SOFTWARE FOR ESTIMATING SPARSE JACOBIAN MATRICES
by
Thomas F. Coleman and Jorge J. Moré

CERN LIBRARIES, GENEVA
CM-P00069197

ARGONNE NATIONAL LABORATORY, ARGONNE, ILLINOIS
Prepared for the U. S. DEPARTMENT OF ENERGY
under Contract W-31-109-Eng-38
The facilities of Argonne National Laboratory are owned by the United States Government. Under the terms of a contract (W-31-109-Eng-38) among the U.S. Department of Energy, Argonne Universities Association and The University of Chicago, the University employs the staff and operates the Laboratory in accordance with policies and programs formulated, approved and reviewed by the Association.

MEMBERS OF ARGONNE UNIVERSITIES ASSOCIATION

<table>
<thead>
<tr>
<th>The University of Arizona</th>
<th>The University of Kansas</th>
<th>The Ohio State University</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carnegie-Mellon University</td>
<td>Kansas State University</td>
<td>Ohio University</td>
</tr>
<tr>
<td>Case Western Reserve University</td>
<td>Loyola University of Chicago</td>
<td>The Pennsylvania State University</td>
</tr>
<tr>
<td>The University of Chicago</td>
<td>Marquette University</td>
<td>Purdue University</td>
</tr>
<tr>
<td>University of Cincinnati</td>
<td>The University of Michigan</td>
<td>Saint Louis University</td>
</tr>
<tr>
<td>Illinois Institute of Technology</td>
<td>Michigan State University</td>
<td>Southern Illinois University</td>
</tr>
<tr>
<td>University of Illinois</td>
<td>University of Minnesota</td>
<td>The University of Texas at Austin</td>
</tr>
<tr>
<td>Indiana University</td>
<td>University of Missouri</td>
<td>Washington University</td>
</tr>
<tr>
<td>The University of Iowa</td>
<td>Northwestern University</td>
<td>Wayne State University</td>
</tr>
<tr>
<td>Iowa State University</td>
<td>University of Notre Dame</td>
<td>The University of Wisconsin-Madison</td>
</tr>
</tbody>
</table>

---

**NOTICE**

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Printed in the United States of America
Available from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road
Springfield, VA 22161

NTIS price codes
Printed copy: A04
Microfiche copy: A01
Software for Estimating Sparse Jacobian Matrices

Thomas F. Coleman and Jorge J. Moré
Mathematics and Computer Science Division

July 1982
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.
1. Introduction.

In many nonlinear problems it is necessary to estimate the Jacobian matrix of a mapping \( F: \mathbb{R}^n \to \mathbb{R}^m \). 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 \( F: \mathbb{R}^n \to \mathbb{R}^m \) 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

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

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 \( d \in \mathbb{R}^n \) is a vector with components \( \delta_j \neq 0 \) if
$a_j$ belongs to $C$ and $\delta_j = 0$ otherwise, then

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

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

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

In view of this observation, it is possible to determine an $m \times n$ 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 $F : \mathbb{R}^n \to \mathbb{R}^m$ 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 $m \times n$ 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 \( F: \mathbb{R}^n \rightarrow \mathbb{R}^m \) 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 \( m \times n \) 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 \( a_{ij} \neq 0 \):

\[
(2.1) \quad (\text{indrow}(k), \text{indcol}(k)). \quad k = 1, \ldots, \text{n pairs}.
\]

These pairs can be provided in any order. Moreover, duplicate pairs are allowed so that \text{n pairs} 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 \text{ngrp} by setting \( \text{ngrp}(jc) \) to the group number of column \( jc \). In addition, the variable \text{mngrp} provides a lower bound on the number of groups possible in a consistent partition, and the variable \text{mngrp} 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 \text{indrow} and \text{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 \( \rho_{\text{max}} \) where \( \rho_{\text{max}} \) is the maximum number of nonzero elements in any row of \( A \). Usually \text{mngrp} is set to \( \rho_{\text{max}} \), but on some problems \text{mngrp} may exceed \( \rho_{\text{max}} \). For example, if

\[
A = \begin{bmatrix}
  \times & \times \\
  \times & \times \\
  \times & \times 
\end{bmatrix}
\]

then \text{mngrp} is set to 3. In practice DSM needs, at worst, one or two more
groups than the bound specified by \textit{mingrp}. For many problems \textit{maxgrp} agrees with \textit{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

\begin{equation}
\sum_{i=1}^{n} \rho_i^2,
\end{equation}

where \(\rho_i\) is the number of nonzeros 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 \(A^TA\) 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 \textit{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 \texttt{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 \textit{column-oriented} and \textit{row-oriented} definitions of the sparsity pattern. The arrays \textit{indrow} and \textit{jptr} provide a column-oriented definition of the sparsity pattern if the nonzero elements of the \(j\)-th column are

\(\textit{indrow}(k), \ k = \textit{jptr}(j), \ldots, \textit{jptr}(j+1)-1\).

The arrays \textit{indcol} and \textit{iptr} provide a row-oriented definition of the sparsity pattern if the nonzero elements of the \(i\)-th row are
\[ \text{indcol}(k), \quad k = \text{jptr}(i), \ldots, \text{jptr}(i+1)-1. \]

Given the ordered pairs (2.1) for which \( a_i \neq 0 \), 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 \( \text{numgrp} \) can be obtained by specifying a difference parameter array \( d \) with \( d(jcol) \) non-zero if and only if \( jcol \) is a column in group \( \text{numgrp} \), and an approximation to \( F'(x)d \) in the array \( fjac \). The approximation to the columns of the Jacobian matrix in group \( \text{numgrp} \) 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

\[ fjac(k), \quad k = \text{jptr}(j), \ldots, \text{jptr}(j+1)-1. \]

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.

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

Program 3.1

The array \( \text{eta} \) 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

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

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.

\[
\begin{align*}
\text{do } 30 \text{ jcol } = 1, n \\
\quad \text{if (ngrp(jcol) .ne. numgrp) go to 20} \\
\quad jpl = jptr(jcol) \\
\quad jpu = jptr(jcol+1) - 1 \\
\quad \text{if (jpu .lt. jpl) go to 20} \\
\quad \text{do 10 } \text{ jp } = jpl, jpu \\
\quad \quad \text{itr } = \text{indrow(jp)} \\
\quad \quad fjac(jp) = fjacd(itr)/d(jcol) \\
10 \quad \text{continue} \\
20 \quad \text{continue} \\
30 \quad \text{continue}
\end{align*}
\]

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 \( 1,2, \ldots, \text{maxgrp} \). 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 \( F: \mathbb{R}^n \rightarrow \mathbb{R}^n \) such that \( F'(x) \) has a sparsity pattern of the form

\[
\begin{pmatrix}
T_1 & D_3 \\
D_1 & T_2 & D_5 \\
D_2 & D_4 & B
\end{pmatrix}
\]

(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:

\[
call \text{ dsms(n,n,nzn,indrow,indcol,ngrp,maxgrp,mingrp,} \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{info,ipntr,jpnt1,} \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{nuw,iniw,bwu)}
\]

We are mainly interested in the first seven parameters of the calling sequence. The parameters \( nzn, \text{ indrow, } \text{ and indcol } \) 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).

