Software for Estimating Sparse Jacobian Matrices

THOMAS F. COLEMAN
Cornell University
BURTON S. GARBOW
and
JORGE J. MORÉ
Argonne National Laboratory

In many nonlinear problems it is necessary to estimate the Jacobian matrix of a nonlinear mapping F. In large-scale problems the Jacobian of F is usually sparse, and then estimation by differences is attractive because the number of differences can be small compared with the dimension of the problem. For example, if the Jacobian matrix is banded, then the number of differences needed to estimate the Jacobian matrix is, at most, the width of the band. In this paper we describe a set of subroutines whose purpose is to estimate the Jacobian matrix of a mapping F with the least possible number of function evaluations.

Categories and Subject Descriptors: E.1 [**Data**]: Data Structures—*graphs*; E.2 [**Data**]: Data Storage Representation—*linked representations*; G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*sparse and very large systems*; G.1.5 [**Numerical Analysis**]: Roots of Nonlinear Equations—*systems of equations*; G.1.6 [**Numerical Analysis**]: Optimization—*least squares methods*

General Terms: Algorithms

Additional Key Words and Phrases: Numerical differentiation, Jacobian matrix, large sparse optimization, nonlinear problems, graph coloring

The Algorithm: FORTRAN Subroutines for Estimating Sparse Jacobian Matrices. *ACM Trans. Math. Softw. 10*, 3 (Sept. 1984), 346–347.

1. INTRODUCTION

In many nonlinear problems it is necessary to estimate the Jacobian matrix of a mapping F: $R^n \rightarrow R^m$. In large-scale problems the Jacobian $F'(x)$ is usually sparse, and then estimation by differences is attractive because the number of differences can be small compared with the dimension of the problem. For example, if the Jacobian matrix is banded, then the number of differences needed to estimate the Jacobian matrix is, at most, the width of the band. In this paper

T. F. Coleman's work was supported in part by the Applied Mathematical Sciences Research Program (KC-04-02) of the Office of Energy Research of the U.S. Department of Energy under Contract DE-AC02-83ER13069.

B. S. Garbow and J. J. Moré's work was supported in part by the Applied Mathematical Sciences Research Program (KC-04-02) of the Office of Energy Research of the U.S. Department of Energy under Contract W-31-109-Eng-38.

Authors' addresses: B. S. Garbow, and J. J. Moré, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439; T. F. Coleman, Department of Computer Science, Cornell University, Ithaca, NY 14853.

we describe a set of subroutines whose purpose is to estimate the Jacobian matrix of a mapping $F: R^n \rightarrow R^m$ with the least possible number of function evaluations.

The problem of estimating a sparse Jacobian matrix can be phrased in the following terms: Given a sparse m by n matrix A, obtain vectors $d_1, d_2, \ldots, d_p$ such that $Ad_1, Ad_2, \ldots, Ad_p$ determine A uniquely. In this formulation, A is associated with the Jacobian matrix $F'(x)$ and the product Ad is associated with an estimate of $F'(x)\ d$. Typically, the estimate of $F'(x)\ d$ is obtained by the forward difference

$$F(x+d) - F(x) = F'(x)d + o(\| d \|),$$

or the central difference

$$\tfrac{1}{2}[F(x+d) - F(x-d)] = F'(x)d + o(\| d \|^2)$$

approximations. Thus each evaluation of Ad requires at least one function evaluation.

Our algorithms for determining a matrix A are based on the observation of Curtis, Powell, and Reid [4] that a group of columns can be determined with an evaluation of Ad if no two columns in this group have a nonzero in the same row position. To establish this claim, let $a_1, \ldots, a_n$ be the columns of A, and let $\{a_j : a_j \in C\}$ be a group of columns such that no two columns in this group have a nonzero in the same row position. If $d \in R^n$ is a vector with components $\delta_j \neq 0$ if a_j belongs to C and $\delta_j = 0$ otherwise, then

$$Ad = \sum_{j \in C} \delta_j a_j,$$

and since no two columns in C have a nonzero in the same row position, for each nonzero a_{ij} with $j \in C$ we have

$$(Ad)_i = \delta_j a_{ij}.$$

In view of this observation, it is possible to determine an m by n matrix A if we partition the columns of A into groups $C_1, \ldots, C_p$ so that each column belongs to one and only one group, and so that no two columns in a group have a nonzero in the same row position. A partition of the columns of A with this property is *consistent* with the determination of A.

In the CPR algorithm as proposed by Curtis, Powell, and Reid [4], the groups $C_1, \ldots, C_p$ are formed one at a time by scanning the columns in the order $a_1, a_2, \ldots, a_n$, and by including a column in the current group if it has not been included in a previous group and if it does not have a nonzero in the same row position as another column already in the group. By looking at the problem from a graph theory point of view, Coleman and Moré [2] showed that it is possible to improve the CPR algorithm by scanning the columns in a carefully selected order. Various orderings were considered and analyzed by Coleman and Moré [2]; One of our purposes here is to describe the implementation of the resulting algorithms.

Many users will only be interested in subroutines DSM and FDJS. These are the interface routines for the package, and with these two subroutines it is quite easy to estimate the Jacobian matrix of a mapping $F: R^n \rightarrow R^m$. An example illustrating the use of DSM and FDJS appears in Section 4.

