Cornell Theory Center Technical Report CTC96TR238

Structure and efficient Jacobian calculation *

Thomas F. Coleman† Arun Verma‡

March 12, 1996

Abstract

Many computational tasks require the determination of the Jacobian matrix, at a given argument, for a large nonlinear system of equations. Calculation or approximation of a Newton step is a related task. The development of robust automatic differentiation (AD) software allows for "painless" and accurate calculation of these quantities; however, straightforward application of AD software on large-scale problems can require an inordinate amount of computation. Fortunately, large-scale systems of nonlinear equations typically exhibit either sparsity or structure in their Jacobian matrices. In this paper we proffer general approaches for exploiting sparsity and structure to yield efficient ways to determine Jacobian matrices (and Newton steps) via automatic differentiation.

1 Introduction

A fundamental computation with regard to a nonlinear system, $F : \Re^n \to \Re^m$, is the evaluation of the Jacobian matrix of F at any given argument x: $J(x) \in \Re^{m\times n}$. Given a computer code to evaluate $F(x)$, the techniques of automatic differentiation (AD) can be used to compute $J(x)$. There are two basic modes of automatic differentiation, forward and reverse, e.g., [11, 12]. Forward mode AD yields J in time proportional to $n \cdot \omega(F)$, where $\omega(F)$ is the number of flops to evaluate $F(x)$. Alternatively, reverse mode AD yields $J(x)$ in time proportional to $m \cdot \omega(F)$.

For large problems the computation of J by a straightforward application of either mode of AD can be unacceptably expensive. The purpose of this paper is to show how it is possible to dramatically lower the cost of computing J by exploiting structure and sparsity in the application of AD.

Recently, techniques for the efficient determination of *sparse* Jacobian matrices J, via AD, have been developed [1, 9, 13]. The bi-coloring approach of Coleman and Verma [9], as discussed in §2, rests on the observation that is is usually possible to define "thin" matrices V, W such that the nonzero elements of J can be readily extracted from the pair

*To appear: Proceedings of the Second SIAM International Workshop on Computational Differentiation, Sante Fe, February, 1996. This research was partially supported by the Applied Mathematical Sciences Research Program (KC-04-02) of the Office of Energy Research of the U.S. Department of Energy under grant DE-FG02-90ER25013, and in part by the Advanced Computing Research Institute, a unit of the Cornell Theory Center which receives major funding from the National Science Foundation and IBM Corporation, with additional support from New York State and members of its Corporate Research Institute.

†Computer Science Department and Center for Applied Mathematics, Cornell University, Ithaca NY 14850.

‡Computer Science Department, Cornell University, Ithaca NY 14850.

$(W^T J, JV)$. Suppose W is an m-by-t_W matrix and V is an n-by-t_V matrix. The matrix $W^T J$ can be computed, directly, using AD in the reverse mode in time proportional to $t_W \cdot \omega(F)$; the matrix JV can be computed, directly, using AD in the forward mode in $t_V \cdot \omega(F)$ time.

For example consider the following n-by-n Jacobian, symmetric in structure but not in value:

$$J = \begin{pmatrix} \Box & \triangle & \triangle & \triangle & \triangle \\ \Box & \diamond & & & \\ \Box & & \diamond & & \\ \Box & & & \diamond & \\ \Box & & & & \diamond \end{pmatrix}. \tag{1}$$

Define $V = (e_1, e_2 + e_3 + e_4 + e_5)$; $W = (e_1)$, where we follow the usual convention of representing the i^{th} column of the identity matrix with e_i. Clearly elements $\Box, \diamond$ are directly determined from the product JV; elements $\triangle$ are directly determined from the product $W^T J$. Hence the work required to detemine J by computing $(W^T J, JV)$ using AD in reverse and forward mode respectively, is $3 \cdot \omega(F)$, *regardless of* n.