\[
\begin{align*}
l &= n/3 \\
nnz &= 0 \\
do \ 60 & \ j = 1, n \\
nnz &= nnz + 1 \\
\text{indrow}(nnz) &= j \\
\text{indcol}(nnz) &= j \\
\text{if} \ (j \ .eq. \ l \ or \ j \ .eq. \ 2l \ or \ j \ .eq. \ 3l) \ \text{go to} \ 10 \\
nnz &= nnz + 1 \\
\text{indrow}(nnz) &= j + 1 \\
\text{indcol}(nnz) &= j \\
10 & \ \text{continue} \\
\text{if} \ (j \ .eq. \ l \ or \ j \ .eq. \ l+1 \ or \ j \ .gt. \ 2l) \ \text{go to} \ 20 \\
nnz &= nnz + 1 \\
\text{indrow}(nnz) &= j - 1 \\
\text{indcol}(nnz) &= j \\
20 & \ \text{continue} \\
\text{if} \ (j \ .gt. \ 2l) \ \text{go to} \ 30 \\
nnz &= nnz + 1 \\
\text{indrow}(nnz) &= j + l \\
\text{indcol}(nnz) &= j \\
30 & \ \text{continue} \\
\text{if} \ (j \ .le. \ l) \ \text{go to} \ 40 \\
nnz &= nnz + 1 \\
\text{indrow}(nnz) &= j - l \\
\text{indcol}(nnz) &= j \\
40 & \ \text{continue} \\
\text{if} \ (j \ .gt. \ l) \ \text{go to} \ 50 \\
nnz &= nnz + 1 \\
\text{indrow}(nnz) &= j + 2l \\
\text{indcol}(nnz) &= j \\
50 & \ \text{continue} \\
60 & \ \text{continue}
\end{align*}
\]

Program 4.1

Table 4.1 provides the output values of \( \text{mingrp} \) and \( \text{maxgrp} \) 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.

<table>
<thead>
<tr>
<th>( n )</th>
<th>nnz</th>
<th>mingrp</th>
<th>maxgrp</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>1295</td>
<td>5</td>
<td>6</td>
<td>1.25</td>
</tr>
<tr>
<td>600</td>
<td>2595</td>
<td>5</td>
<td>6</td>
<td>2.75</td>
</tr>
<tr>
<td>900</td>
<td>3895</td>
<td>5</td>
<td>6</td>
<td>3.60</td>
</tr>
<tr>
<td>1200</td>
<td>5195</td>
<td>5</td>
<td>6</td>
<td>4.93</td>
</tr>
</tbody>
</table>

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

Also note that \( \text{maxgrp} \) does not agree with \( \text{mingrp} \). In some cases it is not possible to determine a matrix \( A \) with \( \text{mingrp} \) 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
\[
\begin{align*}
\text{ngrp}(j) &= \text{mod}(j - 1.5) + 1, \quad 1 \leq j \leq l, \\
\text{ngrp}(j) &= \text{mod}(j - l + 1.5) + 1, \quad l < j \leq 2l, \\
\text{ngrp}(j) &= \text{mod}(j - 2l + 3.5) + 1, \quad 2l < j \leq 3l.
\end{align*}
\]

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 \( F: \mathbb{R}^n \rightarrow \mathbb{R}^n \) with components \( f_i: \mathbb{R}^n \rightarrow \mathbb{R} \) defined by

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

where \( \xi_k \) 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 \( \text{indcol} \) and \( \text{ipntr} \) a row-oriented definition of the sparsity pattern.

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

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,ipnt,func)
do 30 numgrp = 1, maxgrp
   do 10 j = 1, n
      d(j) = 0.0
      if (ngrp(j) .eq. numgrp) d(j) = 0.001
      x(j) = x(j) + d(j)
   continue
10 call fcn(n,xd,indcol,ipnt,fjac)
do 20 i = 1, n
   fjacd(i) = fjacd(i) - func(i)
20 continue
call fdjs(n,indrow,jpnt,ngp,numgrp,d,fjac,fjac)
30 continue

Program 4.3

   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 order-
ing of the columns and then calling the sequential algorithm to obtain the con-
sistent 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 parti-
tion 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 sub-
routines briefly.

   Subroutine SRTDAT permutes indrow and indcol so that indcol is in non-
decreasing order, and determines jpnt so that indrow and jpnt provide a
column-oriented definition of the sparsity pattern. This is done by first deter-
mining 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 nonzeros 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 nonzeros in the rows of $A$, then setting pointers to the start of the rows in $\text{indcol}$, and finally filling $\text{indcol}$. It is straightforward to show that the execution time for SETR is proportional to the number of nonzeros in $A$.

For ease of reference, we next provide 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 More' [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 $nkntr(\cdot)$ and $nghbr(\cdot)$ such that the vertices adjacent to the $j$-th vertex are

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

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 $2\tau$ words of storage where $\tau$ 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 = jptr(j), jptr(j+1)-1
      i = indrow(jp)
      do 10 ip = ipptr(i), ipptr(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 un-marks 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

\[ (6.1) \sum_{a_j = 0} \rho_i + deg(a_j). \]
where $\rho_i$ 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

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

Since

$$\sum_{j=1}^{n} \left( \sum_{a_{ij} \neq 0} \rho_i \right) = \sum_{i=1}^{n} \rho_i \cdot \sum_{j=1}^{n} \text{deg}(a_j) \leq \sum_{i=1}^{n} \rho_i \cdot \sum_{j=1}^{n} \text{deg}(a_j),$$

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 minimal 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 $\text{head}$, $\text{prev}$, and $\text{next}$. 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 $\text{deg}$ is $\text{head}(\text{deg})$ unless $\text{head}(\text{deg}) = 0$. In this case there are no columns in the $\text{deg}$ list. The column before $j$ in the degree list of column $j$ is $\text{prev}(j)$ unless $\text{prev}(j) = 0$. In this case $j$ is the first column in the degree list. The column after $j$ in the degree list of column $j$ is $\text{next}(j)$ unless $\text{next}(j) = 0$. 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 $\text{numdeg}$ is the degree of column $j$ then Program 6.3 deletes column $j$ from the $\text{numdeg}$ list and inserts it into the $\text{numdeg} + 1$ list.

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

\[(6.2) \quad \frac{1}{n} \sum_{i=1}^{n} \rho_i^2\]

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 $\min\gamma_p$ 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 $\rho_{\text{max}}$ where $\rho_{\text{max}}$ 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 $\text{mingrp}$ 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 $\text{list}$ by letting $\text{list}(k)$ be the $k$-th vertex in the ordering. On output from the sequential algorithm the array $\text{ngrp}$ specifies a consistent partition of the columns of $A$ by setting $\text{ngrp}(j)$ 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 $j = \text{list}(k)$ with Program 6.1.
  b) Un-mark the columns adjacent to column $\text{list}(k)$.
  c) Mark all the groups of the columns adjacent to column $\text{list}(k)$.
  d) Let $\text{ngrp}(\text{list}(k))$ 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 $\text{list}(k)$ 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 $\text{list}(k)$ 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 $\text{list}(k)$.

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 $\text{mingrp}$.

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 2880. In addition to the order $n$ of the matrix, Table 7.1 contains the number $\text{nnz}$ of nonzeros in the matrix, and the output values
for \textit{mingrp} and \textit{maxgrp}. Note that on 19 of the problems \textit{maxgrp} agrees with \textit{mingrp} and therefore DSM is optimal on these problems. DSM may still be optimal on the other problems because \textit{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 $n \geq 3$.

