SOFTWARE FOR ESTIMATING SPARSE JACOBIAN MATRICES[*]

Thomas F. Coleman[+]

Jorge J. More[++]

TR 82-502

June 1982

Department of Computer Science
Cornell University
Ithaca, New York 14853

[*]Also available as ANL-82-37, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439

[+]Department of Computer Science, Cornell University, Ithaca, New York 14853

[++]Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439

ABSTRACT

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 to 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.

Software for Estimating Sparse Jacobian Matrices

Thomas F. Coleman and Jorge J. Moré
Cornell University and Argonne National Laboratory

1. Introduction.

In many nonlinear problems it is necessary to estimate the Jacobian matrix of a mapping $F: R^n \to R^m$. 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 to 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 $F: R^n \to R^m$ 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 [1974] 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

Work 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.

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 [1974], 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 [1981] showed that it is possible to considerably improve the CPR algorithm by scanning the columns in a carefully selected order. Various orderings were considered and analyzed by Coleman and Moré [1981]; 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 $F : R^n \rightarrow R^m$ with a minimal or nearly minimal number of function evaluations. An example illustrating the uses of DSM and FDJS appears in Section 4.

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 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 is determined by calling FDJS for each group in the partition. Subroutine FDJS stores the Jacobian matrix with a column-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. For this reason the coding of FDJS is described in Section 3.

An example illustrating the uses of subroutines DSM and FDJS is provided in Section 4. This example also serves as a test program for our package. In Section 5 we provide 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 sparsity patterns of Everstine [1979] and show that on practical problems DSM and FDJS can estimate the Jacobian matrix of a mapping $F: R^n \to R^m$ with a minimal or nearly minimal number of function evaluations. All computations were done on a VAX 11/780.

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 $a_{ij} \neq 0$:

$$(2.1) \qquad (indrow(k), indcol(k)), \quad k = 1,2,\ldots,npairs\ .$$

These pairs can be provided in any order. Moreover, duplicate pairs are allowed so that *npairs* need not agree with the number of nonzeroes 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 which is more appropriate for the the algorithms used by DSM. This alternative specification is explained 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. In practice DSM needs, at worst, one or two more groups 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

$$(2.2)\qquad \sum_{i=1}^{m} \rho_i^2 ,$$

where ρ_i is the number of nonzeroes in the i-th 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 $A^T A$ 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 non-trivial 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 comparing DSM with a subroutine for computing the sparse LU factorization of a matrix. We used the factorization subroutine F01BRF from the NAG library because it is an excellent program. For test problems we used the 30 sparsity patterns in the Everstine [1979] collection with the nonzero entries of the matrix being uniformly distributed random numbers in (0,1). On these problems DSM was always faster than the factorization subroutine. Indeed, on twelve of the problems (in particular, on the 7 problems with $n \geq 878$) DSM was at least 10 times faster than the factorization subroutine. The storage requirements of DSM also compare very favorably because the factorization subroutine requires an unspecified amount of storage to handle the fill-in of the factorization in addition to the storage required for the matrix and the sparsity pattern. Finally, note that in a typical nonlinear problem DSM is only called once whereas the factorization subroutine is usually called many times.

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 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 nonzero elements of the j-th column are

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

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

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

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* are stored in the array *fjac*. Subroutine FDJS stores the Jacobian matrix in an array *fjac* with a column-oriented definition of the sparsity pattern; that is, the nonzero elements of the j-th column of the Jacobian matrix are stored in positions

$$fjac(k), \quad k = jpntr(j), \ldots, jpntr(j+1)-1 .$$

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 the user can define the difference parameter array d with the section of code in Program 3.1.

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

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. The user must also provide an estimate for $F'(x)d$ in an array *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.

Given d and *fjacd*, it is then possible to determine all the elements in the columns of the Jacobian matrix in group *numgrp*. The method for storing this information depends on the data structure used to store the Jacobian matrix; if the Jacobian matrix is stored with a column-oriented definition of the sparsity pattern then the section of code in Program 3.2 can be used.

```
      do 30 jcol = 1, n
         if (ngrp(jcol) .ne. numgrp) go to 20
         jpl = jpntr(jcol)
         jpu = jpntr(jcol+1) - 1
         if (jpu .lt. jpl) go to 20
         do 10 jp = jpl, jpu
            ir = indrow(jp)
            fjac(jp) = fjacd(ir)/d(jcol)
10          continue
20       continue
30    continue
```

Program 3.2

This code stores in the array *fjac* an approximation to the columns of the Jacobian matrix $F'(x)$ which are in group *numgrp*. To estimate the whole Jacobian it is necessary to execute Programs 3.1 and 3.2 for *numgrp* set to $1, 2, \ldots, maxgrp$. An example of the use of FDJS can be found in Section 4.

4. Example.

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

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

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 [1981].

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

```
      call dsm(n,n,nnz,indrow,indcol,ngrp,maxgrp,mingrp,
     *         info,ipntr,jpntr,iwa,liwa,bwa)
```

We are mainly interested in the first seven 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

$n = 3l$, and *nnz* denotes the number of nonzero elements in (4.1).

```
      l = n/3
      nnz = 0
      do 60 j = 1, n
         nnz = nnz + 1
         indrow(nnz) = j
         indcol(nnz) = j
         if (j .eq. l .or. j .eq. 2*l .or. j .eq. 3*l) go to 10
         nnz = nnz + 1
         indrow(nnz) = j + 1
         indcol(nnz) = j
10       continue
         if (j .eq. 1 .or. j .eq. l+1 .or. j .gt. 2*l) go to 20
         nnz = nnz + 1
         indrow(nnz) = j - 1
         indcol(nnz) = j
20       continue
         if (j .gt. 2*l) go to 30
         nnz = nnz + 1
         indrow(nnz) = j + l
         indcol(nnz) = j
30       continue
         if (j .le. l) go to 40
         nnz = nnz + 1
         indrow(nnz) = j - l
         indcol(nnz) = j
40       continue
         if (j .gt. l) go to 50
         nnz = nnz + 1
         indrow(nnz) = j + 2*l
         indcol(nnz) = j
50       continue
60       continue
```

Program 4.1

Table 4.1 provides the output values of *mingrp* and *maxgrp* produced by DSM. These results show that DSM requires 6 groups to determine a matrix of the form (4.1) for each of the tested dimensions.

n	nnz	mingrp	maxgrp	time
300	1295	5	6	1.28
600	2595	5	6	2.75
900	3895	5	6	3.80
1200	5195	5	6	4.93

Table 4.1. Output from DSM for the Neutron Kinetics Problem.

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

$$ngrp(j) = mod(j-1,5) + 1, \quad 1 \le j \le l,$$

$$ngrp(j) = mod(j-l+1,5) + 1, \quad l < j \le 2l,$$

$$ngrp(j) = mod(j-2l+3,5) + 1, \quad 2l < j \le 3l.$$

This example shows that for regular structures like (4.1) it is sometimes possible to improve on DSM. Finally, note that the execution time grows linearly with n. This is to be expected since it is proportional to (2.2).

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

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

where ξ_k is the k-th component of x, and the set S_i represents the sparsity pattern of the i-th 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.

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

Program 4.2

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* The difference parameters used in this code are only for illustrative purposes.

```
         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, n
               fjacd(i) = fjacd(i) - fvec(i)
20             continue
            call fdjs(n,indrow,jpntr,ngrp,numgrp,d,fjacd,fjac)
30          continue
```

Program 4.3

5. Subroutines for Estimating Sparse Jacobian Matrices.

We have already described the interface subroutines DSM and FDJS in our package. The purpose of this section is to provide a brief overview of the remainder of the package.

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 an ordering of the columns of A. In the remaining sections we shall 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 leads to a consistent partition with *mingrp* groups then 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. This is done by first determining the number of nonzeroes in the columns of A and setting pointers to the start of the columns in *indrow*. The sorting is accomplished by examining each component of *indcol*, and if the current component is not in position then it is placed in position and the displaced component is made the current component.

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 nonzeroes in A. After execution of SRTDAT it is easy to eliminate any duplicates in the input pairs (2.1), so we now 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 nonzeroes 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 nonzeroes in A.

For ease of reference, we next provide a brief description of the purposes 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 just a simple bucket sort.

Subroutine DEGR: Given the sparsity pattern of an m by n matrix A, this subroutine determines the degree sequence for the intersection graph of the columns of A.

Subroutine DSM: This subroutine determines an optimal or near-optimal consistent partition of the columns of a sparse m by n matrix A.

Subroutine FDJS: Given a consistent partition of the columns of an m by n Jacobian matrix into groups, 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 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 non-zero 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 non-decreasing order.

6. Implementation Details.

The sequential algorithm and the ordering algorithms used by subroutine DSM have been described by Coleman and Moré [1981]. Implementation of these algorithms is not straightforward, and thus we now describe the implementation of these algorithms and, in particular, show 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 non-empty 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 the vertices in a given group are not adjacent. At the k-th stage of the *sequential algorithm* the groups $ngrp(v_1), \ldots, ngrp(v_{k-1})$ have been assigned, and $ngrp(v_k)$ is 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 $1 \leq j < k$.