Unfortunately, not all large systems exhibit sparse Jacobian matrices. For example, the following composite structure is common in large-scale problems:

$$F(x) = \bar{F}(y) \tag{2}$$

where y is the solution to a large sparse positive definite system,

$$Ay = \tilde{F}(x), \tag{3}$$

and $A = A(x)$. The Jacobian of $F(x)$, $J(x)$, is almost always dense even when matrices $\bar{J}, \tilde{J}$, and $A_x y$ are sparse (which is typical) where $\bar{J}$ is the Jacobian of $\bar{F}$ with respect to y, $\tilde{J}$ is the Jacobian of $\tilde{F}$, and $A_x y$ is the Jacobian of the mapping $A(x)y$ (with respect to x). To see this consider that

$$J = \bar{J} A^{-1} [\tilde{J} - A_x y]. \tag{4}$$

It is the application of A^{-1} that causes matrix J to be dense – this will almost surely be the case unless A^{-1} is very special, e.g., diagonal.

So, direct application of *sparse AD* techniques offers no advantage in this case. However, it is possible to exploit the structure of this composite function and apply the sparse AD techniques at a deeper level. To see this consider the following "program" to evaluate $z = F(x)$, given x:

$$\boxed{\begin{array}{lrcl} \text{“Solve” } \textit{for } y_1 : & y_1 - \tilde{F}(x) & = & 0 \\ \textit{Solve for } y_2 : & Ay_2 - y_1 & = & 0 \\ \text{“Solve” } \textit{for } z : & z - \bar{F}(y_2) & = & 0. \end{array}}$$

But this program can be viewed as a nonlinear system of equations in (x, y_1, y_2) with corresponding Newton equations:

$$J_E \begin{pmatrix} \delta_x \\ \delta_{y_1} \\ \delta_{y_2} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ -F(x) \end{pmatrix}, \tag{5}$$

where

$$J_E = \begin{bmatrix} -\tilde{J} & I & 0 \\ A_x y_2 & -I & A \\ 0 & 0 & -\bar{J} \end{bmatrix}. \tag{6}$$

Here is a key point: the "extended" Jacobian matrix J_E is sparse and clearly sparse AD-techniques, e.g., [1, 9, 13], can be applied with respect to

$$F_E(x,y) = \begin{pmatrix} y_1 - \tilde{F}(x) \\ A(x)y_2 - y_1 \\ -\bar{F}(y_2) \end{pmatrix}. \tag{7}$$

to efficiently determine J_E. For example, the work required by the bi-coloring technique developed in [9] is $\chi \cdot \omega(F_E) = \chi \cdot \omega(F)$ where χ is a "bi-chromatic number" dependent on the sparsity of J_E. Typically, $\chi << \min(m,n)$. Additional linear algebra work is needed to extract J from J_E: compute the Schur complement (introducing zero matrices in positions $(3,2),(3,3)$) and obtain,

$$J = \bar{J}A^{-1}[\tilde{J} - A_x y].$$

If it is the Newton step $\delta_x = -J^{-1}F(x)$ that is required then it is not necessary to explicitly form J. For example the extended system (5) can be solved directly. This can afford significant savings. To illustrate, consider the following experiment. We define a composite function $F(x)$ following the form described above. The functions $\tilde{F}$ and $\bar{F}$ are defined to be the Broyden [3] function (the Jacobian is tridiagonal). The structure of A is based on the 5-point Laplacian defined on a regular $\sqrt{n}$-by-$\sqrt{n}$ grid. Each nonzero element of $A(x)$ depends on x in a trivial way such that the stucture of matrix $A_x \cdot v$, for an arbitrary vector v, is equal to the structure of matrix A. In particular, for all (i,j), $i \neq j$ where A_{ij} is nonzero the function $A_{ij}(x)$ is defined, $A_{ij} = x_j$.

Figure 1 plots the time to calculate the Newton step, given J_E, via the formulation of J using (4) versus the computation of the Newton step using a direct sparse solve for equation (5). Experiments were perfomed in MATLAB, with sparse system solving implemented using the "backslash" function. All matrices are sparse in this example except for the final Jacobian matrix J. Clearly it pays to avoid the formulation of J and the advantage grows with n.