Given the sparsity pattern of an m by n matrix A, subroutine DSM determines a consistent partition of the columns of A. The consistent partition is specified by an array *ngrp* of length n by setting *ngrp*(*jcol*) to the group number of column *jcol*. Subroutine DSM is an interface routine for the ordering algorithms and is quite easy to use; additional details can be found in Section 2.

Given a consistent partition of the columns of the Jacobian matrix, subroutine FDJS determines an approximation to those columns in a given group of the partition. The entire Jacobian matrix can be determined by calling FDJS for each group in the partition. Subroutine FDJS stores the Jacobian matrix with either a column-oriented or a row-oriented definition of the sparsity pattern. If the user is storing the Jacobian matrix with a different data structure, it is necessary to modify FDJS or to provide an interface between the two data structures. To facilitate this, the coding of FDJS is described in Section 3.

An example illustrating the use of subroutines DSM and FDJS is provided in Section 4. This example also serves as a test program for our package. Section 5 provides an overview of the subroutines included in the package and a description of the transition from the data structure used by DSM to the data structure used by the algorithms called by DSM. Implementation details and an analysis of the running time of the algorithms used by DSM appear in Section 6. It is only in this section that we need a modest amount of graph theory.

Section 7 contains some of the numerical results that we have obtained with subroutine DSM. These results were obtained with the 30 sparsity patterns of Everstine [5] and show that on these problems DSM outperforms the CPR algorithm, always obtaining an optimal or nearly optimal partition of the columns of A. Optimality can sometimes be recognized because DSM determines a lower bound *mingrp* on the number of groups possible in any consistent partition. DSM was optimal on 19 of the Everstine problems, and never required more than *mingrp* + 2 groups. In contrast, CPR was optimal on only 6 problems and required at least *mingrp* + 3 groups on 12 problems. CPR never obtained a better partition than DSM.

2. SUBROUTINE DSM

Given the sparsity pattern of an m by n matrix A, subroutine DSM determines an optimal or nearly optimal consistent partition of the columns of A.

The user specifies a definition of the sparsity pattern of A by providing the ordered pairs (i, j) for which $a_{ij} \neq 0$

$$(\mathit{indrow}(k), \mathit{indcol}(k)), \qquad k = 1, 2, \ldots, \mathit{npairs}. \tag{2.1}$$

These pairs can be provided in any order. Moreover, duplicate pairs are allowed so that *npairs* need not agree with the number of nonzeros in A.

On output DSM defines a consistent partition of the columns of A via the integer array *ngrp* by setting *ngrp*(*jcol*) to the group number of column *jcol*. In addition, the variable *mingrp* provides a lower bound on the number of groups possible in a consistent partition, and the variable *maxgrp* is the number of groups in the partition obtained by DSM. On output, DSM also transforms the specification of the sparsity pattern (2.1), provided by *indrow* and *indcol*, into an alternative specification that is more appropriate for the algorithms used by DSM. The original data are effectively preserved because this alternative specification allows the user to recover the ordered pairs (i, j) for which $a_{ij} \neq 0$. Details are provided at the end of this section.

A lower bound on the number of groups in a consistent partition is $\rho_{\max}$ where $\rho_{\max}$ is the maximum number of nonzero elements in any row of A. Usually *mingrp* is set to $\rho_{\max}$, but on some problems *mingrp* may exceed $\rho_{\max}$. For example, if

$$A = \begin{pmatrix} \times & \times & \\ \times & & \times \\ & \times & \times \end{pmatrix},$$

then *mingrp* is set to 3. Our experience on practical problems has been that DSM requires at most three groups more than the bound specified by *mingrp*. For many problems *maxgrp* agrees with *mingrp* and then DSM is optimal.

Execution times for subroutine DSM are quite satisfactory since the number of operations required by one call is proportional to

$$\sum_{i=1}^{m} \rho_i^2, \tag{2.2}$$

where ρ_i is the number of nonzeros in the ith row of A. This bound is appropriate because many sparse matrix computations require at least (2.2) operations. For example, the number of operations needed to compute $A^{\mathrm{T}}A$ is at least a constant multiple of (2.2).

The claim that (2.2) is a measure of the running time of DSM assumes that *npairs*, *m*, and *n* are not more than a constant times (2.2). This is certainly the case in any nontrivial situation since (2.2) is not less than the number of nonzero elements of A.

An impression of the overhead required by DSM can be obtained by noting that the number of operations needed to evaluate the Jacobian matrix by differences is on the order of (2.2) when the mapping F is linear. Indeed, in this case ρ_i operations are needed to evaluate the ith component of F, and thus the number of operations needed to approximate the Jacobian matrix by differences is at least

$$\rho_{\max} \sum_{i=1}^{m} \rho_i. \tag{2.3}$$

It should be clear that (2.2) is bounded above by (2.3). If F is a nonlinear mapping, then estimation of the Jacobian matrix is likely to require considerably more operations than (2.3). Moreover, in a typical nonlinear problem DSM will only be called once, whereas it will be necessary to estimate the Jacobian matrix many times. These arguments further support our claim that the execution times for DSM are quite satisfactory.