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. In graph theory terminology $G(A)$ in the *intersection graph* of the columns of the matrix A. 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 [1981]; in this paper we de-emphasize 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 $\{deg(v_j)\}$ is non-increasing. The description of the other two ordering algorithms require additional graph theory terminology: Given a graph $G = (V,E)$ and a non-empty 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 k-th 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 first two algorithms are well-known in the graph coloring literature, but the incidence degree ordering was introduced by Coleman and Moré [1981]. 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 j-th vertex are

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

See, for example, Matula and Beck [1981]. 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 the column-oriented and the 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.

We now describe the ordering algorithms and the sequential algorithm. We only attempt to cover the important details and not a complete description of the algorithms. The description of certain tasks is done in Fortran 77. For ease of portability, however, the package is written in Fortran 66.

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.

Program 6.1 determines an array *adj* such that *adj*(1) ,..., *adj*(*deg*) are the columns adjacent to column j and *deg* is the degree of column j. The array *mark* is used to mark column j and those columns adjacent to column j.

```
      mark(j) = .true.
      deg = 0
      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 (.not. mark(adjcol)) then
               deg = deg + 1
               adj(deg) = adjcol
               mark(adjcol) = .true.
               end if
10          continue
20       continue
```

Program 6.1

After the degree of column j is determined Program 6.2 un-marks the columns adjacent to column j and also unmarks column j.

```
      do 10 l = 1, deg
         mark(adj(l)) = .false.
10       continue
      mark(j) = .false.
```

Program 6.2

If we execute Programs 6.1 and 6.2 for $j = 1,2, \ldots, n$ then the degrees of $G(A)$ are obtained. The following algorithm uses the array *ngrp* to store the degree sequence of $G(A)$ by setting $ndeg(j)$ to the degree of column j.

Algorithm. Degrees of G(A).

For $j = 1,2, \ldots, n$

a) Find all columns adjacent to column j with Program 6.1.

b) Un-mark column j.

c) Un-mark the columns adjacent to column j.

d) Let $ndeg(j) = deg$.

The running time of this algorithm can be analyzed by noting that the number of operations needed to execute Programs 6.1 and 6.2 is proportional to

$$(6.1) \qquad \sum_{a_{ij} \neq 0} \rho_i + deg(a_j),$$

where ρ_i is the number of nonzero elements in the i-th row. The total amount of work needed to execute the programs for $j = 1,2, \ldots, n$ is thus proportional to

$$\sum_{j=1}^{n}\left[\sum_{a_{ij}\neq 0}\rho_i\right] + \sum_{j=1}^{n} deg(a_j) .$$

Since

$$\sum_{j=1}^{n}\left[\sum_{a_{ij}\neq 0}\rho_i\right] = \sum_{i=1}^{m}\rho_i^2 , \quad \sum_{j=1}^{n} deg(a_j) \le \sum_{i=1}^{m}\rho_i^2 ,$$

we have shown 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 are very similar. In the smallest-last ordering the column chosen at the k-th stage has maximal degree in the graph induced by the un-ordered columns, while in the incidence degree ordering the chosen column has maximal incidence degree among the un-ordered columns. From this description it is clear that we need a data structure which permits the easy updating of the two types of degrees. A doubly-linked list is a standard structure which satisfies this requirement.

We can implement a doubly-linked list with the three arrays *head*, *prev*, and *next*. Each un-ordered 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 un-ordered columns or to the incidence degree for an un-ordered 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 un-ordered columns.

It is easy to update the degree lists for the un-ordered columns. For example, if *numdeg* is the degree of column j then Program 6.3 deletes column j from the *numdeg* list and inserts it into the *numdeg* + 1 list.

We now have all the ingredients necessary to implement the smallest-last and incidence degree orderings.

```
c
c       Delete column j from the numdeg list.
c
        if (prev(j) .eq. 0) head(numdeg) = next(j)
        if (prev(j) .gt. 0) next(prev(j)) = next(j)
        if (next(j) .gt. 0) prev(next(j)) = prev(j)
c
c       Add column j to the numdeg + 1 list.
c
        prev(j) = 0
        next(j) = head(numdeg+1)
        if (head(numdeg+1) .gt. 0) prev(head(numdeg+1)) = j
        head(numdeg+1) = j
```

Program 6.3

Algorithm. Smallest-last ordering.

For $k = n, n-1, \ldots, 1$

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 un-marked columns adjacent to column j with Program 6.1.

d) Update the degree lists for the un-ordered columns.

e) Un-mark the columns adjacent to column j.

In this algorithm we must not un-mark column j. Thus, in step c the un-marked columns are precisely those columns which have not been assigned a place in the ordering. The incidence degree ordering is quite similar to the smallest-last ordering. The only difference is that it is now necessary to update the incidence degrees of the graph induced by the un-ordered columns.

Algorithm. Incidence degree ordering.

For $k = 1, 2, \ldots, n$

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 un-marked columns adjacent to column j with Program 6.1.

d) Update the incidence degree lists for the un-ordered columns.

e) Un-mark the columns adjacent to column j.

In both ordering algorithms it is necessary to keep track of the degrees for the un-ordered columns. An 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 j-th column if j is an un-ordered column, while 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 k-th 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 ordering because the number of operations needed to order column j is proportional to (6.1). Thus both the smallest-last and incidence degree orderings execute in time proportional to (2.2).

The implementation of the incidence degree ordering in subroutine IDO has a feature that is not present in the above algorithm. This additional feature can be motivated by noting that the ordering produced by IDO is arbitrary at certain stages. In particular, the choice of the first column is arbitrary because initially all columns have zero incidence degree. Also note that if $G(A)$ is disconnected then the choice of the first column in each connected component is also arbitrary because the maximal incidence degree is zero at the start of the ordering process for each component. For these reasons IDO attempts to choose a column of maximal degree among the columns of maximal incidence. This is done by searching the first

$$(6.2) \qquad \frac{1}{n}\sum_{i=1}^{m}\rho_i^2$$

elements in the list of columns of maximal incidence degree and choosing a column with the largest degree. Note that if the whole list of columns of maximal incidence is searched then IDO executes in time proportional to n^2 on certain problems, but that if the length of the search is bounded by (6.2) then IDO runs in time proportional to (2.2). On the other hand, by limiting the length of the search we may fail to find a column of largest degree. This last objection is handled by initially sorting the list of columns of maximal incidence degree in decreasing order according to their degrees. If this is done then the first element in the list of columns of maximal incidence has the largest degree whenever the maximal incidence is zero (recall that this happens at the start of the ordering process for each connected component); if the maximal incidence is greater than zero then the list of columns of maximal incidence is usually small, so (6.2) is not a severe restriction on the length of the search.

In the version of the incidence degree ordering used by Coleman and Moré [1981] the chosen column had the least number of nonzero elements among the columns of maximal incidence. This choice is often quite effective. For example, this choice produces optimal results in the neutron kinetics problem of Section 4. The version of the incidence degree ordering used in IDO, however, usually leads to slightly better results for DSM.

A by-product of the smallest-last and incidence degree orderings is the 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 k-th column in either ordering has degree $k - 1$ then $G(A)$ has a clique of size k. Note that this property is shared by the smallest-last and incidence degree orderings, but not 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 k-th 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 j-th column.

Algorithm. Sequential algorithm.

For $k = 1, 2, \ldots, n$

a) Find all columns adjacent to column j = *list*(k) with Program 6.1.

b) Un-mark the columns adjacent to column *list*(k).

c) Mark all the groups of the columns adjacent to column *list*(k).

d) Let *ngrp*(*list*(k)) be the smallest un-marked group.

e) Un-mark the groups.

The running time of this algorithm is proportional to (2.2) because the number of operations needed to determine the group of column *list*(k) is proportional to (6.1). It is possible to improve the running time of this algorithm by noting that we can mark the groups of columns adjacent to column *list*(k) as we determine the adjacent columns. Thus, in the subroutine SEQ steps a, b, and c are replaced by a section of code which marks the group number of columns which are adjacent to column *list*(k).

7. Numerical Results.

In Section 2 we have already discussed the overhead required by DSM and have shown that the requirements of DSM are quite modest. In this section we present evidence which shows that on practical problems DSM usually requires, at worst, one or two groups more than the bound specified by *mingrp*.

Table 7.1 shows the results of using DSM on the 30 sparsity patterns of the Everstine [1979] collection. These are symmetric patterns for matrices of order n with n ranging from 59 to 2680. In addition to the order n of the matrix, Table 7.1 contains the number *nnz* of nonzeroes 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 with a non-zero 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 $n \geq 3$.

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

Table 7.1 Output from DSM for Naval Problems.

Acknowledgment. The codes described in this paper have benefited considerably from Burt Garbow's contribution; we are very grateful for his help.

8. References.

Carver, M. B. and MacEwen, S. R. [1981]. On the use of sparse matrix approximations to the Jacobian in integrating large sets of ordinary differential equations, SIAM J. Sci. Stat. Comput. 2, 51-64.