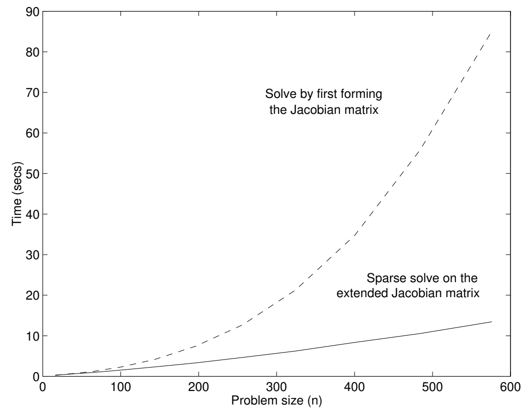

FIG. 1. *Comparison of two approaches to calculate the Newton step*

It is also possible to compute an approximate Newton step, without forming J, using an iterative solver. Specifically, if a sparse factorization of A is computed, an iterative solver involving only matrix-vector products can be applied to

$$(\bar{J}A^{-1}[\tilde{J} - A_x y])s = -F(x). \tag{8}$$

The main purpose of this paper is to illustrate how these ideas can be applied more generally: in many cases the natural "coarse-grained" program yields a sparse "extended" Jacobian matrix which in turn, can be efficiently computed by sparse AD-techniques.

2 Calculation of sparse Jacobians via bi-coloring

New techniques for the efficient computation of *sparse* Jacobian matrices are developed in [1, 9, 13]. In this section we briefly highlight the approach proposed in [9], coined "bi-coloring".

Bi-coloring is an attempt to define "thin" matrices V and W, based on the the "coloring" of two graphs representing the sparsity structure of the Jacobian matrix, such that the nonzero elements of J can easily be extracted from the calculated pair $(W^T J, JV)$. Given an arbitrary n-by-t_V matrix V, product JV can be directly calculated using automatic differentiation in the "forward mode"; given an arbitrary m-by-t_W matrix W, the product $W^T J$ can be calculated using automatic differentiation in the "reverse mode", e.g., [11, 12]. The motivation for the bi-coloring approach stems from the sparse finite-differencing literature [4, 5, 6, 7, 8, 10] where graph coloring is used to partition the columns of a sparse Jacobian matrix J and subsequently define a matrix V such that J can be determined from the product JV. However, matrix V is not guaranteed to be thin, even if J has considerable sparsity : consider a sparse matrix J with a single dense row. Alternatively, a solution based on partitioning of rows can be employed to define a matrix W such that J can be determined from $W^T J$. Again, it is very easy to construct examples, where defining thin W is not possible : consider the case where J has a single dense column.

Bi-coloring circumvents the dense row/dense column problem as illustrated in (1). In [9] bi-coloring approaches are proposed that allow for both the *direct* calculation of J – the non-zero elements of J are are extracted for the AD-computed products $(W^T J, JV)$ *directly*, without further computations – and determination by *substitution* where the the non-zero elements of J are are extracted for the AD-computed products $(W^T J, JV)$ using a *substitution process*. The advantage of the latter is that typically thinner matrices V, W can be defined and therefore the application of AD is less costly; the disadvantage of determination by substitution is that the substitution process itself incurs round-off error. Hence, the computed Jacobian matrix is usually less accurate than the directly determined Jacobian. Nevertheless, as discussed in [9], the loss of accuracy is usually minimal.

Computational experiments are reported in [9] which illustrate the effectiveness of bi-coloring as opposed to strictly 1-sided schemes. A 1-sided scheme may be column-based: partition the columns of J to define a thin matrix V such that the non-zeroes of J can be determined form the product JV (computed via forward mode AD). Alternatively, a 1-sided scheme may be row-based: partition the rows of J to define a thin matrix W such that the non-zeroes of J can be determined form the product $W^T J$ (computed via reverse mode AD).

Bi-coloring preformance on a collection of test matrices arising in linear programming is summarized in Table 1. There are 15 matrices in this collection; the table entries are the total number of "colors" required by the different approaches to compute Jacobian matrices with these structures. The amount of work to determine a sparse Jacobian matrix in this context is proportional to the number of colors required times the work to evaluate the function F.

Clearly on this set of sparsity structures bi-coloring is a significant win over 1-sided calculations . Moreover, bi-coloring combined with determination by substitution represents the least-cost approach. Additional experiments, with more detail, and further related discussion is given in [9].