<table>
<thead>
<tr>
<th>n</th>
<th>nnz</th>
<th>mingrp</th>
<th>maxgrp</th>
</tr>
</thead>
<tbody>
<tr>
<td>59</td>
<td>267</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>66</td>
<td>320</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>72</td>
<td>222</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>87</td>
<td>541</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>102</td>
<td>1182</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>193</td>
<td>3493</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>198</td>
<td>1392</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>209</td>
<td>1743</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>221</td>
<td>1629</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>234</td>
<td>834</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>245</td>
<td>1481</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>307</td>
<td>2523</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>310</td>
<td>2448</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>346</td>
<td>3226</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>361</td>
<td>2953</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>419</td>
<td>3563</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>492</td>
<td>3156</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>503</td>
<td>6027</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>512</td>
<td>3502</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>592</td>
<td>5104</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>607</td>
<td>5131</td>
<td>14</td>
<td>16</td>
</tr>
<tr>
<td>758</td>
<td>5994</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>869</td>
<td>7285</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>878</td>
<td>7448</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>918</td>
<td>7304</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>992</td>
<td>16744</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>1005</td>
<td>8621</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>1007</td>
<td>8575</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>1242</td>
<td>10426</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>2880</td>
<td>25028</td>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.1 Output from DSM for Naval Problems.

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


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 precision subroutine; to obtain a double precision version it is only necessary to replace the single precision declaration by a double precision declaration.
**********

This is a test program for subroutines dsm and fdjs.
The test data represents a neutron kinetics problem.

**********

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), liwa(7200)
logical bwa(1200)
real dsm, errj, errmax, ffact, sum
real d(1200), ffac(6000), ffacd(1200), fvec(1200), x(1200), xd(1200)

Logical output unit is assumed to be number 6.

data nwrite /6/

tiwa = 7200

Test for dsm and fdjs.

write (nwrite,1000)
do 150 n = 300, 1200, 300
  write (nwrite,2000)

  Definition of sparsity pattern.

  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
  continue 10
    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
  continue 20
    if (j .gt. 2*l) go to 30
    nnz = nnz + 1
    indrow(nnz) = j + l
    indcol(nnz) = j
  continue 30
    if (j .le. l) go to 40
    nnz = nnz + 1
    indrow(nnz) = j - l
    indcol(nnz) = j
  continue 40
if (j .gt. l) go to 50
  nnz = nnz + 1
  indrow(nnz) = j + 2*n
  indcol(nnz) = j
50     continue
60     continue

c  Call dsm.

c  call dsm(n,n,nnz,indrow,indcol,ngrp,mxgrp,mngrp,
           info,ipntr,ipntr,wa,biwa,biwa)
if (info .le. 0) write (nwrite,4000) info

Statistics for the matrix.

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,mngrp,mxgrp

Test for fdjs.

do 80 j = 1, n
    x(j) = float(j)/float(n)
80     continue

call fcn(n,x,indcol,ipntr,fvec)

Approximate the Jacobian matrix.

do 110 numgrp = 1, mxgrp
    do 90 j = 1, n
        d(j) = 0.0
        if (ngrp(j) .eq. numgrp) d(j) = 0.001
        xid(j) = x(j) + d(j)
90     continue

call fcn(n,xid,indcol,ipntr,fjacd)
do 100 i = 1, n
    fjacid(i) = fjacid(i) - fvec(i)
100     continue

call fdjs(n,indrow,ipntr,ngrp,numgrp,d,fjacd,fjac)
continue

Test the approximation to the Jacobian.

errmax = 0.0
do 140 j = 1, n
    jpl = ipntr(j)
    jpu = ipntr(j+1) - 1
    do 130 jp = jpl, jpu
        t = indrow(jp)
130     continue
sum = 0.0
ipl = ipntr(i)
jp = ipntr(i+1) - 1
do 120 ip = ipl, jp
   k = indcol(ip)
   sum = sum + x(k)
   continue
120
   sum = sum + x(i)
fact = 1.0 + 2.0*sum
   if (i .eq. j) fact = 2.0*fact
   errij = fact(ip) - fact
   if (fact.gt.0.0) errij = errij/fact
   errmax = amaxi(errmax, errij)
   continue
130
   continue
140
   write (6,5000) errmax
150 continue
stop

Format statements.

1000 format(// ' Tests for dsm and fdjs - Neutron Kinetics Problem' //
   ' Statistics generated ' //
   ' n - number of columns '
   ' nnz - number of non-zero elements'
   ' dsm - 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, 'dsm', 5x,
   'minrow', 4x, 'maxrow', 4x, 'mingrp', 4x, 'mazgrp'//)
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)

Function subroutine for testing fdjs.

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

Last card of subroutine fcn.

end
subroutine dsm(m,n,npairs,indrow,indcol,ngrp,maxgrp,mingrp, info,jpntr,jpntr,liwa,liwa,bwa)
    !
    integer m,n,npairs,maxgrp,mingrp,info,liwa
    integer indrow(npairs),indcol(npairs),ngrp(n),
    ! jpntr(1),jpntr(1),liwa(liwa)
    logical bwa(n)

**********
subroutine dsm

The purpose of dsm is to determine an optimal or near-
optimal consistent partition of the columns of a sparse
m by n matrix A.

The sparsity pattern of the matrix A is specified by
the arrays indrow and indcol. On input the indices
for the non-zero elements of A are

    indrow(k),indcol(k), k = 1,2,...,npairs.

The (indrow,indcol) pairs may be specified in any order.
Duplicate input pairs are permitted, but the subroutine
eliminates them.

The subroutine partitions the columns of A into groups
such that columns in the same group do not have a
non-zero in the same row position. A partition of the
columns of A with this property is consistent with the
direct determination of A.

The subroutine statement is

    subroutine dsm(m,n,npairs,indrow,indcol,ngrp,maxgrp,mingrp, info,jpntr,jpntr,liwa,liwa,bwa)

where

m is a positive integer input variable set to the number
of rows of A.

n is a positive integer input variable set to the number
of columns of A.

npairs is a positive integer input variable set to the
number of (indrow,indcol) pairs used to describe the
sparsity pattern of A.

indrow is an integer array of length npairs. On input indrow
must contain the row indices of the non-zero elements of A.
On output indrow is permuted so that the corresponding
column indices are in non-decreasing order. The column
indices can be recovered from the array jpntr.

indcol is an integer array of length npairs. On input indcol
must contain the column indices of the non-zero elements of
A. On output indcol is permuted so that the corresponding
row indices are in non-decreasing order. The row indices
can be recovered from the array ipntr.

ngrp is an integer output array of length n which specifies
the partition of the columns of A. Column jcol belongs
to group ngrp(jcol).

mazgrp is an integer output variable which specifies the
number of groups in the partition of the columns of A.

mingrp is an integer output variable which specifies a lower
bound for the number of groups in any consistent partition
of the columns of A.

info is an integer output variable set as follows. For
normal termination info = 1. If m, n, or npairs is not
positive or liwa is less than \text{max}(m, n^2), then info = 0.
If the k-th element of indrow is not an integer between
1 and m or the k-th element of indcol is not an integer
between 1 and n, then info = -k.

ipntr is an integer output array of length m + 1 which
specifies the locations of the column indices in indcol.
The column indices for row i are

\text{indcol}(k), k \text{ = ipntr}(i), \ldots, \text{ipntr}(i+1)-1.

