JCL

Library for Reducing Bandwidth and Profile of Sparse Matrix

Feb. 11, 2008

LIST OF CONTENTS

Chapter 1. Introduction 1
1.1 Non-zero fill-ins and Non-zero coefficients 2

Chapter 2. Calling Syntax 3
2.1 A Fortran Example 4
2.2 A Bad Example 7

Chapter 3. Renumbering Before the Determination of Sparse Matrix 10

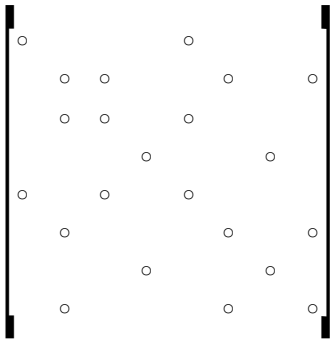
Chapter 4. Renumbering Asymmetric Sparse Matrix 14

Chapter 5. Data Storage Schemes 15
5.1 Constant Bandwidth Sparse Symmetric Matrix 15
5.2 Variable Bandwidth Sparse Symmetric Matrix 16

Chapter 1. Introduction

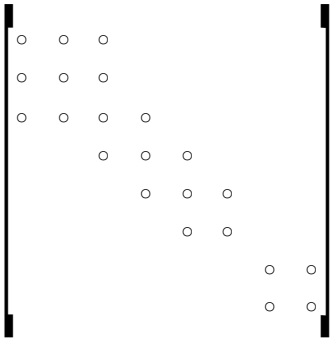
JCL is a Fortran library for reducing bandwidth and profile of sparse matrix. JCL is also callable by C language. JCL applies the algorithm, "*Algorithms for reducing the bandwidth and profile of a sparse matrix*", authored by J.-C. Luo, published in *computers & structures*, vol. 44, no. 3, 1992. The algorithm has been proved to be the most reliable.

Most scientific and engineering problems are formulated into a symmetric system of linear/non-linear equations, e. g., $[A]\{X\}=\{B\}$. Because of piecewise approximation, the symmetric matrix $[A]$ may lead to a sparse figure, e. g.,



(1.1)

where ◦ represents a non-zero fill-in, and other blanks represent zero coefficient. An arbitrary symmetric sparse matrix, e. g., the above example, cannot take advantage of sparse solvers. JCL renumbers the unknowns, e. g., rearrangement of rows and columns, such that non-zero fill-ins are around the diagonal as:



(1.2)

The sparse matrix, as shown in Eq. (1.2), then can be solved by constant bandwidth solvers or variable bandwidth solvers. Variable bandwidth solver is also called skyline

solver.

The cost to solve [A]{X}={B} depends on bandwidth (or half bandwidth) and profile. Half bandwidth is defined as the maximal $|i-j|$ where i and j are subscripts of non-zero fill-in A_{ij} or A_{ji}, and profile is defined as $\sum_{i=1}^{n} max\,of\,|i-j|$ where i and j are subscripts of non-zero fill-in A_{ji}. Renumbering can significantly reduce solution costs.

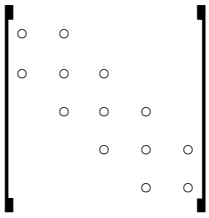
JCL provides an efficient and reliable way to reduce bandwidth and profile of a sparse matrix. JCL is thread-safe, and can be called simultaneously in a program.

1.1 Non-zero fill-ins and Non-zero coefficients

Non-zero fill-in is different from non-zero coefficient. For the following example,

$$[A]=\begin{bmatrix} 1.0 & 2.0 & & & & \\ 3.0 & 4.0 & 5.0 & & & \\ & 5.0 & 6.0 & 7.0 & & \\ & & 7.0 & 8.0 & 9.0 & \\ & & & 9.0 & 1.0 & \end{bmatrix}$$

The matrix has 13 non-zero coefficients. The non-zero coefficients are asymmetric because of $A_{12} \neq A_{21}$. Non-zero fill-in is an entry with non-zero coefficient. The non-zero fill-ins of matrix [A] are represented as follows:



In the above example, non-zero coefficients are asymmetric, but the non-zero fill-ins are symmetric. Once non-zero fill-ins are symmetric, JCL is applicable.