Bi-coloring		1-sided Coloring	
Direct	Substitution	column	row
337	270	1753	452

TABLE 1
Totals for LP Collection (http://www.netlib.org/lp/data)

3 Structure

Our thesis can be summarized as follows. Large-scale nonlinear systems $F(x) = 0$ often exhibit a natural lower Hessenberg form. Usually, an easy programmatic way to describe F, at a high level, is to state this lower Hessenberg description, or program, F_E. The corresponding Jacobian of F_E, J_E, is typically a sparse matrix: sparse AD techniques can be applied to efficiently compute J_E. The Jacobian of F and/or the Newton step $\delta_x = -J^{-1}F(x)$ can subsequently be computed from J_E. Two key points are:

- The matrix J_E, though larger than J, is usually considerably sparser.
- The high level program F_E is usually readily available to the user: it is often the most natural way of expressing F.

To be more precise, a natural way to evaluate the nonlinear systems $z = F(x)$ is via the lower Hessenberg program illustrated in Figure 2 where we assume equation i *uniquely* determines y^i, $i = 1 : p$. We take the point of view that the function F_E is explicitly

$$\begin{array}{l}
\textit{Solve for } y_1: \;\; F_1(x, y_1) = 0 \\
\textit{Solve for } y_2: \;\; F_2(x, y_1, y_2) = 0 \\
\vdots \\
\textit{Solve for } y_p: \;\; F_p(x, y_1, y_2, \ldots, y_p) = 0 \\
\textit{“Solve” for output } z: \;\; z - F_{p+1}(x, y_1, y_2, \ldots, y_p) \;=\; 0
\end{array}$$

FIG. 2. *A General Structured Computation*

available, where

$$F_E = \begin{pmatrix} F_1 \\ F_2 \\ \vdots \\ F_p \end{pmatrix}.$$

Indeed, usually the component functions of F_E, F_i, $i = 1 : p$, are conveniently available to the user.

Many well-known structured problems are covered by this view: e.g., partially-separable functions, dynamical (recursive) systems, composite functions, various gradient computations. Examples are illustrated in Section §4.

The program in Figure 2 can be differentiated to give the extended Newton system:

$$J_E \begin{pmatrix} \delta x \\ \delta y_1 \\ \delta y_2 \\ \vdots \\ \delta y_p \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ -F \end{pmatrix} \tag{9}$$

where

$$J_E = \left(\begin{array}{c|cccc} \frac{\partial F_1}{\partial x} & \frac{\partial F_1}{\partial y_1} & & & \\ \frac{\partial F_2}{\partial x} & \frac{\partial F_2}{\partial y_1} & \frac{\partial F_2}{\partial y_2} & & \\ \vdots & \vdots & & \ddots & \\ \frac{\partial F_p}{\partial x} & \frac{\partial F_p}{\partial y_1} & \cdots & & \frac{\partial F_p}{\partial y_p} \\ \hline \frac{\partial F_{p+1}}{\partial x} & \frac{\partial F_{p+1}}{\partial y_1} & \cdots & & \frac{\partial F_{p+1}}{\partial y_p} \end{array}\right). \tag{10}$$

If each "interior vector" y_i is uniquely determined then the the super-diagonal of (10) has nonsingular blocks, $\frac{\partial F_i}{\partial y_i}$, and δ_x is the Newton step $\delta_x = -J^{-1}F(x)$.

We contend that J_E will likely be sparse, considerably sparse than J; hence, sparse AD techniques can be used to obtain J_E. The Jacobian matrix J can be computed from J_E by zeroing the $(2,2)$-block row in J_E using block Gauss transformations. Matrix J shows up in the $(2,1)$ location after elimination of the $(2,2)$-block row.

If it is the Newton step that is required, and not matrix J per se, then there are two natural alternatives to the explicit computation of J, given J_E. First, system (9) can be solved directly using a direct sparse factorization[1]: this is clearly an attractive option in some cases, e.g., Figure 1. A second possibility is to perform the $(2,1)$-elimination symbolically to produce a "product form" expression for J, in the $(2,1)$ location, which could then be used in any iterative linear solver requiring only matrix-vector products. An example of this latter possibility is given in the penultimate paragraph in §1.

