $c_{9,2}$ [$(1, 8)$, $(2, 8)$, $(1, 9)$, and $(2, 9)$] in C' [G] is shown in Figure 3 (b) [(a)]. A total of $p!$ different row [vertex] elimination orderings in C' [G'] are possible. In this paper we focus on *reverse elimination* $(n + p, \cdots, n + 1)$. Hence, the Jacobian F' [the bipartite graph G'] is derived from C' [G] by elimination of all intermediate rows [vertices] in order $(6, 5, 4)$. The result is shown in Figure 4 (b) [(a)].

2 Symbolic Elimination Algorithm

Our symbolic fill-in detection algorithm uses a bit pattern $B = B(F)$ to hold the sparsity structure of C'. Figure 5 (a) shows the corresponding integer matrix for the extended Jacobian C' in Figure 1 (c). The binary representation is shown in Figure 5 (b). The symbolic algorithm is implemented in C++. Therefore we start counting with zero. Whenever we refer to the j-th row in B we mean the row with index $j - 1$.

Algorithm 1 (Symbolic Algorithm)
IN : B — bit pattern of C'



(a)

$$\begin{bmatrix} -1 & & & & & & & & \\ & -1 & & & & & & & \\ & & -1 & & & & & & \\ c_{4,1} & & c_{4,3} & -1 & & & & & \\ \bigcirc & \bigcirc & & & 0 & & & & \\ & c_{6,2} & c_{6,3} & & & -1 & & & \\ & & & c_{7,4} & & c_{7,6} & -1 & & \\ \times & \times & & c_{8,4} & \bigcirc & & & -1 & \\ \times & \times & & & \bigcirc & c_{9,6} & & & -1 \end{bmatrix}$$

(b)

Figure 3: G $[C']$ after elimination of vertex [row/column] 5 (a) [(b)].

$$\begin{pmatrix} 0 \\ 0 \\ 0 \\ 40960 \\ \mathbf{49152} \\ 24576 \\ 5120 \\ 6144 \\ 3072 \end{pmatrix} \qquad \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$

(a) (b)

Figure 5: Bit pattern B as an integer matrix (a) and binary representation of C' (b).

OUT: B — filled bit pattern after reverse elimination

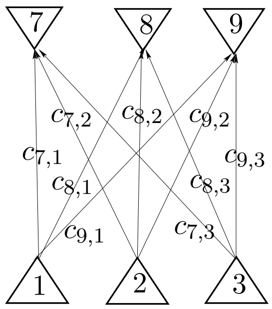
[1] **FOR** $i = n + p - 1, \ldots, n$
[2] **FOR** $j = q - 1, \ldots, i$
[3] $k := i \gg 4;$
[4] **IF** $(\ B[j][k] \ \wedge \ 1 \ll (15 - i\%16)\ )$
[5] **FOR** $m = 0, \ldots, k$
[6] $B[j][m] := B[j][m] \vee B[i][m];$

Consider the symbolic elimination of row 6 in Figure 5 (a) using Algorithm 1 with $i = 5$ and $j = 8$ in line *[1]* and *[2]*, respectively. The integer values corresponding to rows 6 and 9 are stored in column $k = 0$ (line *[3]*) with $B[5][0] = 24576$ and $B[8][0] = 3072$. $6 \prec 9$ as in line *[4]* $24576 \wedge 2^{15-5} = true$. Hence, $B[8][0] = 27648 = 24576 \vee 3072$. Line *[5]* in Algorithm 1 performs the bitwise OR for all affected columns of B.

In the following we apply Algorithm 1 to the bit pattern of F shown in Figure 5 (a). The result is shown in Figure 6 (b). Symbolic elimination proceeds as follows:

$$\begin{pmatrix} 0 \\ 0 \\ 0 \\ 40960 \\ 49152 \\ 24576 \\ 5120 \\ 6144 \\ 3072 \end{pmatrix} \overset{elim(6)}{\rightarrow} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 40960 \\ 49152 \\ 24576 \\ \mathbf{29696} \\ 6144 \\ \mathbf{27648} \end{pmatrix}$$

Proceedings of the 9th WSEAS International Conference on Applied Mathematics, Istanbul, Turkey, May 27-29, 2006 (pp536-541)



(a)

$$\begin{bmatrix}
-1 & & & & & & & & \\
 & -1 & & & & & & & \\
 & & -1 & & & & & & \\
\bigcirc & & \bigcirc & 0 & & & & & \\
\bigcirc & \bigcirc & & & 0 & & & & \\
 & \bigcirc & \bigcirc & & & 0 & & & \\
\times & \times & \times & \bigcirc & & \bigcirc & -1 & & \\
\times & \times & \times & \bigcirc & \bigcirc & & & -1 & \\
\times & \times & \times & & \bigcirc & \bigcirc & & & -1
\end{bmatrix}$$

(b)