Coleman, T. F. and Moré, J. J. [1981]. Estimation of sparse Jacobian matrices and graph coloring problems, Report ANL-81-39, Argonne National Laboratory, Argonne, Illinois.

Curtis, A. R., Powell, M. J. D. and Reid, J. K. [1974]. On the estimation of sparse Jacobian matrices, J. Inst. Math. Appl. 13, 117-119.

Everstine, G. C. [1979]. A comparison of three resequencing algorithms for the reduction of matrix profile and wavefront, Internat. J. Numer. Methods Engrg. 14, 837-853.

Matula, D. W. and Beck, L. L. [1981]. Smallest-last ordering and clustering and graph coloring algorithms, Report 8104, Department of Computer Science and Engineering, Southern Methodist University, Dallas, Texas.

APPENDIX

This appendix contains the listings for the test program and for the package for estimating sparse Jacobian matrices. We first list the test program and then list DSM and (in alphabetical order) the subroutines called by DSM: DEGR, IDO, NUMSRT, SEQ, SETR, SLO, SRTDAT. The last subroutine listed is FDJS. Note that this is a single precison subroutine; to obtain a double precision version it is only necessary to replace the single precision declaration by a double precision declaration.

```
c     **********
c
c     This is a test program for subroutines dsm and fdjs.
c     The test data represents a neutron kinetics problem.
c
c     **********
      integer i,info,ip,ipl,ipu,j,jp,jpl,jpu,k,l,liwa,
     *        maxgrp,maxrow,mingrp,minrow,n,nnz,numgrp,nwrite
      integer indcol(6000),indrow(6000),ipntr(1201),jpntr(1201),
     *        ngrp(1200),iwa(7200)
      logical bwa(1200)
      real dnsm,errij,errmax,fjact,sum
      real d(1200),fjac(6000),fjacd(1200),fvec(1200),x(1200),xd(1200)
c
c     Logical output unit is assumed to be number 6.
c
      data nwrite /6/
c
      liwa = 7200
c
c     Test for dsm and fdjs.
c
      write (nwrite,1000)
      do 150 n = 300, 1200, 300
         write (nwrite,2000)
c
c        Definition of sparsity pattern.
c
         l = n/3
         nnz = 0
         do 60 j = 1, n
            nnz = nnz + 1
            indrow(nnz) = j
            indcol(nnz) = j
            if (j .eq. l .or. j .eq. 2*l .or. j .eq. 3*l) go to 10
            nnz = nnz + 1
            indrow(nnz) = j + 1
            indcol(nnz) = j
   10       continue
            if (j .eq. 1 .or. j .eq. l+1 .or. j .gt. 2*l) go to 20
            nnz = nnz + 1
            indrow(nnz) = j - 1
            indcol(nnz) = j
   20       continue
            if (j .gt. 2*l) go to 30
            nnz = nnz + 1
            indrow(nnz) = j + l
            indcol(nnz) = j
   30       continue
            if (j .le. l) go to 40
            nnz = nnz + 1
            indrow(nnz) = j - l
            indcol(nnz) = j
   40       continue
```

```
            if (j .gt. l) go to 50
            nnz = nnz + 1
            indrow(nnz) = j + 2*l
            indcol(nnz) = j
   50       continue
   60    continue
c
c        Call dsm.
c
         call dsm(n,n,nnz,indrow,indcol,ngrp,maxgrp,mingrp,
     *            info,ipntr,jpntr,iwa,liwa,bwa)
         if (info .le. 0) write (nwrite,4000) info
c
c        Statistics for the matrix.
c
         maxrow = 0
         minrow = n
         do 70 l = 1, n
            maxrow = max0(maxrow,ipntr(l+1)-ipntr(l))
            minrow = min0(minrow,ipntr(l+1)-ipntr(l))
   70       continue
         dnsm = float(100*nnz)/float(n**2)
         write (nwrite,3000) n,nnz,dnsm,minrow,maxrow,mingrp,maxgrp
c
c        Test for fdjs.
c
         do 80 j = 1, n
            x(j) = float(j)/float(n)
   80       continue
         call fcn(n,x,indcol,ipntr,fvec)
c
c        Approximate the Jacobian matrix.
c
         do 110 numgrp = 1, maxgrp
            do 90 j = 1, n
               d(j) = 0.0
               if (ngrp(j) .eq. numgrp) d(j) = 0.001
               xd(j) = x(j) + d(j)
   90          continue
            call fcn(n,xd,indcol,ipntr,fjacd)
            do 100 i = 1, n
               fjacd(i) = fjacd(i) - fvec(i)
  100          continue
            call fdjs(n,indrow,jpntr,ngrp,numgrp,d,fjacd,fjac)
  110       continue
c
c        Test the approximation to the Jacobian.
c
         errmax = 0.0
         do 140 j = 1, n
            jpl = jpntr(j)
            jpu = jpntr(j+1) - 1
            do 130 jp = jpl, jpu
               i = indrow(jp)
```

```
            sum = 0.0
            ipl = ipntr(i)
            ipu = ipntr(i+1) - 1
            do 120 ip = ipl, ipu
               k = indcol(ip)
               sum = sum + x(k)
 120           continue
            sum = sum + x(i)
            fjact = 1.0 + 2.0*sum
            if (i .eq. j) fjact = 2.0*fjact
            errij = fjac(jp) - fjact
            if (fjact .gt. 0.0) errij = errij/fjact
            errmax = amax1(errmax,errij)
 130        continue
 140     continue
      write (6,5000) errmax
 150  continue
      stop
c
c     Format statements.
c
 1000 format(// ' Tests for dsm and fdjs - Neutron Kinetics Problem' //
     *       ' Statistics generated '//
     *       '        n - number of columns '/
     *       '      nnz - number of non-zero elements'/
     *       '     dnsm - matrix density (percentage)'/
     *       '   minrow - minimum number of non-zeros in any row'/
     *       '   maxrow - maximum number of non-zeros in any row'//)
 2000 format (// 3x,'n',6x,'nnz',5x,'dnsm',5x,
     *       'minrow',4x,'maxrow',4x,'mingrp',4x,'maxgrp'//)
 3000 format (2(i5,3x),f6.2,4x,4(i5,5x))
 4000 format (// ' *** mistake in input data, info is ***',i6)
 5000 format (// ' largest relative error of approximation is ',e10.2)
      end
```

```
      subroutine fcn(n,x,indcol,ipntr,fvec)
      integer n
      integer indcol(1),ipntr(1)
      real x(n),fvec(n)
c
c     Function subroutine for testing fdjs.
c
      integer i,ip,ipl,ipu,k
      real sum
      do 20 i = 1, n
         sum = 0.0
         ipl = ipntr(i)
         ipu = ipntr(i+1) - 1
         do 10 ip = ipl, ipu
            k = indcol(ip)
            sum = sum + x(k)
   10       continue
         sum = sum + x(i)
         fvec(i) = sum*(1.0 + sum) + 1.0
   20    continue
      return
c
c     Last card of subroutine fcn.
c
      end
```

```
      subroutine dsm(m,n,npairs,indrow,indcol,ngrp,[illegible]grp,mingrp,
     *               info,ipntr,jpntr,iwa,liwa,bwa)
      integer m,n,npairs,maxgrp,mingrp,info,liwa
      integer indrow(npairs),indcol(npairs),ngrp(n),
     *        ipntr(1),jpntr(1),iwa(liwa)
      logical bwa(n)
c     **********
c
c     subroutine dsm
c
c     The purpose of dsm is to determine an optimal or near-
c     optimal consistent partition of the columns of a sparse
c     m by n matrix A.
c
c     The sparsity pattern of the matrix A is specified by
c     the arrays indrow and indcol. On input the indices
c     for the non-zero elements of A are
c
c           indrow(k),indcol(k), k = 1,2,...,npairs.
c
c     The (indrow,indcol) pairs may be specified in any order.
c     Duplicate input pairs are permitted, but the subroutine
c     eliminates them.
c
c     The subroutine partitions the columns of A into groups
c     such that columns in the same group do not have a
c     non-zero in the same row position. A partition of the
c     columns of A with this property is consistent with the
c     direct determination of A.
c
c     The subroutine statement is
c
c       subroutine dsm(m,n,npairs,indrow,indcol,ngrp,maxgrp,mingrp,
c                      info,ipntr,jpntr,iwa,liwa,bwa)
c
c     where
c
c       m is a positive integer input variable set to the number
c         of rows of A.
c
c       n is a positive integer input variable set to the number
c         of columns of A.
c
c       npairs is a positive integer input variable set to the
c         number of (indrow,indcol) pairs used to describe the
c         sparsity pattern of A.
c
c       indrow is an integer array of length npairs. On input indrow
c         must contain the row indices of the non-zero elements of A.
c         On output indrow is permuted so that the corresponding
c         column indices are in non-decreasing order. The column
c         indices can be recovered from the array jpntr.
c
c       i[illegible]ol is an integer array of length npairs. On input indcol
```

```
c             must contain the column indices of the non-zero elements of
c             A. On output indcol is permuted so that the corresponding
c             row indices are in non-decreasing order. The row indices
c             can be recovered from the array ipntr.
c
c           ngrp is an integer output array of length n which specifies
c             the partition of the columns of A. Column jcol belongs
c             to group ngrp(jcol).
c
c           maxgrp is an integer output variable which specifies the
c             number of groups in the partition of the columns of A.
c
c           mingrp is an integer output variable which specifies a lower