3.1 Gradient computation

An important special case of Jacobian evaluation is the computation of the gradient of a scalar valued function, $f : \Re^n \rightarrow \Re$. The gradient of f is merely the transpose of the Jacobian of f; hence, the last "block row" of J_E in (10) is a single row vector. In general the strategies discussed above cannot improve upon a direct reverse-mode application of AD to evaluate the gradient, in terms of time, since the reverse mode computes $\nabla f(x)$ in time proportional to $\omega(f)$. However, unveiling underlying stucture as discussed above can certainly help significantly if only forward-mode AD is to be used.

For example, consider the case where f is a partially separable function, $f = f_1 + f_2 + \cdots + f_p$, where $f_i : \Re^n \rightarrow \Re$, $i = 1 : p$, and each component function f_i depends on only a few components of x. A natural way to evaluate f at a given argument x is to evaluate

$$F(x) = \begin{pmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_p(x) \end{pmatrix}$$

[1]Griewank [11] has proposed a similar idea in a more extreme form: F_E is defined with respect to *all* intermediate variables. The resulting extended Jacobian matrix J_E is huge, but very sparse.

and then sum the components of $F(x)$. A program to do this can be written:

$$\boxed{\begin{array}{lrcl} \text{“Solve” for } y\text{:} & y - F(x) & = & 0 \\ \text{“Solve” for } z\text{ :} & z - e^T y & = & 0. \end{array}}$$

In this case

$$J_E = \begin{pmatrix} -J_F & I \\ 0 & -e^T \end{pmatrix}. \tag{11}$$

Clearly J_E can be computed in time proportional to $\chi_I(J_F) \cdot \omega(f)$ by applying the bi-coloring/AD approach to F to obtain J_F. This special case, the efficient determination of a gradient of a partially separable function via the forward-mode of AD, is studied in detail in [2].

4 Examples

We discuss three common classes of structured nonlinear systems. In each case the Jacobian matrix is potentially dense; however, by differentiating the natural high-level program to compute F, as discussed in §3, underlying sparsity can be exploited in the use of AD tools. The result is often a dramatic increase in efficiency.

4.1 Composite Functions and Dynamical Systems

A composite function $F : \Re^n \to \Re^m$ can be written

$$F(x) = \bar{F}(T_p(T_{p-1}(\ldots(T_1(x))\ldots), \tag{12}$$

where, in general, functions $\bar{F}, T_i, i = 1 : p$ are vector maps. Recursively applying the chain rule yields J, the Jacobian of $F(x)$:

$$J = \bar{J} \cdot J_p \cdot J_{p-1} \cdot \ldots \cdot J_1$$

where J_i is the Jacobian of T_i evaluated at $T_{i-1}(T_{i-2}(\ldots(T_1(x))\ldots)$; $\bar{J}$ is the Jacobian of $\bar{F}$ evaluated at argument $T_p(T_{p-1}(\ldots(T_1(x))\ldots)$.

A natural high-level program to evaluate F is given below where we let y_0 denote x:

$$\boxed{\begin{array}{l} \text{for } i = 1 : p \\ \qquad \text{“Solve” for } y_i\text{: } y_i - T_i(y_{i-1}) \;=\; 0 \\ \text{“Solve” for } z\text{: } z - \bar{F}(y_p) \;=\; 0. \end{array}}$$

Clearly this program is a special case of the general form F_E given in Figure 2, with corresponding Jacobian matrix J_E:

$$J_E = \begin{pmatrix} -J_1 & I & & & \\ & -J_2 & I & & \\ & & \ddots & \ddots & \\ & & & -J_p & I \\ & & & & -\bar{J} \end{pmatrix}$$

If a Newton step is required, it can be much more costly to determine J directly via AD compared to determining J_E using AD and then solving the extended system (9) to determine the Newton step.

For example, consider the special case of a dynamical system: $T_i = T, \quad i = 1 : p-1$, $\bar{F} = T$, $T : \Re^n \to \Re^n$ is a square nonsingular mapping, and

$$F(x) \;=\; T(T(\ldots(T(x))\ldots)).$$

Assume that the Jacobian of T is a tri-diagonal Jacobian matrix.