Note that ipntr(m+1)-1 is then the number of non-zero
elements of the matrix A.

jpntr is an integer output array of length n + 1 which
specifies the locations of the row indices in indrow.
The row indices for column j are

\text{indrow}(k), k \text{ = jpntr}(j), \ldots, \text{jpntr}(j+1)-1.

Note that jpntr(n+1)-1 is then the number of non-zero
elements of the matrix A.

iwa is an integer work array of length liwa.
iwa is a positive integer input variable not less than
\text{max}(m, n^2).
bwa is a logical work array of length n.

Subprograms called
MINPACK-supplied ... degr, ido, numsrt, seg, setr, slo, srtdat
FORTRAN-supplied ... max0
****

integer i,itr,j,jpl,jpu,k,mmaxq,nnz,numgrp

Check the input data.

info = 0
if (m .lt. 1 .or. n .lt. 1 .or. npairs .lt. 1 .or.
    itwa .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

Sort the data structure by columns.

call srdtad(n,npairs,indrow,indcol,jptr,iwa(1))

Compress the data and determine the number of
non-zero elements of A.

do 20 i = 1, m
   iwa(i) = 0
20 continue
nnz = 0

   do 70 j = 1, n
      jpl = jptr(j)
      jpu = jptr(j+1) - 1
      jptr(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 = jptr(j)
   do 50 jp = jpl, nnz
      ir = indrow(jp)
      iwa(ir) = 0
50 continue
60 continue
70 continue
   jptr(n+1) = nnz + 1

Extend the data structure to rows.

call srdtad(m,n,indrow,jptr,indcol,jptr,iwa(1))
Determine a lower bound for the number of groups.

```
mingrp = 0
do 80 i = 1, m
    mingrp = max0(mingrp, ipntr(i+1)-ipntr(i))
80 continue
```

Determine the degree sequence for the intersection graph of the columns of A.

call degr(n, indrow, ipntr, indcol, ipntr, iwa(5*n+1), iwa(n+1), bwa)

Color the intersection graph of the columns of A with the smallest-last (SL) ordering.

call slo(n, indrow, ipntr, indcol, ipntr, iwa(5*n+1), iwa(n+1), *maxclq,iwa(1),iwa(n+1),iwa(2*n+1),iwa(3*n+1),bwa)
call seq(n, indrow, ipntr, indcol, ipntr, iwa(4*n+1), ngrp, maxgrp, *iwa(n+1), bwa)
mingrp = max0(mingrp, maxclq)
if (maxgrp .eq. mingrp) go to 130

Color the intersection graph of the columns of A with the incidence-degree (ID) ordering.

call ido(m, n, indrow, ipntr, 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, ipntr, 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

Color the intersection graph of the columns of A with the largest-first (LF) ordering.

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, ipntr, 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
```

Exit from program.
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)
********

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.

In graph-theory terminology, the intersection graph of
the columns of A is the loopless graph G with vertices
a(j), j = 1, 2,...,n where a(j) is the j-th column of A
and with edge (a(i), a(j)) if and only if columns i and j
have a non-zero in the same row position.

Note that the value of m is not needed by degr and is
therefore not present in the subroutine statement.

The subroutine statement is

subroutine degr(n, indrow, jpntr, indcol, ipntr, ndeg, iwa, bwa)

where

n is a positive integer input variable set to the number
of columns of A.

indrow is an integer input array which contains the row
indices for the non-zeroes in the matrix A.

jpntr is an integer input array of length n + 1 which
specifies the locations of the row indices in indrow.
The row indices for column j are

indrow(k), k = jpntr(j),...jpntr(j+1)-1.

Note that jpntr(n+1)-1 is then the number of non-zero
elements of the matrix A.

indcol is an integer input array which contains the
column indices for the non-zeroes in the matrix A.

ipntr is an integer input array of length m + 1 which
specifies the locations of the column indices in indcol.
The column indices for row i are

indcol(k), k = ipntr(i),...ipntr(i+1)-1.

Note that ipntr(m+1)-1 is then the number of non-zero
elements of the matrix A.
ndeg is an integer output array of length n which specifies the degree sequence. The degree of the j-th column of A is ndeg(j).

tua is an integer work array of length n.

bua is a logical work array of length n.

Thomas F. Coleman, Burton S. Garbow, Jorge J. More

**********

integer deg,ic,ip,jpl,ipu,ir,jcol,jp,jpl,jpu

Initialization block.

do 10 jp = 1, n
   ndeg(jp) = 0
   bua(jp) = .false.
10   continue

Compute the degree sequence by determining the contributions to the degrees from the current(jcol) column and further columns which have not yet been considered.

if (n .lt. 2) go to 90
do 80 jcol = 2, n
   bua(jcol) = .true.
   deg = 0
80

Determine all positions (ir,jcol) which correspond to non-zeros in the matrix.

jpl = jpntr(jcol)
jpu = jpntr(jcol+1) - 1
if (jpu .lt. jpl) go to 50
do 40 jp = jpl, jpu
   tr = indrow(jp)
40

For each row ir, determine all positions (ir,ic) which correspond to non-zeros in the matrix.

ipl = ipntr(ir)
ipu = ipntr(ir+1) - 1
do 30 ip = ipl, ipu
   ic = indcol(ip)
30

Array bua marks columns which have contributed to the degree count of column jcol. Update the degree counts of these columns. Array iwa records the marked columns.

if (bua(ic)) go to 20
   bua(ic) = .true.
ndeg(ic) = ndeg(ic) + 1
deg = deg + 1
twa(deg) = ic
20  continue
30  continue
40  continue
50  continue

c  Un-mark the columns recorded by twa and finalize the
c  degree count of column jcol.

c  if (deg .lt. 1) go to 70
   do 60 jp = 1, deg
      ic = twa(jp)
      bwa(ic) = .false.
60  continue
   ndeg(jcol) = ndeg(jcol) + deg
70  continue
60  continue
90  continue
return

c  Last card of subroutine degr.

c  end
subroutine ido(m,n,indrow,jpmtr,indcol,jpmtr,ndeg,list,
  *  maxclq,iwa1,iwa2,iwa3,iwa4,bw
integer m,n,maxclq
integer indrow(1),jpmtr(1),indcol(1),jpmtr(1),ndeg(n),list(n),
  *  iwa1(n),iwa2(n),iwa3(n),iwa4(n)
logical bw(n)
********

subroutine ido

Given the sparsity pattern of an m by n matrix A, this
subroutine determines an incidence-degree ordering of the
columns of A.

The incidence-degree ordering is defined for the loopless
graph G with vertices a(j), j = 1,2,...,n where a(j) is the
j-th column of A and with edge (a(i),a(j)) if and only if
columns i and j have a non-zero in the same row position.

At each stage of ido, a column of maximal incidence is
chosen and ordered. If jcol is an un-ordered column, then
the incidence of jcol is the number of ordered columns
adjacent to jcol in the graph G. Among all the columns of
maximal incidence, ido chooses a column of maximal degree.

The subroutine statement is

    subroutine ido(m,n,indrow,jpmtr,indcol,jpmtr,ndeg,list,
      *  maxclq,iwa1,iwa2,iwa3,iwa4,bw)

where

m is a positive integer input variable set to the number
of rows of A.

n is a positive integer input variable set to the number
of columns of A.

indrow is an integer input array which contains the row
indices for the non-zeroes in the matrix A.

jpmtr is an integer input array of length n+1 which
specifies the locations of the row indices in indrow.
The row indices for column j are

  indrow(k), k = jpmtr(j)....jpmtr(j+1)-1.

Note that jpmtr(n+1)-1 is then the number of non-zero
elements of the matrix A.

indcol is an integer input array which contains the
column indices for the non-zeroes in the matrix A.

jpmtr is an integer input array of length m+1 which
specifies the locations of the column indices in indcol.  
The column indices for row i are

indcol(k), k = ipntr(i),...,ipntr(i+1)-1.

Note that ipntr(m+1)-1 is then the number of non-zero 
elements of the matrix A.

ndeg is an integer input array of length n which specifies 
the degree sequence. The degree of the j-th column 
of A is ndeg(j).

list is an integer output array of length n which specifies 
the incidence-degree ordering of the columns of A. The j-th 
column in this order is list(j).

maxclq is an integer output variable set to the size 
of the largest clique found during the ordering.

iwa1,iwa2,iwa3, and iwa4 are integer work arrays of length n.

bwa is a logical work array of length n.

Subprograms called :

MINPACK-supplied ... numsrt

FORTRAN-supplied ... max0

Thomas F. Coleman, Burton S. Garbow, Jorge J. More

***********
integer deg,head,ic,ip,ipl,iplu,ir,jcol,jp,jpl,jpu,l,maxin,
*  maxist,ncomp,numinc,numlist,numord,numugt

Sort the degree sequence.

call numsrt(n,n-1,ndeg,-1,iwa4,iwa1,iwa3)

Initialization block.

Create a doubly-linked list to access the incidences of the 
columns. The pointers for the linked list are as follows.

Each un-ordered column jcol is in a list (the incidence list) 
of columns with the same incidence.

iwa1(numinc+1) is the first column in the numinc list 
unless iwa1(numinc+1) = 0. In this case there are 
no columns in the numinc list.

iwa2(jcol) is the column before jcol in the incidence list 
unless iwa2(jcol) = 0. In this case jcol is the first
column in this incidence list.

If $jcol$ is an un-ordered column, then list($jcol$) is the incidence of $jcol$ in the graph. If $jcol$ is an ordered column, then list($jcol$) is the incidence-degree order of column $jcol$.