c             bound for the number of groups in any consistent partition
c             of the columns of A.
c
c           info is an integer output variable set as follows. For
c             normal termination info = 1. If m, n, or npairs is not
c             positive or liwa is less than max(m,6*n), then info = 0.
c             If the k-th element of indrow is not an integer between
c             1 and m or the k-th element of indcol is not an integer
c             between 1 and n, then info = -k.
c
c           ipntr is an integer output array of length m + 1 which
c             specifies the locations of the column indices in indcol.
c             The column indices for row i are
c
c                   indcol(k), k = ipntr(i),...,ipntr(i+1)-1.
c
c             Note that ipntr(m+1)-1 is then the number of non-zero
c             elements of the matrix A.
c
c           jpntr is an integer output array of length n + 1 which
c             specifies the locations of the row indices in indrow.
c             The row indices for column j are
c
c                   indrow(k), k = jpntr(j),...,jpntr(j+1)-1.
c
c             Note that jpntr(n+1)-1 is then the number of non-zero
c             elements of the matrix A.
c
c           iwa is an integer work array of length liwa.
c
c           liwa is a positive integer input variable not less than
c             max(m,6*n).
c
c           bwa is a logical work array of length n.
c
c         Subprograms called
c
c           MINPACK-supplied ... degr,ido,numsrt,seq,setr,slo,srtdat
c
c           FORTRAN-supplied ... max0
c
```

```
c     Argonne National Laboratory. MINPACK Project. June 1982.
c     Thomas F. Coleman, Burton S. Garbow, Jorge J. More
c
c     **********
      integer i,ir,j,jp,jpl,jpu,k,maxclq,nnz,numgrp
c
c     Check the input data.
c
      info = 0
      if (m .lt. 1 .or. n .lt. 1 .or. npairs .lt. 1 .or.
     *    liwa .lt. max0(m,6*n)) go to 130
      do 10 k = 1, npairs
         info = -k
         if (indrow(k) .lt. 1 .or. indrow(k) .gt. m .or.
     *       indcol(k) .lt. 1 .or. indcol(k) .gt. n) go to 130
   10    continue
      info = 1
c
c     Sort the data structure by columns.
c
      call srtdat(n,npairs,indrow,indcol,jpntr,iwa(1))
c
c     Compress the data and determine the number of
c     non-zero elements of A.
c
      do 20 i = 1, m
         iwa(i) = 0
   20    continue
      nnz = 0
      do 70 j = 1, n
         jpl = jpntr(j)
         jpu = jpntr(j+1) - 1
         jpntr(j) = nnz + 1
         if (jpu .lt. jpl) go to 60
         do 40 jp = jpl, jpu
            ir = indrow(jp)
            if (iwa(ir) .ne. 0) go to 30
            nnz = nnz + 1
            indrow(nnz) = ir
            iwa(ir) = 1
   30       continue
   40       continue
         jpl = jpntr(j)
         do 50 jp = jpl, nnz
            ir = indrow(jp)
            iwa(ir) = 0
   50       continue
   60    continue
   70    continue
      jpntr(n+1) = nnz + 1
c
c     Extend the data structure to rows.
c
      call setr(m,n,indrow,jpntr,indcol,ipntr,iwa(1))
```

```
c
c     Determine a lower bound for the number of groups.
c
      mingrp = 0
      do 80 i = 1, m
         mingrp = max0(mingrp,ipntr(i+1)-ipntr(i))
   80    continue
c
c     Determine the degree sequence for the intersection
c     graph of the columns of A.
c
      call degr(n,indrow,jpntr,indcol,ipntr,iwa(5*n+1),iwa(n+1),bwa)
c
c     Color the intersection graph of the columns of A
c     with the smallest-last (SL) ordering.
c
      call slo(n,indrow,jpntr,indcol,ipntr,iwa(5*n+1),iwa(4*n+1),
     *         maxclq,iwa(1),iwa(n+1),iwa(2*n+1),iwa(3*n+1),bwa)
      call seq(n,indrow,jpntr,indcol,ipntr,iwa(4*n+1),ngrp,maxgrp,
     *         iwa(n+1),bwa)
      mingrp = max0(mingrp,maxclq)
      if (maxgrp .eq. mingrp) go to 130
c
c     Color the intersection graph of the columns of A
c     with the incidence-degree (ID) ordering.
c
      call ido(m,n,indrow,jpntr,indcol,ipntr,iwa(5*n+1),iwa(4*n+1),
     *         maxclq,iwa(1),iwa(n+1),iwa(2*n+1),iwa(3*n+1),bwa)
      call seq(n,indrow,jpntr,indcol,ipntr,iwa(4*n+1),iwa(1),numgrp,
     *         iwa(n+1),bwa)
      mingrp = max0(mingrp,maxclq)
      if (numgrp .ge. maxgrp) go to 100
      maxgrp = numgrp
      do 90 j = 1, n
         ngrp(j) = iwa(j)
   90    continue
      if (maxgrp .eq. mingrp) go to 130
  100 continue
c
c     Color the intersection graph of the columns of A
c     with the largest-first (LF) ordering.
c
      call numsrt(n,n-1,iwa(5*n+1),-1,iwa(4*n+1),iwa(2*n+1),iwa(n+1))
      call seq(n,indrow,jpntr,indcol,ipntr,iwa(4*n+1),iwa(1),numgrp,
     *         iwa(n+1),bwa)
      if (numgrp .ge. maxgrp) go to 120
      maxgrp = numgrp
      do 110 j = 1, n
         ngrp(j) = iwa(j)
  110    continue
  120 continue
c
c     Exit from program.
c
```

```
 130 continue
     return
c
c    Last card of subroutine dsm.
c
     end
```

```
      subroutine degr(n,indrow,jpntr,indcol,ipntr,ndeg,iwa,bwa)
      integer n
      integer indrow(1),jpntr(1),indcol(1),ipntr(1),ndeg(n),iwa(n)
      logical bwa(n)
c     **********
c
c     subroutine degr
c
c     Given the sparsity pattern of an m by n matrix A,
c     this subroutine determines the degree sequence for
c     the intersection graph of the columns of A.
c
c     In graph-theory terminology, the intersection graph of
c     the columns of A is the loopless graph G with vertices
c     a(j), j = 1,2,...,n where a(j) is the j-th column of A
c     and with edge (a(i),a(j)) if and only if columns i and j
c     have a non-zero in the same row position.
c
c     Note that the value of m is not needed by degr and is
c     therefore not present in the subroutine statement.
c
c     The subroutine statement is
c
c       subroutine degr(n,indrow,jpntr,indcol,ipntr,ndeg,iwa,bwa)
c
c     where
c
c       n is a positive integer input variable set to the number
c         of columns of A.
c
c       indrow is an integer input array which contains the row
c         indices for the non-zeroes in the matrix A.
c
c       jpntr is an integer input array of length n + 1 which
c         specifies the locations of the row indices in indrow.
c         The row indices for column j are
c
c               indrow(k), k = jpntr(j),...,jpntr(j+1)-1.
c
c         Note that jpntr(n+1)-1 is then the number of non-zero
c         elements of the matrix A.
c
c       indcol is an integer input array which contains the
c         column indices for the non-zeroes in the matrix A.
c
c       ipntr is an integer input array of length m + 1 which
c         specifies the locations of the column indices in indcol.
c         The column indices for row i are
c
c               indcol(k), k = ipntr(i),...,ipntr(i+1)-1.
c
c         Note that ipntr(m+1)-1 is then the number of non-zero
c         elements of the matrix A.
c
```

```
c         ndeg is an integer output array of length n which
c           specifies the degree sequence. The degree of the
c           j-th column of A is ndeg(j).
c
c         iwa is an integer work array of length n.
c
c         bwa is a logical work array of length n.
c
c       Argonne National Laboratory. MINPACK Project. June 1982.
c       Thomas F. Coleman, Burton S. Garbow, Jorge J. More
c
c       **********
        integer deg,ic,ip,ipl,ipu,ir,jcol,jp,jpl,jpu
c
c       Initialization block.
c
        do 10 jp = 1, n
           ndeg(jp) = 0
           bwa(jp) = .false.
   10      continue
c
c       Compute the degree sequence by determining the contributions
c       to the degrees from the current(jcol) column and further
c       columns which have not yet been considered.
c
        if (n .lt. 2) go to 90
        do 80 jcol = 2, n
           bwa(jcol) = .true.
           deg = 0
c
c          Determine all positions (ir,jcol) which correspond
c          to non-zeroes in the matrix.
c
           jpl = jpntr(jcol)
           jpu = jpntr(jcol+1) - 1
           if (jpu .lt. jpl) go to 50
           do 40 jp = jpl, jpu
              ir = indrow(jp)
c
c             For each row ir, determine all positions (ir,ic)
c             which correspond to non-zeroes in the matrix.
c
              ipl = ipntr(ir)
              ipu = ipntr(ir+1) - 1
              do 30 ip = ipl, ipu
                 ic = indcol(ip)
c
c                Array bwa marks columns which have contributed to
c                the degree count of column jcol. Update the degree
c                counts of these columns. Array iwa records the
c                marked columns.
c
                 if (bwa(ic)) go to 20
                 bwa(ic) = .true.
```

```
                  ndeg(ic) = ndeg(ic) + 1
                  deg = deg + 1
                  iwa(deg) = ic
   20             continue
   30             continue
   40          continue
   50       continue
c
c           Un-mark the columns recorded by iwa and finalize the
c           degree count of column jcol.
c
            if (deg .lt. 1) go to 70