Implementation of DSM so that the execution time is proportional to (2.2) requires an appropriate data structure. The ordered pairs (i, j) for which $a_{ij} \neq 0$ is a convenient data structure for the user, but DSM requires a different data

```
      do 10 jcol = 1, n
         d(jcol) = zero
         if (ngrp(jcol) .eq. numgrp) d(jcol) = eta(jcol)
10       continue
```

Program 3.1

structure. The algorithms called by DSM require both column-oriented and row-oriented definitions of the sparsity pattern. The arrays *indrow* and *jpntr* provide a column-oriented definition of the sparsity pattern if the row indices for the nonzero elements of the jth column are

$$indrow(k), \qquad k = jpntr(j), \ldots, jpntr(j+1) - 1.$$

The arrays *indcol* and *ipntr* provide a row-oriented definition of the sparsity pattern if the column indices for the nonzero elements of the ith row are

$$indcol(k), \qquad k = ipntr(i), \ldots, ipntr(i+1) - 1.$$

Given the ordered pairs (2.1) for which $a_{ij} \neq 0$, subroutine DSM generates column-oriented and row-oriented definitions of the sparsity pattern. The transition from (2.1) is not difficult and is described in more detail in Section 5.

3. SUBROUTINE FDJS

Given a consistent partition of the columns of the Jacobian matrix, subroutine FDJS determines an approximation to those columns in a given group of the partition.

An approximation to the columns of the Jacobian matrix in group *numgrp* can be obtained by specifying a difference parameter array d with $d(jcol)$ nonzero if and only if *jcol* is a column in group *numgrp*, and an approximation to $F'(x)d$ in the array *fjacd*. The approximation to the columns of the Jacobian matrix in group *numgrp* is stored in the array *fjac*. Subroutine FDJS stores the Jacobian matrix in *fjac* with either a column-oriented or a row-oriented definition of the sparsity pattern. If the user is storing the Jacobian matrix with a different data structure, it is then necessary to modify FDJS or to provide an interface between the two data structures.

If the consistent partition is specified by an array *ngrp* by setting *ngrp*(*jcol*) to the group number of column *jcol*, then the user can define the difference parameter array d with the section of code in Program 3.1. The array *eta* contains the difference parameters used to estimate the Jacobian matrix. The user must provide suitable values for this array. Curtis and Reid [3], and Gill, Murray, and Wright [6], for example, discuss techniques for choosing the difference parameters.

The user must also provide an estimate for $F'(x)d$ in *fjacd*. For example, the estimate

$$F(x - d) - F(x)$$

corresponds to the forward difference formula, and the estimate

$$\tfrac{1}{2}[F(x + d) - F(x - d)]$$

corresponds to the central difference formula. It might have been simpler to have the user provide d and a subroutine *fcn* to evaluate $F(x)$. This, however, would force the user to transfer to *fcn* all the information needed to evaluate $F(x)$. For large-scale problems this can be a serious disadvantage.

Given d and *fjacd*, it is then possible to determine all the elements in the columns of the Jacobian matrix in group *numgrp* with a call to subroutine FDJS:

call fdjs(m, n, col, ind, npntr, ngrp, numgrp, d, fjacd, fjac)

The method used to store the Jacobian matrix is specified by the logical parameter *col*. If *col* is true, then the Jacobian matrix is stored with a column-oriented definition ($ind \equiv indrow$ and $npntr \equiv jpntr$) of the sparsity pattern; the nonzero elements of column j are then

$$fjac(k), \qquad k = npntr(j), \ldots, npntr(j+1) - 1.$$

If row-oriented storage ($ind \equiv indcol$ and $npntr \equiv ipntr$) is desired, set *col* to *false*; the nonzero elements of row i are then

$$fjac(k), \qquad k = npntr(i), \ldots, npntr(i+1) - 1.$$

An example of the use of FDJS can be found in the next section.

4. EXAMPLE

The use of subroutines DSM and FDJS can be illustrated by considering the problem of approximating the Jacobian matrix $F'(x)$ of a mapping $F\colon R^n \to R^n$ such that $F'(x)$ has a sparsity pattern of the form

$$\begin{pmatrix} T_1 & D_3 & \\ D_1 & T_2 & D_5 \\ D_2 & D_4 & B \end{pmatrix}, \tag{4.1}$$

where the T's have tridiagonal patterns, the D's have diagonal patterns, and B is of lower bidiagonal form. This is a simplified form of the neutron kinetics problem described by Carver and MacEwen [1].

A consistent partition of the columns of (4.1) can be determined with a call to DSM:

call dsm (m, n, nnz, indrow, indcol, ngrp, maxgrp, mingrp,
info, ipntr, jpntr, iwa, liwa).

We are mainly interested in the first eight parameters of the calling sequence. The parameters *nnz*, *indrow*, and *indcol* define the sparsity pattern of (4.1). These parameters can be determined with the section of code in Program 4.1, where it is assumed that each of the submatrices in (4.1) is of order l so that $m = n = 3l$, and *nnz* denotes the number of nonzero elements in (4.1).