```
maxinc = 0
    do 10 j = 1, n
        list(j) = 0
        iwa(j) = .false.
        iwa1(j) = 0
        l = iwa4(j)
        if (j .ne. 1) iwa2(l) = iwa4(j-1)
        if (j .ne. n) iwa3(l) = iwa4(j+1).
          continue
        iwa1(l) = iwa4(1)
        l = iwa4(1)
        iwa2(l) = 0
        l = iwa4(n)
        iwa3(l) = 0

10  continue
```

Determine the maximal search length for the list of columns of maximal incidence.

```
maxlst = 0
    do 20 ir = 1, m
        maxlst = maxlst + (ipntr(ir+1) - ipntr(ir))**2
          continue
    maxlst = maxlst/n
    maxclq = 1
```

Beginning of iteration loop.

```
do 140 numord = 1, n
    Choose a column $jcol$ of maximal degree among the columns of maximal incidence.

    $jp = iwa1(maxinc+1)$
    numlst = 1
    numwgt = -1
      continue
      if (ndeg($jp$) .le. numwgt) go to 40
    numwgt = ndeg($jp$)
    jcol = $jp$
    continue
    $jp = iwa3(jp)$
    numlst = numlst + 1
    if ($jp$ .gt. 0 .and. numlst .le. maxlst) go to 30
    list($jcol$) = numord
```
Delete column jcol from the list of columns of maximal incidence.

\[ l = \text{iwa3}(jcol) \]
\[ \text{if (l .eq. 0) iwa1(maxinc+1) = iwa3(jcol)} \]
\[ \text{if (l .gt. 0) iwa3(l) = iwa3(jcol)} \]
\[ l = \text{iwa3}(jcol) \]
\[ \text{if (l .gt. 0) iwa2(l) = iwa2(jcol)} \]

Update the size of the largest clique found during the ordering.

\[ \text{if (maxinc .eq. 0) ncomp = 0} \]
\[ \text{ncomp = ncomp + 1} \]
\[ \text{if (maxinc + 1 .eq. ncomp) maxclq = max0(maxclq,ncomp)} \]

Update the maximal incidence count.

\[ 50 \text{ continue} \]
\[ \text{if (iwa1(maxinc+1) .gt. 0) go to 60} \]
\[ \text{maxinc = maxinc - 1} \]
\[ \text{if (maxinc .ge. 0) go to 50} \]
\[ 60 \text{ continue} \]

Find all columns adjacent to column jcol.

\[ \text{bwa}(jcol) = .true. \]
\[ \text{deg} = 0 \]

Determine all positions (ir,jcol) which correspond to non-zeroes in the matrix.

\[ jpl = \text{jpntr}(jcol) \]
\[ jpu = \text{jpntr}(jcol+1) - 1 \]
\[ \text{if (jpu .lt. jpl) go to 100} \]
\[ \text{do 90 fp = jpl, jpu} \]
\[ \text{ir = indcol(fp)} \]

For each row ir, determine all positions (ir,ic) which correspond to non-zeroes in the matrix.

\[ ipl = \text{ipntr}(ir) \]
\[ ipu = \text{ipntr}(ir+1) - 1 \]
\[ \text{do 80 ip = 1pl, ipu} \]
\[ \text{ic = indcol(ip)} \]

Array bwa marks columns which are adjacent to column jcol. Array iwa4 records the marked columns.

\[ \text{if (bwa(ic)) go to 70} \]
\[ \text{bwa(ic) = .true.} \]
\[ \text{deg = deg + 1} \]
\[ \text{iwa4(deg) = ic} \]
70         continue
80         continue
90         continue
100        continue

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)
110        continue

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) = iwa2(ic)
   l = iwa3(ic)
   if (l .gt. 0) iwa2(l) = iwa2(ic)

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
   continue

130        continue

c
Un-mark column ic in the array bwa.
c
   bwa(ic) = .false.
120        continue
130        continue
   bwa(jcol) = .false.

c
End of iteration loop.
c
140        continue

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

Last card of subroutine ido.

c

dend
subroutine numsrt(n,nmax,num,mode,index,last,next)
  integer n,nmax,mode
  integer num(n),index(n),last(1),next(n)

*********

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.

The sequence of integers is defined by the array num,
  and it is assumed that the integers are each from the set
  0,1,...,nmax. On output the indices k such that num(k) = l
  for any l = 0,1,...,nmax can be obtained from the arrays
  last and next as follows.

  k = last(l+1)
  while (k .ne. 0) k = next(k)

Optionally, the subroutine produces an array index so that
  the sequence num(index(i)), i = 1,2,...,n is sorted.

The subroutine statement is

  subroutine numsrt(n,nmax,num,mode,index,last,next)

where

n is a positive integer input variable.

nmax is a positive integer input variable.

num is an input array of length n which contains the
  sequence of integers to be grouped and sorted. It
  is assumed that the integers are each from the set
  0,1,...,nmax.

mode is an integer input variable. The sequence num is
  sorted in ascending order if mode is positive and in
  descending order if mode is negative. If mode is 0,
  no sorting is done.

index is an integer output array of length n set so
  that the sequence

  num(index(i)), i = 1,2,...,n

is sorted according to the setting of mode. If mode
  is 0, index is not referenced.

last is an integer output array of length nmax + 1. The
  index of num for the last occurrence of l is last(l+1)
for any \( l = 0.1, \ldots, n_{\text{max}} \) unless last\((l+1) = 0 \). In this case \( l \) does not appear in num.

next is an integer output array of length \( n \). If
num\((k) = l \), then the index of num for the previous occurrence of \( l \) is next\((k) \) for any \( l = 0.1, \ldots, n_{\text{max}} \) unless next\((k) = 0 \). In this case there is no previous occurrence of \( l \) in num.

Thomas F. Coleman, Burton S. Garbow, Jorge J. More

*******
integer i,j,jp,k,l,nmaxp1,nmaxp2

Determine the arrays next and last.

\[ n_{\text{maxp1}} = n_{\text{max}} + 1 \]
\[ \text{do} \ 10 \ i = 1, n_{\text{maxp1}} \]
\[ \text{last}(i) = 0 \]
\[ 10 \ \text{continue} \]
\[ \text{do} \ 20 \ k = 1, n \]
\[ l = \text{num}(k) \]
\[ \text{next}(k) = \text{last}(l+1) \]
\[ \text{last}(l+1) = k \]
\[ 20 \ \text{continue} \]
\[ \text{if (mode .eq. 0) go to 60} \]

Store the pointers to the sorted array in index.

\[ i = 1 \]
\[ n_{\text{maxp2}} = n_{\text{maxp1}} + 1 \]
\[ \text{do} \ 50 \ j = 1, n_{\text{maxp1}} \]
\[ \text{jp} = j \]
\[ \text{if (mode .lt. 0) } \text{jp} = n_{\text{maxp2}} - j \]
\[ k = \text{last(jp)} \]
\[ 30 \ \text{continue} \]
\[ \text{if (k .eq. 0) go to 40} \]
\[ \text{index}(k) = k \]
\[ i = i + 1 \]
\[ k = \text{next}(k) \]
\[ \text{go to 30} \]
\[ 40 \ \text{continue} \]
\[ 50 \ \text{continue} \]
\[ 60 \ \text{continue} \]
\[ \text{return} \]

Last card of subroutine numsrt.

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)
********

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.

A consistent partition is defined in terms of the loopless
graph G with vertices a(j), j = 1,2,...,n where a(j) is the
j-th column of A and with edge (a(i),a(j)) if and only if
columns i and j have a non-zero in the same row position.

A partition of the columns of A into groups is consistent
if the columns in any group are not adjacent in the graph G.
In graph-theory terminology, a consistent partition of the
columns of A corresponds to a coloring of the graph G.

The subroutine examines the columns in the order specified
by the array list, and assigns the current column to the
group with the smallest possible number.

Note that the value of m is not needed by seq and is
therefore not present in the subroutine statement.

The subroutine statement is

subroutine seq(n,indrow,jpntr,indcol,ipntr,list,ngrp,maxgrp,
   & iwa,bwa)

where

n is a positive integer input variable set to the number
of columns of A

indrow is an integer input array which contains the row
indices for the non-zeroes in the matrix A

jpntr is an integer input array of length n + 1 which
specifies the locations of the row indices in indrow.
The row indices for column j are

indrow(k), k = jpntr(j),...jpntr(j+1)-1.

Note that jpntr(n+1)-1 is then the number of non-zero
elements of the matrix A.

indcol is an integer input array which contains the
column indices for the non-zeros in the matrix A.

ipntr is an integer input array of length m + 1 which
specifies the locations of the column indices in indcol.
The column indices for row i are

\[ \text{indcol}(k), \ k = \text{ipntr}(i), \ldots, \text{ipntr}(i+1)-1. \]

Note that ipntr(m+1)-1 is then the number of non-zero
elements of the matrix A.

list is an integer input array of length n which specifies
the order to be used by the sequential algorithm.
The j-th column in this order is list(j).

ngrp is an integer output array of length n which specifies
the partition of the columns of A. Column jcol belongs
to group ngrp(jcol).

maxgrp is an integer output variable which specifies the
number of groups in the partition of the columns of A.

twa is an integer work array of length n.

twu is a logical work array of length n.

**Argonne National Laboratory, MINPACK Project, June 1982.**
Thomas F. Coleman, Burton S. Garbow, Jorge J. More

***********

t:integer, io, ip, ipl, iplu, ir, j, jcol, jp, jpl, jpu, l, numgrp

Initialization block.

maxgrp = 0
do 10 jp = 1, n
  ngrp(jp) = n
  twu(jp) = .false.
10 continue
  twu(n) = .true.

Beginning of iteration loop.

do 100 j = 1, n
  jcol = list(j)

  Find all columns adjacent to column jcol.

  deg = 0

  Determine all positions (ir,jcol) which correspond
to non-zeros in the matrix.

  jpl = jpntr(jcol)
\( jpu = jptr(jcol+1) - 1 \)
if (jpu .lt. jpl) go to 50
do 40 jp = jpl, jpu
   ir = indrow(jp)
   For each row \(ir\), determine all positions \((ir, ic)\)
   which correspond to non-zeroes in the matrix.
   ipl = ipntr(ir)
tpu = ipntr(ir+1) - 1
do 30 ip = ipl, tpu
   ic = indcol(ip)
l = ngrp(ic)

Array \(bwa\) marks the group numbers of the
   columns which are adjacent to column \(jcol\).
Array \(twa\) records the marked group numbers.
if (bwa(l)) go to 20
   bwa(l) = .true.
deg = deg + 1
twa(ddeg) = l
20   continue
30   continue
40   continue
50   continue

Assign the smallest un-marked group number to \(jcol\).
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)
   continue

Un-mark the group numbers.
if (deg .lt. 1) go to 90
do 80 jp = 1, deg
   l = twa(jp)
   bwa(l) = .false.
80   continue
90   continue
100  continue

End of iteration loop.
return

Last card of subroutine \(seg\).
end
subroutine setr(m,n,indrow,jpntr,indcol,ipntr,iwa)
integer m,n
integer indrow(1),jpntr(1),indcol(1),ipntr(1),iwa(m)
**********

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.

On input the column-oriented definition is specified by
the arrays indrow and jpntr. On output the row-oriented
definition is specified by the arrays indcol and ipntr.

The subroutine statement is

subroutine setr(m,n,indrow,jpntr,indcol,ipntr,iwa)

where

m is a positive integer input variable set to the number
of rows of A.

n is a positive integer input variable set to the number
of columns of A.

indrow is an integer input array which contains the row
indices for the non-zeroes in the matrix A.

jpntr is an integer input array of length n + 1 which
specifies the locations of the row indices in indrow.
The row indices for column j are

indrow(k), k = jpntr(j), ..., jpntr(j+1)-1.

Note that jpntr(n+1)-1 is then the number of non-zero
elements of the matrix A.

indcol is an integer output array which contains the
column indices for the non-zeroes in the matrix A.

ipntr is an integer output array of length m + 1 which
specifies the locations of the column indices in indcol.
The column indices for row i are

indcol(k), k = ipntr(i), ..., ipntr(i+1)-1.

Note that ipntr(1) is set to 1 and that ipntr(m+1)-1 is
then the number of non-zero elements of the matrix A.

iwa is an integer work array of length m.

**fortran**

```fortran
integer ir,jcol,jp,jpl,jpu,l,nnz

Determine the number of non-zeroes in the rows.