            do 60 jp = 1, deg
               ic = iwa(jp)
               bwa(ic) = .false.
   60          continue
            ndeg(jcol) = ndeg(jcol) + deg
   70       continue
   80       continue
   90 continue
         return
c
c        Last card of subroutine degr.
c
         end
```

```
      subroutine ido(m,n,indrow,jpntr,indcol,ipntr,ndeg,list,
     *               maxclq,iwa1,iwa2,iwa3,iwa4,bwa)
      integer m,n,maxclq
      integer indrow(1),jpntr(1),indcol(1),ipntr(1),ndeg(n),list(n),
     *        iwa1(n),iwa2(n),iwa3(n),iwa4(n)
      logical bwa(n)
c     **********
c
c     subroutine ido
c
c     Given the sparsity pattern of an m by n matrix A, this
c     subroutine determines an incidence-degree ordering of the
c     columns of A.
c
c     The incidence-degree ordering is defined for the loopless
c     graph G with vertices a(j), j = 1,2,...,n where a(j) is the
c     j-th column of A and with edge (a(i),a(j)) if and only if
c     columns i and j have a non-zero in the same row position.
c
c     At each stage of ido, a column of maximal incidence is
c     chosen and ordered. If jcol is an un-ordered column, then
c     the incidence of jcol is the number of ordered columns
c     adjacent to jcol in the graph G. Among all the columns of
c     maximal incidence, ido chooses a column of maximal degree.
c
c     The subroutine statement is
c
c       subroutine ido(m,n,indrow,jpntr,indcol,ipntr,ndeg,list,
c                      maxclq,iwa1,iwa2,iwa3,iwa4,bwa)
c
c     where
c
c       m is a positive integer input variable set to the number
c         of rows of A.
c
c       n is a positive integer input variable set to the number
c         of columns of A.
c
c       indrow is an integer input array which contains the row
c         indices for the non-zeroes in the matrix A.
c
c       jpntr is an integer input array of length n + 1 which
c         specifies the locations of the row indices in indrow.
c         The row indices for column j are
c
c               indrow(k), k = jpntr(j),...,jpntr(j+1)-1.
c
c         Note that jpntr(n+1)-1 is then the number of non-zero
c         elements of the matrix A.
c
c       indcol is an integer input array which contains the
c         column indices for the non-zeroes in the matrix A.
c
c       ipntr is an integer input array of length m + 1 which
```

```
c           specifies the locations of the column indices in indcol.
c           The column indices for row i are
c
c                 indcol(k), k = ipntr(i),...,ipntr(i+1)-1.
c
c           Note that ipntr(m+1)-1 is then the number of non-zero
c           elements of the matrix A.
c
c         ndeg is an integer input array of length n which specifies
c           the degree sequence. The degree of the j-th column
c           of A is ndeg(j).
c
c         list is an integer output array of length n which specifies
c           the incidence-degree ordering of the columns of A. The j-th
c           column in this order is list(j).
c
c         maxclq is an integer output variable set to the size
c           of the largest clique found during the ordering.
c
c         iwa1,iwa2,iwa3, and iwa4 are integer work arrays of length n.
c
c         bwa is a logical work array of length n.
c
c       Subprograms called
c
c         MINPACK-supplied ... numsrt
c
c         FORTRAN-supplied ... max0
c
c       Argonne National Laboratory. MINPACK Project. June 1982.
c       Thomas F. Coleman, Burton S. Garbow, Jorge J. More
c
c       **********
        integer deg,head,ic,ip,ipl,ipu,ir,jcol,jp,jpl,jpu,l,maxinc,
       *        maxlst,ncomp,numinc,numlst,numord,numwgt
c
c       Sort the degree sequence.
c
        call numsrt(n,n-1,ndeg,-1,iwa4,iwa1,iwa3)
c
c       Initialization block.
c
c       Create a doubly-linked list to access the incidences of the
c       columns. The pointers for the linked list are as follows.
c
c       Each un-ordered column jcol is in a list (the incidence list)
c       of columns with the same incidence.
c
c       iwa1(numinc+1) is the first column in the numinc list
c       unless iwa1(numinc+1) = 0. In this case there are
c       no columns in the numinc list.
c
c       iwa2(jcol) is the column before jcol in the incidence list
c       unless iwa2(jcol) = 0. In this case jcol is the first
```

```
c       column in this incidence list.
c
c       iwa3(jcol) is the column after jcol in the incidence list
c       unless iwa3(jcol) = 0. In this case jcol is the last
c       column in this incidence list.
c
c       If jcol is an un-ordered column, then list(jcol) is the
c       incidence of jcol in the graph. If jcol is an ordered column,
c       then list(jcol) is the incidence-degree order of column jcol.
c
        maxinc = 0
        do 10 jp = 1, n
           list(jp) = 0
           bwa(jp) = .false.
           iwa1(jp) = 0
           l = iwa4(jp)
           if (jp .ne. 1) iwa2(l) = iwa4(jp-1)
           if (jp .ne. n) iwa3(l) = iwa4(jp+1)
   10      continue
        iwa1(1) = iwa4(1)
        l = iwa4(1)
        iwa2(l) = 0
        l = iwa4(n)
        iwa3(l) = 0
c
c       Determine the maximal search length for the list
c       of columns of maximal incidence.
c
        maxlst = 0
        do 20 ir = 1, m
           maxlst = maxlst + (ipntr(ir+1) - ipntr(ir))**2
   20      continue
        maxlst = maxlst/n
        maxclq = 1
c
c       Beginning of iteration loop.
c
        do 140 numord = 1, n
c
c          Choose a column jcol of maximal degree among the
c          columns of maximal incidence.
c
           jp = iwa1(maxinc+1)
           numlst = 1
           numwgt = -1
   30      continue
              if (ndeg(jp) .le. numwgt) go to 40
              numwgt = ndeg(jp)
              jcol = jp
   40         continue
              jp = iwa3(jp)
              numlst = numlst + 1
              if (jp .gt. 0 .and. numlst .le. maxlst) go to 30
           list(jcol) = numord
```

```
c
c        Delete column jcol from the list of columns of
c        maximal incidence.
c
         l = iwa2(jcol)
         if (l .eq. 0) iwa1(maxinc+1) = iwa3(jcol)
         if (l .gt. 0) iwa3(l) = iwa3(jcol)
         l = iwa3(jcol)
         if (l .gt. 0) iwa2(l) = iwa2(jcol)
c
c        Update the size of the largest clique
c        found during the ordering.
c
         if (maxinc .eq. 0) ncomp = 0
         ncomp = ncomp + 1
         if (maxinc + 1 .eq. ncomp) maxclq = max0(maxclq,ncomp)
c
c        Update the maximal incidence count.
c
   50    continue
            if (iwa1(maxinc+1) .gt. 0) go to 60
            maxinc = maxinc - 1
            if (maxinc .ge. 0) go to 50
   60    continue
c
c        Find all columns adjacent to column jcol.
c
         bwa(jcol) = .true.
         deg = 0
c
c        Determine all positions (ir,jcol) which correspond
c        to non-zeroes in the matrix.
c
         jpl = jpntr(jcol)
         jpu = jpntr(jcol+1) - 1
         if (jpu .lt. jpl) go to 100
         do 90 jp = jpl, jpu
            ir = indrow(jp)
c
c           For each row ir, determine all positions (ir,ic)
c           which correspond to non-zeroes in the matrix.
c
            ipl = ipntr(ir)
            ipu = ipntr(ir+1) - 1
            do 80 ip = ipl, ipu
               ic = indcol(ip)
c
c              Array bwa marks columns which are adjacent to
c              column jcol. Array iwa4 records the marked columns.
c
               if (bwa(ic)) go to 70
               bwa(ic) = .true.
               deg = deg + 1
               iwa4(deg) = ic
```

```
   70             continue
   80             continue
   90          continue
  100       continue
c
c          Update the pointers to the incidence lists.
c
           if (deg .lt. 1) go to 130
           do 120 jp = 1, deg
              ic = iwa4(jp)
              if (list(ic) .gt. 0) go to 110
              numinc = -list(ic) + 1
              list(ic) = -numinc
              maxinc = max0(maxinc,numinc)
c
c             Delete column ic from the numinc-1 list.
c
              l = iwa2(ic)
              if (l .eq. 0) iwa1(numinc) = iwa3(ic)
              if (l .gt. 0) iwa3(l) = iwa3(ic)
              l = iwa3(ic)
              if (l .gt. 0) iwa2(l) = iwa2(ic)
c
c             Add column ic to the numinc list.
c
              head = iwa1(numinc+1)
              iwa1(numinc+1) = ic
              iwa2(ic) = 0
              iwa3(ic) = head
              if (head .gt. 0) iwa2(head) = ic
  110         continue
c
c             Un-mark column ic in the array bwa.
c
              bwa(ic) = .false.
  120         continue
  130      continue
           bwa(jcol) = .false.
c
c          End of iteration loop.
c
  140      continue
c
c       Invert the array list.
c
        do 150 jcol = 1, n
           numord = list(jcol)
           iwa1(numord) = jcol
  150      continue
        do 160 jp = 1, n
           list(jp) = iwa1(jp)
  160      continue
        return
c
```

```