For all p sufficiently large the Jacobian matrix J will be dense; hence determination of J by direct application of automatic differentiation requires $O(n{\cdot}\omega(F)) \;=\; O(n{\cdot}p{\cdot}\omega(T))$ flops. Therefore direct determination of J followed be a direct solve requires $O(n \cdot p \cdot \omega(T) \;+\; n^3)$ flops. However, the determination of J_E requires $O(p \cdot \omega(T))$ flops and solution of the banded extended system takes $O(n \cdot p)$ flops for a total of $O(p \cdot \omega(T) \;+\; n \cdot p)$ flops – generally, a much more attractive order of work.

4.2 Generalized partial separability

We define a *generalized* partially separable vector-valued function,

$$F(x) = \bar{F}(y_1, y_2, \ldots, y_p); \qquad y_i = T_i(x), \quad i = 1, 2, \ldots, p. \tag{13}$$

Note that if function $\bar{F}$ is simply a summation then F reduces to the usual notion of partial separability.

Following the general form given in Figure 2, F can be computed with the following program,

$$\boxed{\begin{array}{l} \textit{for } i = 1:p \\ \qquad \text{“}\textit{Solve}\text{” } \textit{for } y_i\textit{: } y_i - T_i(x) \;=\; 0 \\ \text{“}\textit{Solve}\text{” } \textit{for } z : z - \bar{F}(y_1, y_2, \ldots, y_p) \;=\; 0 \end{array}}$$

Of course this program can be inefficient if some of the functions T_i share common sub-expressions. Therefore a more general program can be written if we define a “stacked” vector $Y^T = (y_1^T, \ldots y_p^T)$ and a corresponding vector function

$$\tilde{F}(x) = \begin{pmatrix} T_1(x) \\ T_2(x) \\ \vdots \\ T_p(x) \end{pmatrix}.$$

This yields the simple 2-liner:

$$\boxed{\begin{array}{l} \text{“}\textit{Solve}\text{” } \textit{for } Y: \;\; Y - \tilde{F}(x) = 0 \\ \text{“}\textit{Solve}\text{” } \textit{for } z: \;\; z \;=\; \bar{F}(y_1, y_2, \ldots, y_p). \end{array}}$$

The general extended system (9) reduces to

$$(14)\qquad \left(\begin{array}{c|cccc} -\tilde{J}_1 & I & & & \\ -\tilde{J}_2 & & I & & \\ \vdots & & & \ddots & \\ -\tilde{J}_p & & & & I \\ \hline 0 & \bar{J}_1 & \bar{J}_2 & \cdots & \bar{J}_p \end{array}\right) \begin{pmatrix} \delta x \\ \delta y_1 \\ \delta y_2 \\ \vdots \\ \delta y_p \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ -F \end{pmatrix}$$

where $\bar{J}$ is broken into different components $\bar{J}_i$ corresponding to variables y_i. If F is partially separable in the usual fashion, i.e., function $\bar{F}$ is a summation, then each matrix $\bar{J}_i$ is an identity matrix. It is clear that the computation of J_E will, in general, be considerably more economical than than the straightforward application of AD to detemine $J = \sum_{i=1:p} \tilde{J}_i \cdot \bar{J}_i$ via AD. For example, Jacobians $\tilde{J}$, $\bar{J}$ can be determined by applying the bi-coloring technique [9] to $\tilde{F}$ and $\bar{F}$ respectively. And of course if $\bar{F}$ is simple enough, $\bar{J}$ will be constant (and trivially available).

If the function $\bar{F}$ is itself non-trivial it may be advantageous to exploit this by expanding the 2-line program. For example, let I_j be an index set indentifying a subset of $\{1, \ldots p\}$, $j = 1, \ldots, t$. Further, let $\hat{F}_j$ be a vector function, $j = 1, \ldots, t$ and suppose $\bar{F}(Y) = \sum_{j=1}^{t} \hat{F}_j(\sum_{I_j} y_j)$. So the program to evaluate $\bar{F}(Y)$ can be written

$$\boxed{\begin{array}{l} \textit{for } j = 1:t \\ \qquad w_j \;=\; \sum_{I_j} y_j \\ \qquad v_j \;=\; \hat{F}_j(w_j) \\ z = \sum_{j=1}^{t} v_j \end{array}}$$