\[ \text{do } 10 \text{ ir } = 1, m \]
\[ \text{tiwa}(\text{ir}) = 0 \]
10 continue
\[ \text{nnz } = \text{jpnntr}(n+1) - 1 \]
\[ \text{do } 20 \text{ jp } = 1, \text{nnz} \]
\[ \text{ir } = \text{indrow}(\text{jp}) \]
\[ \text{tiwa}(\text{ir}) = \text{tiwa}(\text{ir}) + 1 \]
20 continue

Set pointers to the start of the rows in indcol.

\[ \text{ipnntr}(1) = 1 \]
\[ \text{do } 30 \text{ ir } = 1, m \]
\[ \text{ipnntr}(\text{ir}+1) = \text{ipnntr}(\text{ir}) + \text{tiwa}(\text{ir}) \]
\[ \text{tiwa}(\text{ir}) = \text{ipnntr}(\text{ir}) \]
30 continue

Fill indcol.

\[ \text{do } 60 \text{ jcol } = 1, n \]
\[ \text{jpl } = \text{jpnntr}(\text{jcol}) \]
\[ \text{jpu } = \text{jpnntr}(\text{jcol}+1) - 1 \]
\[ \text{if } (\text{jpu} .lt. \text{jpl}) \text{ go to } 50 \]
\[ \text{do } 40 \text{ jp } = \text{jpl}, \text{jpu} \]
\[ \text{ir } = \text{indrow}(\text{jp}) \]
\[ \text{indcol}(\text{ir}) = \text{jcol} \]
\[ \text{tiwa}(\text{ir}) = \text{tiwa}(\text{ir}) + 1 \]
40 continue
50 continue
60 continue

return

Last card of subroutine setr.