Table I provides the output values of *mingrp* and *maxgrp* produced by DSM. These results show that DSM requires six groups to determine a matrix of the form (4.1) for each of the tested dimensions. Also note that *maxgrp* does not agree with *mingrp*. In some cases it is not possible to determine a matrix A with *mingrp* groups, but for (4.1) this is indeed the case. This can be shown by noting

```
      m = n
      l = n/3
      nnz = 0
      do 10 j = 1, n
         nnz = nnz + 1
         indrow(nnz) = j
         indcol(nnz) = j
         if (mod(j, l) .ne. 0) then
           nnz = nnz + 1
           indrow(nnz) = j + 1
           indcol(nnz) = j
           end if
         if (j .le. 2*l) then
           nnz = nnz + 1
           indrow(nnz) = j + l                Program 4.1
           indcol(nnz) = j
           if mod(j, l) .ne. 1) then
             nnz = nnz + 1
             indrow(nnz) = j - 1
             indcol(nnz) = j
             end if
           end if
         nnz = nnz + 1
         if (j .gt. l) then
           indrow(nnz) = j - l
         else
           indrow(nnz) = j + 2*l
           end if
         indcol(nnz) = j
10       continue
```

Table I. Output from DSM for the Neutron Kinetics Problem

n	*nnz*	*mingrp*	*maxgrp*	Time
300	1295	5	6	1.10
600	2595	5	6	2.18
900	3895	5	6	3.37
1200	5195	5	6	4.48

that a consistent partition of the columns of (4.1) is obtained if column j is assigned to group $ngrp(j)$ where

$$
\begin{aligned}
ngrp(j) &= \mathrm{mod}(j-1,5)+1, & 1 \le j \le l,\\
ngrp(j) &= \mathrm{mod}(j-l+1,5)+1, & l < j \le 2l,\\
ngrp(j) &= \mathrm{mod}(j-2l+3,5)+1, & 2l < j \le 3l.
\end{aligned}
$$

This example shows that for regular structures like (4.1) it is sometimes possible to improve on DSM. Finally, note that the execution time (measured in seconds

```
      subroutines fcn(n, x, indcol, ipntr, fvec)
      integer n
      integer indcol(*), ipntr(n + 1)
      real x(n), fvec(n)
c
c     Function subroutine for testing FDJS.
c
      integer i, ip
      real sum
      do 20 i = 1, n
         sum = 0.0
         do 10 ip = ipntr(i), ipntr(i + 1) - 1
            sum = sum + x(indcol(ip))
   10       continue
         sum = sum + x(i)
         fvec(i) = sum*(1.0 + sum) + 1.0
   20    continue
      return
      end
```

Program 4.2

on a VAX 11/780) grows linearly with n. This is to be expected since, for this problem, (2.2) is proportional to n.

The use of FDJS can be illustrated by considering the mapping F: $R^n \rightarrow R^n$ with components f_i: $R^n \rightarrow R$ defined by

$$f_i(x) = \varphi\left(\xi_i + \sum_{k \in S_i} \xi_k\right), \quad \varphi(\xi) = \xi(1 + \xi) + 1, \qquad (4.2)$$

where ξ_k is the kth component of x, and the set S_i represents the sparsity pattern of the ith row of the matrix (4.1). This is a simple function, but it serves quite well to illustrate the use of FDJS.

The subroutine in Program 4.2 evaluates F at x and returns $F(x)$ in the array *fvec*. In this program we make use of the fact that DSM returns in *indcol* and *ipntr* a row-oriented definition of the sparsity pattern.

We can now use FDJS to obtain an approximation to the Jacobian matrix of F. The code in Program 4.3 stores the approximation in the array *fjac* with a row-oriented definition of the sparsity pattern.

If a column-oriented definition is desired instead, then it is only necessary to set col to *true*, and change *indcol* to *indrow* and *ipntr* to *jpntr* in the call to *fdjs*. Also note that the difference parameters used in Program 4.3 are only for illustrative purposes; in general, the choice of difference parameters depends on the accuracy and nonlinearity of the problem function.

5. SUBROUTINES FOR ESTIMATING SPARSE JACOBIAN MATRICES

We have already described the interface subroutines DSM and FDJS in our package. In this section we provide a brief overview of the remainder of the package; implementation details for the sequential algorithm and the ordering algorithms are provided in Section 6.

```
      call fcn(n, x, indcol, ipntr, fvec)
      do 30 numgrp = 1, maxgrp
         do 10 j = 1, n
            d(j) = 0.0
            if (ngrp(j) .eq. numgrp) d(j) = 0.001
            xd(j) = x(j) + d(j)
10          continue
         call fcn(n, xd, indcol, ipntr, fjacd)
         do 20 i = 1, m
            fjacd(i) = fjacd(i) - fvec(i)
20          continue
         col = .false.
         call fdjs(m, n, col, indcol, ipntr, ngrp, numgrp, d, fjacd, fjac)
30       continue
```

Program 4.3

All of our algorithms for determining a consistent partition of the columns of an m by n matrix A use the sequential algorithm with some ordering of the columns of A. A consistent partition is obtained by first determining an ordering of the columns and then calling the sequential algorithm to obtain the consistent partition. Subroutine SEQ implements the sequential algorithm and the subroutines DEGR, IDO, and SLO determine orderings of the columns of A. In the remaining sections we describe these subroutines in detail.

Subroutine DSM obtains a consistent partition by calling the sequential algorithm with the ordering subroutines in the order SLO, IDO, and DEGR. All three ordering subroutines are used in an attempt to produce optimal or near-optimal results in all cases. If any of the orderings lead to a consistent partition with *mingrp* groups, DSM terminates at that point; otherwise DSM returns the best result obtained.

The transition from the data structure (2.1) to the column-oriented and row-oriented definitions of the sparsity pattern is accomplished by subroutines SRTDAT and SETR. Because this transition is not difficult, we describe these subroutines briefly.