c       Last card of subroutine ido.
c
        end
```

```
      subroutine numsrt(n,nmax,num,mode,index,last,next)
      integer n,nmax,mode
      integer num(n),index(n),last(1),next(n)
c     **********
c
c     subroutine numsrt
c
c     Given a sequence of integers, this subroutine groups
c     together those indices with the same sequence value
c     and, optionally, sorts the sequence into either
c     ascending or descending order.
c
c     The sequence of integers is defined by the array num,
c     and it is assumed that the integers are each from the set
c     0,1,...,nmax. On output the indices k such that num(k) = l
c     for any l = 0,1,...,nmax can be obtained from the arrays
c     last and next as follows.
c
c           k = last(l+1)
c           while (k .ne. 0) k = next(k)
c
c     Optionally, the subroutine produces an array index so that
c     the sequence num(index(i)), i = 1,2,...,n is sorted.
c
c     The subroutine statement is
c
c       subroutine numsrt(n,nmax,num,mode,index,last,next)
c
c     where
c
c       n is a positive integer input variable.
c
c       nmax is a positive integer input variable.
c
c       num is an input array of length n which contains the
c         sequence of integers to be grouped and sorted. It
c         is assumed that the integers are each from the set
c         0,1,...,nmax.
c
c       mode is an integer input variable. The sequence num is
c         sorted in ascending order if mode is positive and in
c         descending order if mode is negative. If mode is 0,
c         no sorting is done.
c
c       index is an integer output array of length n set so
c         that the sequence
c
c               num(index(i)), i = 1,2,...,n
c
c         is sorted according to the setting of mode. If mode
c         is 0, index is not referenced.
c
c       last is an integer output array of length nmax + 1. The
c         index of num for the last occurrence of l is last(l+1)
```

```
c         for any l = 0,1,...,nmax unless last(l+1) = 0. In
c         this case l does not appear in num.
c
c       next is an integer output array of length n. If
c         num(k) = l, then the index of num for the previous
c         occurrence of l is next(k) for any l = 0,1,...,nmax
c         unless next(k) = 0. In this case there is no previous
c         occurrence of l in num.
c
c     Argonne National Laboratory. MINPACK Project. June 1982.
c     Thomas F. Coleman, Burton S. Garbow, Jorge J. More
c
c     **********
      integer i,j,jp,k,l,nmaxp1,nmaxp2
c
c     Determine the arrays next and last.
c
      nmaxp1 = nmax + 1
      do 10 i = 1, nmaxp1
         last(i) = 0
   10    continue
      do 20 k = 1, n
         l = num(k)
         next(k) = last(l+1)
         last(l+1) = k
   20    continue
      if (mode .eq. 0) go to 60
c
c     Store the pointers to the sorted array in index.
c
      i = 1
      nmaxp2 = nmaxp1 + 1
      do 50 j = 1, nmaxp1
         jp = j
         if (mode .lt. 0) jp = nmaxp2 - j
         k = last(jp)
   30    continue
            if (k .eq. 0) go to 40
            index(i) = k
            i = i + 1
            k = next(k)
            go to 30
   40    continue
   50    continue
   60 continue
      return
c
c     Last card of subroutine numsrt.
c
      end
```

```
      subroutine seq(n,indrow,jpntr,indcol,ipntr,list,ngrp,maxgrp,
     *               iwa,bwa)
      integer n,maxgrp
      integer indrow(1),jpntr(1),indcol(1),ipntr(1),list(n),ngrp(n),
     *        iwa(n)
      logical bwa(n)
c     **********
c
c     subroutine seq
c
c     Given the sparsity pattern of an m by n matrix A, this
c     subroutine determines a consistent partition of the
c     columns of A by a sequential algorithm.
c
c     A consistent partition is defined in terms of the loopless
c     graph G with vertices a(j), j = 1,2,...,n where a(j) is the
c     j-th column of A and with edge (a(i),a(j)) if and only if
c     columns i and j have a non-zero in the same row position.
c
c     A partition of the columns of A into groups is consistent
c     if the columns in any group are not adjacent in the graph G.
c     In graph-theory terminology, a consistent partition of the
c     columns of A corresponds to a coloring of the graph G.
c
c     The subroutine examines the columns in the order specified
c     by the array list, and assigns the current column to the
c     group with the smallest possible number.
c
c     Note that the value of m is not needed by seq and is
c     therefore not present in the subroutine statement.
c
c     The subroutine statement is
c
c       subroutine seq(n,indrow,jpntr,indcol,ipntr,list,ngrp,maxgrp,
c                      iwa,bwa)
c
c     where
c
c       n is a positive integer input variable set to the number
c         of columns of A.
c
c       indrow is an integer input array which contains the row
c         indices for the non-zeroes in the matrix A.
c
c       jpntr is an integer input array of length n + 1 which
c         specifies the locations of the row indices in indrow.
c         The row indices for column j are
c
c               indrow(k), k = jpntr(j),...,jpntr(j+1)-1.
c
c         Note that jpntr(n+1)-1 is then the number of non-zero
c         elements of the matrix A.
c
c       indcol is an integer input array which contains the
```

```
c          column indices for the non-zeroes in the matrix A.
c
c        ipntr is an integer input array of length m + 1 which
c          specifies the locations of the column indices in indcol.
c          The column indices for row i are
c
c                indcol(k), k = ipntr(i),...,ipntr(i+1)-1.
c
c          Note that ipntr(m+1)-1 is then the number of non-zero
c          elements of the matrix A.
c
c        list is an integer input array of length n which specifies
c          the order to be used by the sequential algorithm.
c          The j-th column in this order is list(j).
c
c        ngrp is an integer output array of length n which specifies
c          the partition of the columns of A. Column jcol belongs
c          to group ngrp(jcol).
c
c        maxgrp is an integer output variable which specifies the
c          number of groups in the partition of the columns of A.
c
c        iwa is an integer work array of length n.
c
c        bwa is a logical work array of length n.
c
c     Argonne National Laboratory. MINPACK Project. June 1982.
c     Thomas F. Coleman, Burton S. Garbow, Jorge J. More
c
c     **********
      integer deg,ic,ip,ipl,ipu,ir,j,jcol,jp,jpl,jpu,l,numgrp
c
c     Initialization block.
c
      maxgrp = 0
      do 10 jp = 1, n
         ngrp(jp) = n
         bwa(jp) = .false.
   10    continue
      bwa(n) = .true.
c
c     Beginning of iteration loop.
c
      do 100 j = 1, n
         jcol = list(j)
c
c        Find all columns adjacent to column jcol.
c
         deg = 0
c
c        Determine all positions (ir,jcol) which correspond
c        to non-zeroes in the matrix.
c
         jpl = jpntr(jcol)
```

```
            jpu = jpntr(jcol+1) - 1
            if (jpu .lt. jpl) go to 50
            do 40 jp = jpl, jpu
               ir = indrow(jp)
c
c              For each row ir, determine all positions (ir,ic)
c              which correspond to non-zeroes in the matrix.
c
               ipl = ipntr(ir)
               ipu = ipntr(ir+1) - 1
               do 30 ip = ipl, ipu
                  ic = indcol(ip)
                  l = ngrp(ic)
c
c                 Array bwa marks the group numbers of the
c                 columns which are adjacent to column jcol.
c                 Array iwa records the marked group numbers.
c
                  if (bwa(l)) go to 20
                  bwa(l) = .true.
                  deg = deg + 1
                  iwa(deg) = l
   20             continue
   30          continue
   40       continue
   50    continue
c
c        Assign the smallest un-marked group number to jcol.
c
         do 60 jp = 1, n
            numgrp = jp
            if (.not. bwa(jp)) go to 70
   60       continue
   70    continue
         ngrp(jcol) = numgrp
         maxgrp = max0(maxgrp,numgrp)
c
c        Un-mark the group numbers.
c
         if (deg .lt. 1) go to 90
         do 80 jp = 1, deg
            l = iwa(jp)
            bwa(l) = .false.
   80       continue
   90    continue
  100    continue
c
c        End of iteration loop.
c
      return
c
c     Last card of subroutine seq.
c
      end
```

```
      subroutine setr(m,n,indrow,jpntr,indcol,ipntr,iwa)
      integer m,n
      integer indrow(1),jpntr(1),indcol(1),ipntr(1),iwa(m)
c     **********
c
c     subroutine setr
c
c     Given a column-oriented definition of the sparsity pattern
c     of an m by n matrix A, this subroutine determines a
c     row-oriented definition of the sparsity pattern of A.
c
c     On input the column-oriented definition is specified by
c     the arrays indrow and jpntr. On output the row-oriented