Figure 4: Bipartite graph G' (a) and the corresponding structure of C' (b) after reverse elimination; The Jacobian is the 3×3 matrix in the lower left corner of C' after the elimination procedure.

$$\xrightarrow{elim(5)} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 40960 \\ 49152 \\ 24576 \\ 29696 \\ \mathbf{55296} \\ \mathbf{60416} \end{pmatrix} \xrightarrow{elim(4)} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 40960 \\ 49152 \\ 24576 \\ \mathbf{62464} \\ \mathbf{63488} \\ 60416 \end{pmatrix}$$

where

$$29696 = 2^{14} + 2^{13} + 5120;$$
$$27648 = 2^{14} + 2^{13} + 3072;$$
$$55296 = 2^{15} + 2^{14} + 6144;$$
$$60416 = 2^{15} + 27648;$$
$$62464 = 2^{15} + 29696;$$
$$63488 = 2^{13} + 55296 \quad .$$

3 Numerical Results

We compare runtime and memory consumption of our new symbolic algorithm (**SymAlgOnB**) on bit

$$\begin{pmatrix} 0 \\ 0 \\ 0 \\ 40960 \\ 49152 \\ 24576 \\ \mathbf{62464} \\ 63488 \\ 60416 \end{pmatrix}$$

(a)

$$\begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
\mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & 0 & \mathbf{1} & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0
\end{pmatrix}$$

(b)

Figure 6: B (a) and the corresponding binary representation (b) after symbolic elimination.

pattern B with reverse elimination of all intermediate rows of C' (**REOnEJ**). Both methods are applied to the following function:

Listing 1: f.cpp

```
void f(double* x, int n, int l) {
  double * h = new double [n];
  for(i=0; i<l; i++){
    if(i%2==0) {
      h[0] = x[n-1]*x[0];
```

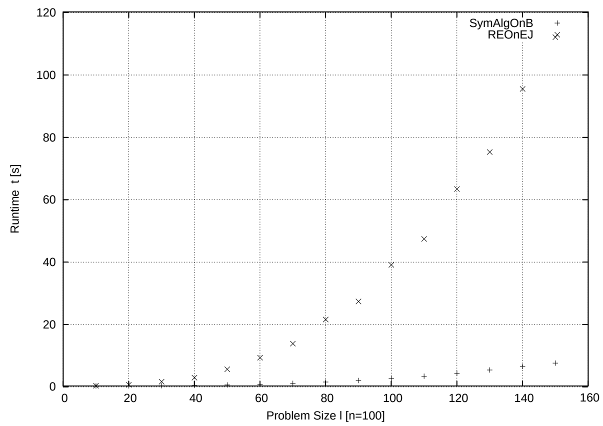


Figure 7: Runtime of **SymAlgOnB** vs. **REOnEJ**.

```
        for(j=1; j<n; j++)
          h[j] = x[j-1]*x[j]; }
      else {
        x[0] = h[n-1]*h[0];
        for(j=1; j<n; j++)
          x[j]=h[j-1]*h[j];
      }
    }
}
```

We set $n = 100$ and $l \in \{10, \cdots, 150\}$. Obviously, $C' \in I\!R^{q\times q}$ where $q = (l + 1) \cdot n$. All results have been obtained on an Intel Pentium 4 CPU running at 3.00GHz with 1GB of memory. We observe that the symbolic reverse elimination on B is about ten times faster than the corresponding procedure on C' as illustrated in Figure 7. On the given computer architecture we are able to handle problems of sizes $l = 250$ and $l = 1000$ (for $n = 100$) using **REOnEJ** and **SymAlgOnB**, respectively.

4 Conclusion

Jacobian accumulation on the extended Jacobian can be improved significantly – both in terms of memory requirement and overall runtime – by using static sparse storage allocated based on the result of a symbolic elimination algorithm to determin the generated fill. The use of bit pattern implementation as integer array has proved suitable for performing the symbolic elimination at a computational cost that undercuts that of the original algorithm significantly. We intent to use the symbolic algorithm in the context of a novel Jacobian accumulation method that uses elimination techniques on a sparse represenation of the extended Jacobian.

References:

[1] M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors. *Computational Differentiation: Techniques, Applications, and Tools*, Proceedings Series, Philadelphia, 1996. SIAM.

[2] G. Corliss, C. Faure, A. Griewank, L. Hascoet, and U. Naumann, editors. *Automatic Differentiation of Algorithms – From Simulation to Optimization*, New York, 2002. Springer.

[3] G. Corliss and A. Griewank, editors. *Automatic Differentiation: Theory, Implementation, and Application*, Proceedings Series, Philadelphia, 1991. SIAM.

[4] A. Griewank. *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Applied Mathematics. SIAM, Philadelphia, 2000.

[5] A. Griewank and S. Reese. On the calculation of Jacobian matrices by the Markovitz rule. In *[3]*, pages 126–135, 1991.

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