end
```
subroutine slo(n,indrow,jpnnr,indcol,ipntr,ndeg,list,
  iaxclq,iwa1,iwa2,iwa3,iwa4,ifa)
integer n,maclq
integer indrow(1),jpnnr(1),indcol(1),ipntr(1),ndeg(n),
  list(n),iwa1(n),iwa2(n),iwa3(n),iwa4(n)
logical bwa(n)
**********
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.

The smallest-last ordering is defined for the loopless
graph G with vertices a(j), j = 1, 2, ..., n where a(j) is the
j-th column of A and with edge (a(i),a(j)) if and only if
columns i and j have a non-zero in the same row position.

The smallest-last ordering is determined recursively by
letting list(k), k = n, ..., 1 be a column with least degree
in the subgraph spanned by the un-ordered columns.

Note that the value of m is not needed by slo and is
therefore not present in the subroutine statement.

The subroutine statement is
subroutine slo(n,indrow,jpnnr,indcol,ipntr,ndeg,list,
  iaxclq,iwa1,iwa2,iwa3,iwa4,ifa)

where

n is a positive integer input variable set to the number
of columns of A.

indrow is an integer input array which contains the row
indices for the non-zeroes in the matrix A.

jpnnr is an integer input array of length n + 1 which
specifies the locations of the row indices in indrow.
The row indices for column j are

  indrow(k), k = jpnnr(j), ...,jpnnr(j+1)-1.

Note that jpnnr(n+1)-1 is then the number of non-zero
elements of the matrix A.

indcol is an integer input array which contains the
column indices for the non-zeroes in the matrix A.

ipntr is an integer input array of length m + 1 which
specifies the locations of the column indices in indcol.
The column indices for row i are
\[ \text{indcol}(k), \quad k = \text{ipntr}(i), \ldots, \text{ipntr}(i+1)-1. \]

Note that \( \text{ipntr}(m+1)-1 \) is then the number of non-zero elements of the matrix \( A \).

\( n \text{deg} \) is an integer input array of length \( n \) which specifies the degree sequence. The degree of the \( j \)-th column of \( A \) is \( \text{ndeg}(j) \).

\( \text{list} \) is an integer output array of length \( n \) which specifies the smallest-last ordering of the columns of \( A \). The \( j \)-th column in this order is \( \text{list}(j) \).

\( \text{maxclq} \) is an integer output variable set to the size of the largest clique found during the ordering.

\( \text{iwa1}, \text{iwa2}, \text{iwa3}, \) and \( \text{iwa4} \) are integer work arrays of length \( n \).

\( \text{bwa} \) is a logical work array of length \( n \).

Subprograms called

FORTRAN-supplied ... \( \min 0 \)


**********

integer \( \text{deg}, \text{head}, \text{ic}, \text{ip}, \text{ipl}, \text{iplu}, \text{ir}, \text{jcol}, \text{jp}, \text{jpl}, \text{jpu} \),

* \( \quad \text{l}, \text{mindeg}, \text{numdeg}, \text{numord} \)

Initialization block.

\( \text{mindeg} = n \)

\[ \text{do} 10 \text{jp} = 1, n \]

\( \quad \text{iwa1}(\text{jp}) = 0 \)

\( \quad \text{bwa}(\text{jp}) = .false. \)

\( \quad \text{list}(\text{jp}) = \text{ndeg}(\text{jp}) \)

\( \quad \text{mindeg} = \min(\text{mindeg}, \text{ndeg}(\text{jp})) \)

\( 10 \quad \text{continue} \)

Create a doubly-linked list to access the degrees of the columns. The pointers for the linked list are as follows.

Each un-ordered column \( \text{jcol} \) is in a list (the degree list) of columns with the same degree.

\( \text{iwa1}(\text{numdeg}+1) \) is the first column in the \( \text{numdeg} \) list unless \( \text{iwa1}(\text{numdeg}+1) = 0 \). In this case there are no columns in the \( \text{numdeg} \) list.

\( \text{iwa2}(\text{jcol}) \) is the column before \( \text{jcol} \) in the degree list unless \( \text{iwa2}(\text{jcol}) = 0 \). In this case \( \text{jcol} \) is the first
column in this degree list.

iwa3(jcol) is the column after jcol in the degree list unless iwa3(jcol) = 0. In this case jcol is the last column in this degree list.

If jcol is an un-ordered column, then list(jcol) is the degree of jcol in the graph induced by the un-ordered columns. If jcol is an ordered column, then list(jcol) is the smallest-last order of column jcol.

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
  continue
20  maxclq = 0
  numord = n

Beginning of iteration loop.

30 continue

Mark the size of the largest clique found during the ordering.

if (mindeg + 1 .eq. numord .and. maxclq .eq. 0)
  maxclq = numord

Choose a column jcol of minimal degree mindeg.

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

Termination test.

if (numord .eq. 0) go to 120

Delete column jcol from the mindeg list.

l = iwa3(jcol)
iwa1(mindeg+1) = l
if (l .gt. 0) iwa2(l) = 0

Find all columns adjacent to column jcol.
bwa(jcol) = .true.
deg = 0

Determine all positions (ir,jcol) which correspond to non-zeroes in the matrix.

jpl = jpntr(jcol)
jpu = jpntr(jcol+1) - 1
if (jpu .lt. jpl) go to 90
do 80 jp = jpl, jpu
   ir = indrow(jp)
   For each row ir, determine all positions (ir,ic)
   which correspond to non-zeroes in the matrix.
  80 continue

ipl = ipntr(ir)
ipu = ipntr(ir+1) - 1
do 70 ip = ipl, ipu
   ic = indcol(ip)

Array bwa marks columns which are adjacent to column jcol. Array iwa4 records the marked columns.

if (bwa(ic)) go to 60
bwa(ic) = .true.
deg = deg + 1
iwa4(deg) = ic

60 continue

70 continue
80 continue
90 continue

Update the pointers to the current degree lists.

if (deg .lt. 1) go to 110
do 100jp = 1, deg
   ic = iwa4(jp)
   numdeg = list(ic)
   list(ic) = list(ic) - 1
   mindeg = min0(mindeg, list(ic))

Delete column ic from the numdeg list.

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)

Add column ic to the numdeg-1 list.

head = iwa1(numdeg)
iwa1(numdeg) = ic
twa2(ic) = 0
twa3(ic) = head
if (head .gt. 0) twa2(head) = ic

Un-mark column ic in the array bwa.

bwa(ic) = .false.
continue
100 continue
110 continue

End of iteration loop.

go to 30
120 continue

Invert the array list.

do 130 jcol = 1, n
numord = list(jcol)
twa1(numord) = jcol
130 continue
do 140 jp = 1, n
list(jp) = twa1(jp)
140 continue
return

Last card of subroutine slo.

end
subroutine srtdat(n,nnz,indrow,indcol,jpntr,iwa)
integer n,nnz
integer indrow(nnz),indcol(nnz),jpntr(1),iwa(n)
**********

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.

On input it is assumed that the elements are specified in