c     definition is specified by the arrays indcol and ipntr.
c
c     The subroutine statement is
c
c       subroutine setr(m,n,indrow,jpntr,indcol,ipntr,iwa)
c
c     where
c
c       m is a positive integer input variable set to the number
c         of rows of A.
c
c       n is a positive integer input variable set to the number
c         of columns of A.
c
c       indrow is an integer input array which contains the row
c         indices for the non-zeroes in the matrix A.
c
c       jpntr is an integer input array of length n + 1 which
c         specifies the locations of the row indices in indrow.
c         The row indices for column j are
c
c               indrow(k), k = jpntr(j),...,jpntr(j+1)-1.
c
c         Note that jpntr(n+1)-1 is then the number of non-zero
c         elements of the matrix A.
c
c       indcol is an integer output array which contains the
c         column indices for the non-zeroes in the matrix A.
c
c       ipntr is an integer output array of length m + 1 which
c         specifies the locations of the column indices in indcol.
c         The column indices for row i are
c
c               indcol(k), k = ipntr(i),...,ipntr(i+1)-1.
c
c         Note that ipntr(1) is set to 1 and that ipntr(m+1)-1 is
c         then the number of non-zero elements of the matrix A.
c
c       iwa is an integer work array of length m.
c
c     Argonne National Laboratory. MINPACK Project. June 1982.
```

```
c      Thomas F. Coleman, Burton S. Garbow, Jorge J. More
c
c      **********
       integer ir,jcol,jp,jpl,jpu,l,nnz
c
c      Determine the number of non-zeroes in the rows.
c
       do 10 ir = 1, m
          iwa(ir) = 0
   10     continue
       nnz = jpntr(n+1) - 1
       do 20 jp = 1, nnz
          ir = indrow(jp)
          iwa(ir) = iwa(ir) + 1
   20     continue
c
c      Set pointers to the start of the rows in indcol.
c
       ipntr(1) = 1
       do 30 ir = 1, m
          ipntr(ir+1) = ipntr(ir) + iwa(ir)
          iwa(ir) = ipntr(ir)
   30     continue
c
c      Fill indcol.
c
       do 60 jcol = 1, n
          jpl = jpntr(jcol)
          jpu = jpntr(jcol+1) - 1
          if (jpu .lt. jpl) go to 50
          do 40 jp = jpl, jpu
             ir = indrow(jp)
             l = iwa(ir)
             indcol(l) = jcol
             iwa(ir) = iwa(ir) + 1
   40        continue
   50     continue
   60     continue
       return
c
c      Last card of subroutine setr.
c
       end
```

```
      subroutine slo(n,indrow,jpntr,indcol,ipntr,ndeg,list,
     *               maxclq,iwa1,iwa2,iwa3,iwa4,bwa)
      integer n,maxclq
      integer indrow(1),jpntr(1),indcol(1),ipntr(1),ndeg(n),
     *        list(n),iwa1(n),iwa2(n),iwa3(n),iwa4(n)
      logical bwa(n)
c     **********
c
c     subroutine slo
c
c     Given the sparsity pattern of an m by n matrix A, this
c     subroutine determines the smallest-last ordering of the
c     columns of A.
c
c     The smallest-last ordering is defined for the loopless
c     graph G with vertices a(j), j = 1,2,...,n where a(j) is the
c     j-th column of A and with edge (a(i),a(j)) if and only if
c     columns i and j have a non-zero in the same row position.
c
c     The smallest-last ordering is determined recursively by
c     letting list(k), k = n,...,1 be a column with least degree
c     in the subgraph spanned by the un-ordered columns.
c
c     Note that the value of m is not needed by slo and is
c     therefore not present in the subroutine statement.
c
c     The subroutine statement is
c
c       subroutine slo(n,indrow,jpntr,indcol,ipntr,ndeg,list,
c                      maxclq,iwa1,iwa2,iwa3,iwa4,bwa)
c
c     where
c
c       n is a positive integer input variable set to the number
c         of columns of A.
c
c       indrow is an integer input array which contains the row
c         indices for the non-zeroes in the matrix A.
c
c       jpntr is an integer input array of length n + 1 which
c         specifies the locations of the row indices in indrow.
c         The row indices for column j are
c
c               indrow(k), k = jpntr(j),...,jpntr(j+1)-1.
c
c         Note that jpntr(n+1)-1 is then the number of non-zero
c         elements of the matrix A.
c
c       indcol is an integer input array which contains the
c         column indices for the non-zeroes in the matrix A.
c
c       ipntr is an integer input array of length m + 1 which
c         specifies the locations of the column indices in indcol.
c         The column indices for row i are
```

```
c
c                indcol(k), k = ipntr(i),...,ipntr(i+1)-1.
c
c           Note that ipntr(m+1)-1 is then the number of non-zero
c           elements of the matrix A.
c
c         ndeg is an integer input array of length n which specifies
c           the degree sequence. The degree of the j-th column
c           of A is ndeg(j).
c
c         list is an integer output array of length n which specifies
c           the smallest-last ordering of the columns of A. The j-th
c           column in this order is list(j).
c
c         maxclq is an integer output variable set to the size
c           of the largest clique found during the ordering.
c
c         iwa1,iwa2,iwa3, and iwa4 are integer work arrays of length n.
c
c         bwa is a logical work array of length n.
c
c       Subprograms called
c
c         FORTRAN-supplied ... min0
c
c       Argonne National Laboratory. MINPACK Project. June 1982.
c       Thomas F. Coleman, Burton S. Garbow, Jorge J. More
c
c       **********
        integer deg,head,ic,ip,ipl,ipu,ir,jcol,jp,jpl,jpu,
      *         l,mindeg,numdeg,numord
c
c       Initialization block.
c
        mindeg = n
        do 10 jp = 1, n
           iwa1(jp) = 0
           bwa(jp) = .false.
           list(jp) = ndeg(jp)
           mindeg = min0(mindeg,ndeg(jp))
     10    continue
c
c       Create a doubly-linked list to access the degrees of the
c       columns. The pointers for the linked list are as follows.
c
c       Each un-ordered column jcol is in a list (the degree
c       list) of columns with the same degree.
c
c       iwa1(numdeg+1) is the first column in the numdeg list
c       unless iwa1(numdeg+1) = 0. In this case there are
c       no columns in the numdeg list.
c
c       iwa2(jcol) is the column before jcol in the degree list
c       unless iwa2(jcol) = 0. In this case jcol is the first
```

```
c     column in this degree list.
c
c     iwa3(jcol) is the column after jcol in the degree list
c     unless iwa3(jcol) = 0. In this case jcol is the last
c     column in this degree list.
c
c     If jcol is an un-ordered column, then list(jcol) is the
c     degree of jcol in the graph induced by the un-ordered
c     columns. If jcol is an ordered column, then list(jcol)
c     is the smallest-last order of column jcol.
c
      do 20 jp = 1, n
         numdeg = ndeg(jp)
         head = iwa1(numdeg+1)
         iwa1(numdeg+1) = jp
         iwa2(jp) = 0
         iwa3(jp) = head
         if (head .gt. 0) iwa2(head) = jp
   20    continue
      maxclq = 0
      numord = n
c
c     Beginning of iteration loop.
c
   30 continue
c
c        Mark the size of the largest clique
c        found during the ordering.
c
         if (mindeg + 1 .eq. numord .and. maxclq .eq. 0)
     *       maxclq = numord
c
c        Choose a column jcol of minimal degree mindeg.
c
   40    continue
            jcol = iwa1(mindeg+1)
            if (jcol .gt. 0) go to 50
            mindeg = mindeg + 1
            go to 40
   50    continue
         list(jcol) = numord
         numord = numord - 1
c
c        Termination test.
c
         if (numord .eq. 0) go to 120
c
c        Delete column jcol from the mindeg list.
c
         l = iwa3(jcol)
         iwa1(mindeg+1) = l
         if (l .gt. 0) iwa2(l) = 0
c
c        Find all columns adjacent to column jcol.
```

```
c
         bwa(jcol) = .true.
         deg = 0
c
c        Determine all positions (ir,jcol) which correspond
c        to non-zeroes in the matrix.
c
         jpl = jpntr(jcol)
         jpu = jpntr(jcol+1) - 1
         if (jpu .lt. jpl) go to 90
         do 80 jp = jpl, jpu
            ir = indrow(jp)
c
c           For each row ir, determine all positions (ir,ic)
c           which correspond to non-zeroes in the matrix.
c
            ipl = ipntr(ir)
            ipu = ipntr(ir+1) - 1
            do 70 ip = ipl, ipu
               ic = indcol(ip)
c
c              Array bwa marks columns which are adjacent to
c              column jcol. Array iwa4 records the marked columns.
c
               if (bwa(ic)) go to 60
               bwa(ic) = .true.
               deg = deg + 1
               iwa4(deg) = ic
   60          continue
   70          continue
   80       continue
   90    continue
c
c        Update the pointers to the current degree lists.
c
         if (deg .lt. 1) go to 110
         do 100 jp = 1, deg
            ic = iwa4(jp)
            numdeg = list(ic)
            list(ic) = list(ic) - 1