Subroutine SRTDAT permutes *indrow* and *indcol* so that *indcol* is in nondecreasing order, and determines *jpntr* so that *indrow* and *jpntr* provide a column-oriented definition of the sparsity pattern. SRTDAT is a standard inplace sort. The execution time for SRTDAT is proportional to the number of input pairs in (2.1), so that if there are no duplicates, then the execution time is proportional to the number of nonzeros in A. After execution of SRTDAT it is easy to eliminate any duplicates among the input pairs (2.1), so we assume that this has been done.

Given a column-oriented definition of the sparsity pattern of a matrix A, subroutine SETR determines a row-oriented definition of the sparsity pattern. This is done by first determining the number of nonzeros in the rows of A, then setting pointers to the start of the rows in *indcol*, and finally filling *indcol*. It is straightforward to show that the execution time for SETR is proportional to the number of nonzeros in A.

For ease of reference, we next provide brief descriptions, in alphabetic sequence, of the subroutines in our package for estimating sparse Jacobian matrices. With the exception of NUMSRT all of the subroutines have been mentioned earlier; NUMSRT is a standard bucket sort.

Subroutine DEGR:	Given the sparsity pattern of an m by n matrix A, this subroutine determines the degree sequence for the graph $G(A)$.
Subroutine DSM:	This subroutine is a driver for determining an optimal or near-optimal consistent partition of the columns of an m by n matrix A.
Subroutine FDJS:	Given a consistent partition of the columns of an m by n Jacobian matrix, this subroutine computes approximations to those columns in a given group.
Subroutine IDO:	Given the sparsity pattern of an m by n matrix A, this subroutine determines the incidence degree ordering of the columns of A.
Subroutine NUMSRT:	Given a sequence of integers, this subroutine groups together those indices with the same sequence value and, optionally, sorts the sequence into either ascending or descending order.
Subroutine SEQ:	Given the sparsity pattern and an ordering of the columns of an m by n matrix A, this subroutine determines a consistent partition of the columns of A by a sequential algorithm.
Subroutine SETR:	Given a column-oriented definition of the sparsity pattern of an m by n matrix A, this subroutine determines a row-oriented definition of the sparsity pattern of A.
Subroutine SLO:	Given the sparsity pattern of an m by n matrix A, this subroutine determines the smallest-last ordering of the columns of A.
Subroutine SRTDAT:	Given the nonzero elements of an m by n matrix A in arbitrary order as specified by their row and column indices, this subroutine permutes these elements so that their column indices are in nondecreasing order.

6. IMPLEMENTATION DETAILS

The sequential algorithm and the ordering algorithms used by subroutine DSM have been described by Coleman and Moré [2]. Implementation of these algorithms is not straightforward, so we now describe these implementations and show, in particular, that these implementations execute in time proportional to (2.2).

The sequential algorithm and the ordering algorithms can be described best with the help of some graph theory terminology. A graph G is an ordered pair (V, E) where V is a finite and nonempty set of *vertices* and the *edges* E are unordered pairs of distinct vertices. The vertices u and v are *adjacent* if (u, v) is an edge with *endpoints u and v. The degree* of a vertex v is the number $\deg(v)$ of edges with v as an endpoint.

Given an ordering $v_1, v_2, \ldots, v_n$ of the vertices of a graph G, we can use a sequential algorithm to partition the vertices of G into groups such that vertices in a given group are not adjacent. At the kth stage of the *sequential algorithm* the group numbers $ngrp(v_1), \ldots, ngrp(v_{k-1})$ have been assigned; $ngrp(v_k)$ is then set to the smallest positive integer such that $ngrp(v_k) \neq ngrp(v_j)$ if (v_k, v_j) is an edge of G for some j with $1 \leq j < k$.

We are interested in the application of these concepts to a special class of graphs. Given an m by n matrix A with columns $a_1, a_2, \ldots, a_n$, we define a graph $G(A)$ with vertices $a_1, a_2, \ldots, a_n$ and edge (a_i, a_j) if and only if $i \neq j$ and columns i and j have a nonzero in the same row position. It should now be clear that the sequential algorithm on $G(A)$ generates a consistent partition of the columns of A, and that the purpose of an ordering is to minimize the number of groups required by the sequential algorithm.

The array *ngrp* defines a *coloring* of G in the sense that $ngrp(u) \neq ngrp(v)$ if u and v are adjacent. Thus the sequential algorithm can be viewed as a graph coloring algorithm. This is the point of view adopted by Coleman and Moré [2]; in this paper we deemphasize the graph-coloring viewpoint and instead prefer to work in terms of consistent partitions, since this concept is closer to the software. On the other hand, the graph-coloring viewpoint is important because the ordering algorithms only make sense when viewed as graph-coloring algorithms.

An ordering $v_1, v_2, \ldots, v_n$ of the vertices of a graph G is a *largest-first* ordering if $deg(v_1) \leq \cdots \leq deg(v_n)$. The descriptions of the other two ordering algorithms require additional graph theory terminology: Given a graph $G = (V, E)$ and a nonempty subset W of V, the *subgraph* $G[W]$ *induced* by W has vertex set W and edge set

$$\{(u, v) : (u, v) \in E \text{ and } u, v \in W\}.$$

In the *smallest-last* ordering the kth vertex v_k is determined after $v_{k+1}, \ldots, v_n$ have been selected by choosing v_k so that its degree in the subgraph induced by

$$V - \{v_{k+1}, \ldots, v_n\}$$

is minimal. In the *incidence degree* ordering v_k is determined after $v_1, \ldots, v_{k-1}$ have been selected by choosing v_k so that its degree in the subgraph induced by $\{v_1, \ldots, v_k\}$ is maximal. The incidence-degree of v_k is the degree of v_k in this subgraph.