Clearly the evaluation of this "group partially separable" function is easily expressed in the lower Hessenberg form given in Figure 2; the corresponding Jacobian matrix J_E is likely to be sparse and economically computable via the AD techniques in [9]

4.2.1 Product function A special case of the generalized partially separable form is a product function: Suppose that $F : \Re^n \rightarrow \Re^n$ is a component-wise product of functions:

$$F(x) = F_1(x). * F_2(x). * \;\ldots\; . * F_p(x),$$

where $F_i : \Re^n \rightarrow \Re^n$, $i = 1 : n$, and the notation ".*" indicates component-wise multiplication (following the MATLAB convention). A natural way to evaluate F at an argument x is to execute the program,

$$\boxed{\begin{array}{l} \textit{for } i = 1:p \\ \qquad \textit{"Solve" for } y_i\textit{:}\; y_i - F_i(x) \;=\; 0 \\ \textit{"Solve" for } z : z \;-\; y_1. * y_2. * \;\ldots\; . * y_p \;=\; 0 \end{array}}$$

The extended Newton system (9) simplifies to:

$$
(15) \qquad \left(\begin{array}{c|cccc}
-J_1 & I & & & \\
-J_2 & & I & & \\
\vdots & & & \ddots & \\
-J_p & & & & I \\
\hline
0 & D_1 & D_2 & \cdots & D_p
\end{array}\right)
\begin{pmatrix} \delta x \\ \delta y_1 \\ \delta y_2 \\ \vdots \\ \delta y_p \end{pmatrix}
=
\begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ -F \end{pmatrix}
$$

where D_i is a diagonal matrix, $D_i = \prod_{j=1, j\neq i}^{p} diag(y_i)$, and "$diag(y_i)$" is the diagonal matrix with the vector y_i defining the diagonal in the natural order. Note that the Jacobian of F reduces to

$$J = \sum J_i(x) \cdot D_i.$$

Clearly it is entirely possible that J is relatively dense even when each component Jacobian J_i is very sparse; hence, direct determination of J via AD is usually unattractive compared to the calculation of the extended Jacobian, J_E, via AD (e.g., using the bi-coloring approach [9]).

4.3 A remark

Our two main examples, composite functions and generalized partially separable functions, are complementary in a structural sense. The evaluation of a composite function is a *depth* computation: each subsequent intermediate variable y_i depends on the previous intermediate y_{i-1}. The computational graph of a composite function is a long chain: See Figure 3.

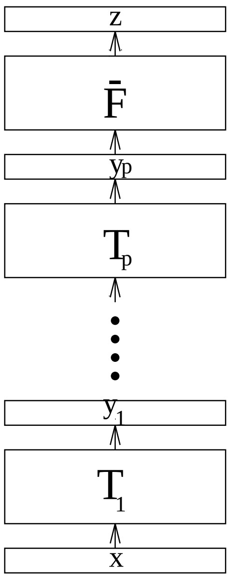

FIG. 3. *Composite Function*

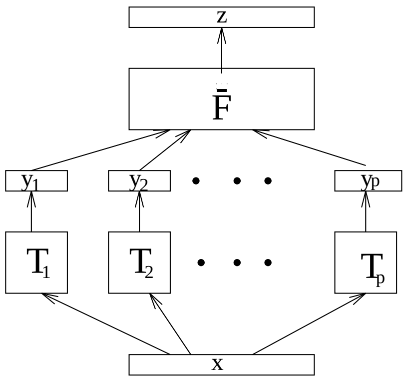

FIG. 4. *Generalized Partially Separable Function*

In contrast, generalized partially separable functions are primarily breadth computations: typically, the intermediate variables y_i are relatively independent of each other: it is the final computation $z = \bar{F}(y_1, y_2, \ldots, y_p)$ that binds the intermediates together. The computational graph is short and fat; see Figure 4.

Despite such grossly different structures the lower Hessenberg format, illustrated in Figure 2, is applicable in both cases: this view represents a convenient way to program the evaluation of F and allows for the efficient application of AD tools to determine the Jacobian matrix and/or the Newton step.