            mindeg = min0(mindeg,list(ic))
c
c           Delete column ic from the numdeg list.
c
            l = iwa2(ic)
            if (l .eq. 0) iwa1(numdeg+1) = iwa3(ic)
            if (l .gt. 0) iwa3(l) = iwa3(ic)
            l = iwa3(ic)
            if (l .gt. 0) iwa2(l) = iwa2(ic)
c
c           Add column ic to the numdeg-1 list.
c
            head = iwa1(numdeg)
            iwa1(numdeg) = ic
```

```
            iwa2(ic) = 0
            iwa3(ic) = head
            if (head .gt. 0) iwa2(head) = ic
c
c           Un-mark column ic in the array bwa.
c
            bwa(ic) = .false.
  100       continue
  110     continue
c
c         End of iteration loop.
c
          go to 30
  120 continue
c
c     Invert the array list.
c
      do 130 jcol = 1, n
         numord = list(jcol)
         iwa1(numord) = jcol
  130    continue
      do 140 jp = 1, n
         list(jp) = iwa1(jp)
  140    continue
      return
c
c     Last card of subroutine slo.
c
      end
```

```
      subroutine srtdat(n,nnz,indrow,indcol,jpntr,iwa)
      integer n,nnz
      integer indrow(nnz),indcol(nnz),jpntr(1),iwa(n)
c     **********
c
c     subroutine srtdat
c
c     Given the non-zero elements of an m by n matrix A in
c     arbitrary order as specified by their row and column
c     indices, this subroutine permutes these elements so
c     that their column indices are in non-decreasing order.
c
c     On input it is assumed that the elements are specified in
c
c           indrow(k),indcol(k), k = 1,...,nnz.
c
c     On output the elements are permuted so that indcol is
c     in non-decreasing order. In addition, the array jpntr
c     is set so that the row indices for column j are
c
c           indrow(k), k = jpntr(j),...,jpntr(j+1)-1.
c
c     Note that the value of m is not needed by srtdat and is
c     therefore not present in the subroutine statement.
c
c     The subroutine statement is
c
c       subroutine srtdat(n,nnz,indrow,indcol,jpntr,iwa)
c
c     where
c
c       n is a positive integer input variable set to the number
c         of columns of A.
c
c       nnz is a positive integer input variable set to the number
c         of non-zero elements of A.
c
c       indrow is an integer array of length nnz. On input indrow
c         must contain the row indices of the non-zero elements of A.
c         On output indrow is permuted so that the corresponding
c         column indices of indcol are in non-decreasing order.
c
c       indcol is an integer array of length nnz. On input indcol
c         must contain the column indices of the non-zero elements
c         of A. On output indcol is permuted so that these indices
c         are in non-decreasing order.
c
c       jpntr is an integer output array of length n + 1 which
c         specifies the locations of the row indices in the output
c         indrow. The row indices for column j are
c
c              indrow(k), k = jpntr(j),...,jpntr(j+1)-1.
c
c         Note that jpntr(1) is set to 1 and that jpntr(n+1)-1
```

```
c           is then nnz.
c
c         iwa is an integer work array of length n.
c
c       Subprograms called
c
c         FORTRAN-supplied ... max0
c
c       Argonne National Laboratory. MINPACK Project. June 1982.
c       Thomas F. Coleman, Burton S. Garbow, Jorge J. More
c
c       **********
        integer i,j,k,l
c
c       Determine the number of non-zeroes in the columns.
c
        do 10 j = 1, n
           iwa(j) = 0
   10      continue
        do 20 k = 1, nnz
           j = indcol(k)
           iwa(j) = iwa(j) + 1
   20      continue
c
c       Set pointers to the start of the columns in indrow.
c
        jpntr(1) = 1
        do 30 j = 1, n
           jpntr(j+1) = jpntr(j) + iwa(j)
           iwa(j) = jpntr(j)
   30      continue
        k = 1
c
c       Begin in-place sort.
c
   40 continue
           j = indcol(k)
           if (k .lt. jpntr(j) .or. k .ge. jpntr(j+1)) go to 50
c
c             Current element is in position. Now examine the
c             next element or the first un-sorted element in
c             the j-th group.
c
              k = max0(k+1,iwa(j))
              go to 60
   50      continue
c
c             Current element is not in position. Place element
c             in position and make the displaced element the
c             current element.
c
              l = iwa(j)
              iwa(j) = iwa(j) + 1
              i = indrow(k)
```

```
          indrow(k) = indrow(l)
          indcol(k) = indcol(l)
          indrow(l) = i
          indcol(l) = j
   60   continue
        if (k .le. nnz) go to 40
      return
c
c     Last card of subroutine srtdat.
c
      end
```

```
      subroutine fdjs(n,indrow,jpntr,ngrp,numgrp,d,fjacd,fjac)
      integer n,numgrp
      integer indrow(1),jpntr(1),ngrp(n)
      real d(n),fjacd(1),fjac(1)
c     **********
c
c     subroutine fdjs
c
c     Given a consistent partition of the columns of an m by n
c     Jacobian matrix into groups, this subroutine computes
c     approximations to those columns in a given group.
c
c     A partition is consistent if the columns in any group
c     do not have a non-zero in the same row position.
c
c     Approximations to the columns of the Jacobian matrix in a
c     given group can be obtained by specifying a difference
c     parameter array d with d(jcol) non-zero if and only if
c     jcol is a column in the group, and an approximation to
c     jac*d where jac denotes the Jacobian matrix of a mapping F.
c
c     d can be defined with the following segment of code.
c
c           do 10 jcol = 1, n
c              d(jcol) = 0.0
c              if (ngrp(jcol) .eq. numgrp) d(jcol) = eta(jcol)
c        10    continue
c
c     In the above code numgrp is the given group number,
c     ngrp(jcol) is the group number of column jcol, and
c     eta(jcol) is the difference parameter used to
c     approximate column jcol of the Jacobian matrix.
c     Suitable values for the array eta must be provided.
c
c     As mentioned above, an approximation to jac*d must
c     also be provided. For example, the approximation
c
c           F(x+d) - F(x)
c
c     corresponds to the forward difference formula at x.
c
c     Note that the value of m is not needed by fdjs and is
c     therefore not present in the subroutine statement.
c
c     The subroutine statement is
c
c       subroutine fdjs(n,indrow,jpntr,ngrp,numgrp,d,fjacd,fjac)
c
c     where
c
c       n is a positive integer input variable set to the number
c         of columns of the Jacobian matrix.
c
c       indrow is an integer input array which contains the row
```

```
c          indices for the non-zeroes in the Jacobian matrix.
c
c       jpntr is an integer input array of length n + 1 which
c         specifies the locations of the row indices in indrow.
c         The row indices for column j are
c
c               indrow(k), k = jpntr(j),...,jpntr(j+1)-1.
c
c         Note that jpntr(n+1)-1 is then the number of non-zero
c         elements of the Jacobian matrix.
c
c       ngrp is an integer input array of length n which specifies
c         the partition of the columns of the Jacobian matrix.
c         Column jcol belongs to group ngrp(jcol).
c
c       numgrp is a positive integer input variable set to a group
c         number in the partition. The columns of the Jacobian
c         matrix in this group are to be estimated on this call.
c
c       d is an input array of length n which contains the
c         difference parameter vector for the estimate of
c         the Jacobian matrix columns in group numgrp.
c
c       fjacd is an input array of length m which contains
c         an approximation to the difference vector jac*d,
c         where jac denotes the Jacobian matrix.
c
c       fjac is an output array of length nnz, where nnz is the
c         number of its non-zero elements. At each call of fdjs,
c         fjac is updated to include the non-zero elements of the
c         Jacobian matrix for those columns in group numgrp. fjac
c         should not be altered between successive calls to fdjs.
c
c     Argonne National Laboratory. MINPACK Project. June 1982.
c     Thomas F. Coleman, Burton S. Garbow, Jorge J. More
c
c     **********
      integer ir,jcol,jp,jpl,jpu
c
c     Compute estimates of Jacobian matrix columns in group
c     numgrp. The array fjacd must contain an approximation
c     to jac*d, where jac denotes the jacobian matrix and d
c     is a difference parameter vector with d(jcol) non-zero
c     if and only if jcol is a column in group numgrp.
c
      do 30 jcol = 1, n
         if (ngrp(jcol) .ne. numgrp) go to 20
         jpl = jpntr(jcol)
         jpu = jpntr(jcol+1) - 1
         if (jpu .lt. jpl) go to 20
         do 10 jp = jpl, jpu
            ir = indrow(jp)
            fjac(jp) = fjacd(ir)/d(jcol)
   10       continue
```

```
 20    continue
 30    continue
       return
c
c      Last card of subroutine fdjs.
c
       end
```