The largest-first and smallest-last orderings are well known in the graph-coloring literature, but the incidence degree ordering was introduced by Coleman and Moré [2]. For a general graph G these algorithms can be implemented to run in time proportional to $|V| + |E|$ provided we are given the adjacency lists for the graph; that is, arrays $npntr(\cdot)$ and $nghbr(\cdot)$ such that the vertices adjacent to the jth vertex are

$$nghbr(k), \qquad k = npntr(j), \ldots, npntr(j+1) - 1.$$

See, for example, Matula and Beck [8]. However, the adjacency lists for $G(A)$ may require storage of order n^2 even if A is sparse, so this data structure is not appropriate. Our ordering algorithms use column-oriented and row-oriented definitions of the sparsity pattern of A and thus require only 2τ words of storage, where τ is the number of nonzero elements of A.

```
         tag(j) = n
         do 20 jp = jpntr(j), jpntr(j + 1) - 1
            i = indrow(jp)
            do 10 ip = ipntr(i), ipntr(i + 1) - 1
               adjcol = indcol(ip)
               if (tag(adjcol) .lt. j) then
                  tag(adjcol) = j
                  ndeg(adjcol) = ndeg(adjcol) + 1
                  ndeg(j) = ndeg(j) + 1
                  end if
10             continue
20          continue
```

Program 6.1

We now describe the ordering algorithms and the sequential algorithm. We only attempt to cover the important details and not complete descriptions of the algorithms.

6.1. Largest-First Ordering

The purpose of subroutine DEGR is to obtain the degree sequence for $G(A)$ and thus the largest-first ordering is determined by DEGR.

Algorithm. Degrees of G(A).

```
For j = 1, 2, ..., n
(a) Mark column j.
(b) Find all unmarked columns adjacent to column j and update the degrees of the
    unmarked columns and of column j.
```

An important part of the implementation of DEGR is a tagging scheme (Gustavson [7]) for efficiently updating the degrees of the unmarked columns adjacent to column j. Our implementation uses an array *ndeg* to record the degrees of the columns, and an array *tag* to mark columns and to update the degrees of the unmarked columns. Initially $ndeg(l) = 0$ and $tag(l) = 0$ for all columns l. We mark column j by setting $tag(j) = n$, and if *adjcol* is an unmarked column adjacent to column j, we set $tag(adjcol) = j$. Program 6.1 shows how the tagging scheme is used to determine the unmarked columns adjacent to column j and update the degrees of the unmarked columns and of column j.

The running time of DEGR can be analyzed by noting that the number of operations needed to execute Program 6.1 is proportional to

$$\sum_{a_{ij}\neq 0} \rho_i, \tag{6.1}$$

where ρ_i is the number of nonzero elements in the ith row. The total amount of work is thus proportional to

$$\sum_{j=1}^{n} \left(\sum_{a_{ij}\neq 0} \rho_i \right) = \sum_{i=1}^{m} \rho_i^2. \tag{6.2}$$

This shows that the time needed to execute DEGR is proportional to (2.2).

6.2. Smallest-Last and Incidence Degree Orderings

The implementations of the smallest-last and incidence degree orderings in subroutines SLO and IDO, respectively, are very similar. In the smallest-last ordering the column chosen at the kth stage has minimal degree in the graph induced by the unordered columns, whereas in the incidence degree ordering the chosen column has maximal incidence degree among the unordered columns. From this description it is clear that we need a data structure that permits the easy updating of the two types of degrees. A doubly linked list is a standard structure that satisfies this requirement.

We can implement a doubly linked list with the three arrays *head*, *prev*, and *next*. Each unordered column j is in a list of columns with the same degree. The first column in the list of columns with degree *deg* is *head*(*deg*) unless *head*(*deg*) = 0. In this case there are no columns in the *deg* list. The column before j in the degree list of column j is $prev(j)$ unless $prev(j) = 0$. In this case j is the first column in the degree list. The column after j in the degree list of column j is $next(j)$ unless $next(j) = 0$. In this case j is the last column in the degree list.

In the above description the term *degree* may refer either to the degree in the graph induced by the unordered columns or to the incidence degree for an unordered column. This permits us to discuss the smallest-last and incidence degree orderings at the same time. In the sequel we shall refer to these degrees as the degrees for the unordered columns.

Algorithm. Smallest-Last Ordering.

```
For $k = n, n - 1, \dots, 1$
(a) Choose a column $j$ of minimal degree and let $list(k) = j$.
(b) Delete column $j$ from the list of columns of minimal degree.
(c) Find all unordered columns adjacent to column $j$ and update the degree lists for the
    unordered columns.
```

The tagging scheme used is similar to the DEGR scheme. Initially $tag(l) = n$ for all l. Ordered columns are marked by setting $tag(j) = 0$, and if column *adjcol* is an unmarked column adjacent to column j, we then set $tag(adjcol) = k$. With this tagging scheme it is easy to update the degree lists for the unordered columns adjacent to column j.

The incidence degree ordering is quite similar to the smallest-last ordering. The main difference is that it is now necessary to update the incidence degrees.