indrow(k),indcol(k), k = 1,...,nnz.

On output the elements are permuted so that indcol is
in non-decreasing order. In addition, the array jpntr
is set so that the row indices for column j are

indrow(k), k = jpntr(j),...,jpntr(j+1)-1.

Note that the value of m is not needed by srtdat and is
therefore not present in the subroutine statement.

The subroutine statement is

subroutine srtdat(n,nnz,indrow,indcol,jpntr,iwa)

where

n is a positive integer input variable set to the number
of columns of A.

nnz is a positive integer input variable set to the number
of non-zero elements of A.

indrow is an integer array of length nnz. On input indrow
must contain the row indices of the non-zero elements of A.
On output indrow is permuted so that the corresponding
column indices of indcol are in non-decreasing order.

indcol is an integer array of length nnz. On input indcol
must contain the column indices of the non-zero elements
of A. On output indcol is permuted so that these indices
are in non-decreasing order.

jpntr is an integer output array of length n + 1 which
specifies the locations of the row indices in the output
indrow. The row indices for column j are

indrow(k), k = jpntr(j),...,jpntr(j+1)-1.

Note that jpntr(1) is set to 1 and that jpntr(n+1)-1
is then nnz.

iwa is an integer work array of length n.

Subprograms called

FORTRAN-supplied ... max0

Thomas F. Coleman, Burton S. Garbow, Jorge J. More

**********

integer i,j,k,l

Determine the number of non-zeros in the columns.

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

Set pointers to the start of the columns in indrow.

jpnt(1) = 1
do 30 j = 1, n
   jpnt(j+1) = jpnt(j) + iwa(j)
   iwa(j) = jpnt(j)
30 continue
   k = 1

Begin in-place sort.

40 continue
   j = indcol(k)
   if (k .lt. jpnt(j) .or. k .ge. jpnt(j+1)) go to 50
   Current element is in position. Now examine the
   next element or the first un-sorted element in
   the j-th group.

   k = max0(k+1,iwa(j))
   go to 60
50 continue
   Current element is not in position. Place element
   in position and make the displaced element the
   current element.
   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

Last card of subroutine srtdat.

end
subroutine fdjs(n,indrow,fpntr,ngrp,numgrp,d,fjacd,fjac)
integer n,numgrp
integer indrow(1),fpntr(1),ngrp(n)
real d(n),fjacd(1),fjac(1)

*********

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.

A partition is consistent if the columns in any group
do not have a non-zero in the same row position.

Approximations to the columns of the Jacobian matrix in a
given group can be obtained by specifying a difference
parameter array d with d(jcol) non-zero if and only if
jcol is a column in the group, and an approximation to
d(*) where jac denotes the Jacobian matrix of a mapping F.

d can be defined with the following segment of code.

    do 10 jcol = 1, n
        d(jcol) = 0.0
        if (ngrp(jcol) .eq. numgrp) d(jcol) = eta(jcol)
    continue

In the above code numgrp is the given group number,
ngrp(jcol) is the group number of column jcol, and
eta(jcol) is the difference parameter used to
approximate column jcol of the Jacobian matrix.
Suitable values for the array eta must be provided.

As mentioned above, an approximation to jac*d must
also be provided. For example, the approximation

        F(x+d) - F(x)

corresponds to the forward difference formula at x.

Note that the value of m is not needed by fdjs and is
therefore not present in the subroutine statement.

The subroutine statement is

    subroutine fdjs(n,indrow,fpntr,ngrp,numgrp,d,fjacd,fjac)

where

    n is a positive integer input variable set to the number
    of columns of the Jacobian matrix.

    indrow is an integer input array which contains the row
indices for the non-zeroes in the Jacobian matrix.

jpntr is an integer input array of length n + 1 which
specifies the locations of the row indices in indrow.
The row indices for column j are

indrow(k), k = jpntr(j),...,jpntr(j+1)-1.

Note that jpntr(n+1)-1 is then the number of non-zero
elements of the Jacobian matrix.

ngrp is an integer input array of length n which specifies
the partition of the columns of the Jacobian matrix.
Column jcol belongs to group ngrp(jcol).

numgrp is a positive integer input variable set to a group
number in the partition. The columns of the Jacobian
matrix in this group are to be estimated on this call.

d is an input array of length n which contains the
difference parameter vector for the estimate of
the Jacobian matrix columns in group numgrp.

fjacd is an input array of length m which contains
an approximation to the difference vector jac*d,
where jac denotes the Jacobian matrix.

fjac is an output array of length nnz, where nnz is the
number of its non-zero elements. At each call of fdjs,
fjac is updated to include the non-zero elements of the
Jacobian matrix for those columns in group numgrp. fjac
should not be altered between successive calls to fdjs.

Argonne National Laboratory. MINPACK Project, June 1982.
Thomas F. Coleman, Burton S. Garbow, Jorge J. More

**********
integer ir,jcol,jp,jpl,jpu

Compute estimates of Jacobian matrix columns in group
numgrp. The array fjacd must contain an approximation
to jac*d, where jac denotes the Jacobian matrix and d
is a difference parameter vector with d(jcol) non-zero
if and only if jcol is a column in group numgrp.

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 fp = jpl, jpu
      tr = indrow(fp)
      fjac(fp) = fjacd(tr)/d(jcol)
      continue
20 continue
30 continue
         return

C C Last card of subroutine fdjs.
C C end
Distribution for ANL-82-37

Internal:

K. L. Kliewer
A. B. Krisciunas
P. C. Messina
J. J. More' (63)
D. M. Pahis
L. M. Phebus (2)
G. W. Pieper
ANL Patent Department
ANL Contract File
ANL Libraries
TIS Files (6)

External:

DOE-TIC, for distribution per UC-32 (181)
Manager, Chicago Operations Office, DOE
President, Argonne Universities Association
D. Austin, BES-DOE
J. C. Browne, U. Texas at Austin
T. F. Coleman, Cornell U. (20)
S. Gerhart, Software Research Associates
W. C. Lynch, Xerox Corp.
J. M. Ortega, U. Virginia
F. H. Westervelt, AUA Trustee, Wayne State U.
M. F. Wheeler, Rice U.