5 Conclusions

This paper is concerned with the efficient determination of Jacobian matrices and/or Newton steps of large systems of nonlinear equations using automatic differentiation. We review a recent graph-coloring technique to enable the efficient determination of *sparse* Jacobian matrices. However, a major point of this paper is that many large-scale problems exhibit dense but *structured* Jacobian matrices. We show how to exploit structure, in a general way, to allow for the efficient application of AD tools.

Our view is that the evaluation of a large-scale function is often naturally programmed in a lower Hessenberg form F_E, illustrated in Figure 2. The "extended" function F_E involves a set of intermediate vectors $\{y_i\}$ yielding a Jacobian matrix J_E with respect to both the original variables and the set of internal vectors $\{y_i\}$. Often, it is considerably more economical to compute J_E and/or the related extended Newton step as opposed to direct determination of J via AD. This is our proposal.

As our examples indicate the selection of intermediate vectors y_i is often a natural product of stating a natural coarse-level program to evaluate F. Typically this lower Hessenberg structure is available to the user. In this situation we propose that a *pre-compiler*, software to exploit this general structure and enable the "surgical" use of AD software, would serve a useful role.

In principle such a pre-compiler can be built based on a lower-Hessenberg high-level program to evaluate F. In many cases this is practical. However, an interesting question is, given an arbitrary (but correct!) program to evaluate F, is it possible to automatically recover the lower Hessenberg form F_E? Of course a fine-grained lower Hessenberg structure is always available from the compiled program [11]; however, we are concerned with the natural *high-level* (coarse-grained) lower-Hessenberg form. This is an open question.

References

[1] B. M. Averick, J. J. Moré, C. H. Bischof, A. Carle, and A. Griewank, *Computing large sparse Jacobian matrices using automatic differentiation*, SIAM Journal on Scientific Computing, 15 (1994), pp. 285–294.

[2] C. H. Bischof, A. Bouaricha, P. M. Khademi, and J. J. Moré, *Computing gradients in large-scale optimization using automatic differentiation*, Tech. Rep. MCS-P488-0195, MCS, June 1995.

[3] C. Broyden, *A class of methods for solving nonlinear simultaneous equations*, Mathematics of Computation, 19 (1965), pp. 577–593.

[4] T. F. Coleman and J.-Y. Cai, *The cyclic coloring problem and estimation of sparse Hessian matrices*, SIAM J. Alg. Disc. Meth., 7 (1986), pp. 221–235.

[5] T. F. Coleman, B. S. Garbow, and J. J. Moré, *Software for estimating sparse Jacobian matrices*, ACM Trans. Math. Software, 10 (1984), pp. 329–345.

[6] ——, *Software for estimating sparse Hessian matrices*, ACM Trans. Math. Software, 11 (1985), pp. 363–377.

[7] T. F. Coleman and J. J. Moré, *Estimation of sparse Hessian matrices and graph coloring problems*, Math. Programming, 28 (1984), pp. 243–270.

[8] ——, *Estimation of sparse Jacobian matrices and graph coloring problems*, SIAM J. on Numerical Analysis, 20 (1984), pp. 187–209.

[9] T. F. Coleman and A. Verma, *The efficient computation of sparse Jacobian matrices using automatic differentiation*, Tech. Rep. TR95-1557, Computer Science Department, Cornell

University, November, 1995.
[10] A. R. Curtis, M. J. D. Powell, and J. K. Reid, *On the estimation of sparse Jacobian matrices*, J. Inst. Math. Appl., 13 (1974), pp. 117–119.
[11] A. Griewank, *Direct calculation of Newton steps without accumulating Jacobians*, in Large-Scale Numerical Optimization, T. F. Coleman and Y. Li, eds., SIAM, Philadelphia, Penn., 1990, pp. 115–137. Also appeared as Preprint MCS–P132–0290, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., February 1990.
[12] ———, *Some Bounds on the Complexity of Gradients, Jacobians, and Hessians*, in Complexity in Nonlinear Optimization, P. Pardalos, ed., World Scientific Publishers, 1993, pp. 128–161.
[13] A. K. M. Hossain and T. Steihaug, *Computing a sparse Jacobian matrix by rows and columns*, Tech. Rep. 109, Department of Informatics, University of Bergen, June 1995.