Algorithm. Incidence Degree Ordering.

```
For $k = 1, 2, \dots, n$
(a) Choose a column $j$ of maximal incidence degree and let $list(k) = j$.
(b) Delete column $j$ from the list of columns of maximal incidence degree.
(c) Find all unordered columns adjacent to column $j$ and update the incidence degree lists
    for the unordered columns.
```

The tagging scheme used by this algorithm initially sets $tag(l) = 0$ for all l. Ordered columns are marked by setting $tag(j) = n$, and if column *adjcol* is an unmarked column adjacent to column j, we set $tag(adjcol) = k$.

In both ordering algorithms it is necessary to keep track of the degrees for the unordered columns. The array *list* can be used for this purpose provided we modify step (a) in both algorithms so that $list(j) = k$. Thus $list(j)$ is the degree of the jth column if j is an unordered column, whereas if j is ordered then *list*(j) is the position of column j in the order. If the array *list* is inverted at the end of the algorithm, then *list*(k) is the kth column in the ordering.

The argument used to analyze the running time of the largest-first ordering also applies to the smallest-last and incidence degree orderings because the number of operations needed to order column j is proportional to (6.1). Thus subroutines SLO and IDO execute in time proportional to (2.2).

A by-product of the smallest-last and incidence degree orderings is a lower bound *mingrp* on the number of groups needed by any consistent partition of the columns of A. This lower bound is obtained by determining a set of columns that are mutually adjacent in $G(A)$; in graph-theory terminology such a set is a *clique* of $G(A)$. The orderings can be used to determine a clique by noting that, if the kth column in either ordering has degree $k - 1$, then $G(A)$ has a clique of size k. Note that this property of the smallest-last and incidence degree orderings is not shared by the largest-first ordering. It is also possible to determine the size of a clique in $G(A)$ by computing $\rho_{\max}$, where $\rho_{\max}$ is the maximum number of nonzero elements in any row of A. This observation is based on the fact that if columns $j_1, \ldots, j_k$ have a nonzero in a given row then these columns form a clique in $G(A)$. Subroutine DSM sets *mingrp* to the size of the largest clique found by one of the techniques discussed above.

6.3. The Sequential Algorithm

In the sequential algorithm the order of the columns is specified by the array *list* by letting *list*(k) be the kth vertex in the ordering. On output from the sequential algorithm the array *ngrp* specifies a consistent partition of the columns of A by setting *ngrp*(j) to the group number of the jth column.

Algorithm. Sequential Algorithm.

```
For $k = 1, 2, \ldots, n$:
(a) Find all columns adjacent to column list($k$) and mark all the groups of the columns
    adjacent to column list($k$).
(b) Let ngrp(list($k$)) be the smallest unmarked group.
```

A tagging scheme is used in the sequential algorithm to determine the groups of the columns adjacent to column *list*(k). We initially set *tag*$(l) = 0$ for all groups l, and if column *adjcol* is adjacent to column j = *list*(k), we mark the group of *adjcol* by setting *tag*$(l) = k$ where l = *ngrp*(*adjcol*). It is then easy to determine the smallest unmarked group.

The number of operations needed to mark the groups of columns adjacent to column j = *list*(k) is proportional to (6.1), whereas the number of operations needed to find the smallest unmarked group is no more than $deg(a_j)$. Since

$$\sum_{j=1}^{n} deg(a_j) \le \sum_{i=1}^{m} \rho_i^2,$$

the running time of subroutine SEQ is proportional to (2.2).

7. NUMERICAL RESULTS

In Section 2 we discussed the overhead required by DSM and showed that the requirements of DSM are quite modest. In this section we present some numerical

Table II. Output from DSM for Naval Problems

n	*nnz*	*mingrp*	*maxgrp*
59	267	6	6
66	320	6	6
72	222	5	5
87	541	13	13
162	1182	9	10
193	3493	30	30
198	1392	12	12
209	1743	17	17
221	1629	12	12
234	834	10	10
245	1461	13	13
307	2523	9	11
310	2448	11	11
346	3226	19	20
361	2953	9	10
419	3563	14	15
492	3156	11	11
503	6027	25	25
512	3502	15	16
592	5104	15	15
607	5131	14	16
758	5994	11	11
869	7285	15	15
878	7448	10	11
918	7384	13	14
992	16744	18	18
1005	8621	27	27
1007	8575	10	11
1242	10426	12	14
2680	25026	19	19

results for DSM. A referee has pointed us to a set of Grenoble computer simulation problems from the Harwell-Boeing collection of sparse matrices. For three of these dimensions, 185, 512, and 1107, the differences between *maxgrp* and *mingrp* are, respectively, 5, 4, and 4. On all other practical problems, however, our experience has been that DSM requires at most three more groups than the bound specified by *mingrp*.

Table II shows the results of using DSM on the 30 sparsity patterns of the Everstine [5] collection. The patterns are for symmetric matrices with orders ranging from 59 to 2680. In addition to the order n of the matrix, Table II contains the number *nnz* of nonzeros in the matrix, and the output values for *mingrp* and *maxgrp*. Note that on 19 of the problems *maxgrp* agrees with *mingrp* and therefore DSM is optimal on these problems. DSM may still be optimal on the other problems because *mingrp* is always set to the size of a clique in $G(A)$, and it is possible for the largest clique in $G(A)$ to be less than the number of groups in an optimal consistent partition of the columns of A. For example, if A is a lower bidiagonal matrix of order n except for a nonzero in the $(1, n)$ position then the largest clique of $G(A)$ has size 2 but a consistent partition of the columns of A needs at least 3 groups if n is odd and $n \geq 5$.

We have tested DSM on all the sparsity patterns described by Coleman and Moré [2]. In particular, we have tested DSM on the nonsymmetric patterns obtained from the Everstine problems by considering the columns in the same order as on the tape provided by Everstine—it turns out that this leads to unsymmetric patterns. The results obtained on these unsymmetric patterns are similar to those in Table II. The only differences occur in the problems with dimensions 162, 307, 346, and 1242. The number of groups required is then 9, 12, 21, and 15, respectively. Thus DSM needs only two additional groups to determine the 30 unsymmetric patterns. This is not surprising because DSM is not strongly dependent on the ordering of the columns of the matrix. In contrast, CPR needs 15 more groups to determine the unsymmetric patterns.

We conclude this section by comparing our codes with those in Harwell's subroutine TD02A.

Given a column-oriented definition of the sparsity pattern, and a subroutine that evaluates the problem function, subroutine TD02A determines a consistent partition of the columns of the Jacobian matrix and computes an approximation to the Jacobian matrix. There are several differences between TD02A and our codes:

TD02A determines both a consistent partition and an approximation to the Jacobian matrix. In our codes DSM determines the consistent partition and FDJS determines the approximation to the Jacobian matrix.

TD02A uses a column-oriented definition of the sparsity pattern. DSM needs both a column-oriented and a row-oriented definition of the sparsity pattern.

TD02A uses the CPR algorithm to determine a consistent pattern. DSM uses the largest-first, smallest-last, and incidence degree orderings.

TD02A requires the user to provide a subroutine which evaluates the problem function. DSM uses reverse communication.

TD02A uses the ideas of Curtis and Reid [3] to automatically choose the difference parameters.

We have discussed throughout the paper the last three differences. We now turn to the first two differences.

One of the main reasons for having a separate subroutine for the calculation of a consistent partition of the columns of a sparse matrix is that we expect this calculation to have other applications. Also, this calculation requires a specialized data structure, and it is natural to keep separate the demands of this data structure.

TD02A needs only a column-oriented definition of the sparsity pattern because this data structure allows the efficient implementation of the sequential algorithm. However, the efficient implementation of the largest-first, smallest-last, and incidence degree orderings require, in addition, a row-oriented definition of the sparsity pattern. If only a column-oriented definition is provided, then determining these orderings would require order n^2 operations; for large scale problems, this cost would be prohibitive.

The length of the code for TD02A is shorter than for DSM/FDJS; the storage requirements of DSM/FDJS are no larger, however, because DSM and FDJS can share storage. TD02A needs $2nnz$ storage locations; nnz locations for the column-oriented definition of the storage pattern and nnz locations for the approximation to the Jacobian matrix. DSM requires $2nnz$ storage locations for the ordered pairs (i, j) for which $a_{ij} \neq 0$, but once DSM computes the partition it is necessary to retain only the n-element array *ngrp*. FDJS needs *nnz* locations for either a column-oriented or a row-oriented definition of the storage pattern and *nnz* locations for the approximation *fjac* to the Jacobian matrix, but the *nnz* locations for *fjac* can certainly be shared with DSM. If the storage of *fjac* is row-oriented, as in Program 4.3, then the statement

equivalence(*indrow*, *fjac*),

in the main program, allows *indrow* and *fjac* to share storage. If the storage of *fjac* is column-oriented, then the appropriate statement is

equivalence(*indcol*, *fjac*)

Finally, note that the storage requirements of DSM/FDJS are much less than, for example, the storage required by an algorithm for the *LU* decomposition of the approximation to the Jacobian matrix.

ACKNOWLEDGMENT

We are grateful to the referees for their suggestions and, in particular, for bringing to our attention the tagging scheme of Gustavson [7].

REFERENCES

1. Carver, M.B., and MacEwen, S.R. On the use of sparse matrix approximations to the Jacobian in integrating large sets of ordinary differential equations. *SIAM J. Sci. Stat. Comput. 2* (1981), 51–64.
2. Coleman, T.F., and Moré, J.J. Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM J. Numer. Anal. 20* (1983), 187–209.
3. Curtis, A.R., and Reid, J.K. The choice of step lengths when using differences to approximate Jacobian matrices. *J. Inst. Math. Appl. 13* (1974), 121–126.
4. Curtis, A.R., Powell, M.J.D., and Reid, J.K. On the estimation of sparse Jacobian matrices. *J. Inst. Math. Appl. 13* (1974), 117–119.
5. Everstine, G.C. A comparison of three resequencing algorithms for the reduction of matrix profile and wavefront. *Int. J. Numer. Methods Eng. 14* (1979), 837–853.
6. Gill, P.E., Murray, W., and Wright, M.H. *Practical Optimization.* Academic Press, New York, 1981, 339–345.
7. Gustavson, F.G. Finding the block lower triangular form of a sparse matrix. *Sparse Matrix Computations*, J.R. Bunch and D.J. Rose, Eds. Academic Press, New York, 1976, 275–289.
8. Matula, D.W., and Beck, L.L. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM 30* 3 (1983), 417–427.