Chapter 2. Calling Syntax

JCL is programmed in Fortran. The entry subroutine to JCL is defined as follows:

```
jcl(nas,n,adj,output,work,limit,flag)
```

where

1. *nas*: 4-byte integer array, dimension(*n*), input the number of vertices adjacent to each vertex. Concept of adjacent vertices comes from graph theory by which the renumbering algorithms are developed. In finite element, adjacent vertices are similar to connected degrees of freedom. In algebra, adjacent vertices are similar to coupled unknowns.
2. *n*: a 4-byte integer, input the number of unknowns, or system size, or order.
3. *adj*: 4-byte integer array, dimension(sum of values in array *nas*), input the vertices adjacent to each vertex. E. g., vertices adjacent to the first vertex, and then vertices adjacent to the second vertex, and so on. The required length of array *adj* is equal to the sum of each value in array *nas*.
4. *output*: 4-byte integer array, dimension(*n*), returns the new number of vertices.
5. *work*: 4-byte integer array, dimension(*limit*), is a working array. This array should have a sufficient space.
6. *limit*: a 4-byte integer, input the dimension of array *work*. The dimension cannot be determined in advance because of the nature of algorithm. An estimation is as:

   ```
   13*n+(sum of values in array nas)
   ```

   JCL is a memory eater, but is not a CPU time eater. Computer memory is a not an issue on modern computer. The estimation can fit for most problems.
7. *flag*: 4-byte integer, returns the status
   - = 1 , successful entry. The array *output* has new number of vertices.
   - = 0 , the limit of array *work* is insufficient. User can re-run JCL with a big dimension of *work*.

$= -i$, incorrect adjacent vertices, where *i* is an integer value. User should check vertices adjacent to the vertex "*i*" and vertices adjacent to adjacent vertices of vertex "*i*".

2.1 A Fortran Example

Let us consider the 8×8 example as shown in Eq. (1.1). The information of adjacent vertices can be summarized as:

Vertex	***Adjacent Vertices***	***Number of Adjacent Vertices***
1	5	1
2	3,6,8	3
3	2,5	2
4	7	1
5	1,3	2
6	2,8	2
7	4	1
8	2,6	2

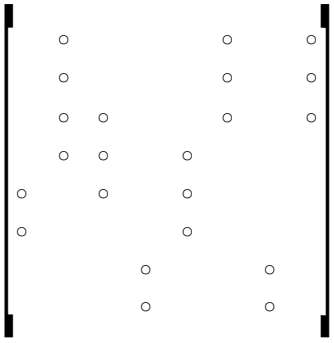
The following is a Fortran example showing how to call JCL.

```
   integer (4), parameter :: n = 8
   integer (4) :: nas(n)
   integer (4) :: output(n)
   integer (4) :: adj(14) !! why 14?
                          !! see program segment of
                          !! "number of adjacent vertices"
   integer (4), parameter :: limit = 13*n+14
                             !! JCL is memory eater
   integer (4) :: work(limit)
   integer (4) :: flag

!
!  number of adjacent vertices
!  by the 3rd column of above table
!
```

```
   nas(1) = 1
   nas(2) = 3
   nas(3) = 2
   nas(4) = 1
   nas(5) = 2
   nas(6) = 2
   nas(7) = 1
   nas(8) = 2  !! sum of values = 14
               !! adj is declared as dimension(14)

!
!  adjacent vertices
!  by the 2nd column of the above table
!
   !! vertex adjacent to vertex 1
   adj(1) = 5

   !! vertices adjacent to vertex 2
   adj(2) = 3
   adj(3) = 6
   adj(4) = 8

   !! vertices adjacent to vertex 3
   adj(5) = 2
   adj(6) = 5

   !! vertex adjacent to vertex 4
   adj(7) = 7

   !! vertices adjacent to vertex 5
   adj(8) = 1
   adj(9) = 3

   !! vertices adjacent to vertex 6
   adj(10) = 2
   adj(11) = 8

   !! vertex adjacent to vertex 7
   adj(12) = 4

   !! vertices adjacent to vertex 8
   adj(13) = 2
   adj(14) = 6

!
!  renumber by JCL
!
   call jcl(nas,n,adj,output,work,limit,flag)

!
!  output the new number
!
   write(*,*) "flag",flag
   if(flag.eq.1) write(*,*) output
```

```
!
!  end of program
!
   end
```

The output of new order is: `6 3 4 8 5 2 7 1`. That means the vertex 1 should be numbered as 6, and vertex 2 should be numbered as 3, and so on. By the new order, user can copy rows of the original matrix in Eq. (1.1) to a new matrix. For example, user can copy row 1 to row 6 of new matrix, and similarly copy row 2 to row 3, and so on. The row arrangement gets the following sparse figure:



(1.3)

Then, by the new order (6 3 4 8 5 2 7 1), user can copy columns in Eq. (1.3) to a new matrix. For example, copy column 1 to column 6 of new matrix, and similarly copy column 2 to column 3, and so on. Arrangement of columns of the matrix in Eq. (1.3) gets the perfect sparse figure as shown in Eq. (1.2).

The above example shows that bandwidth and profile of a sparse matrix are reduced by rearranging rows and columns of matrix [A]. Renumbering after the matrix [A] is determined is less efficient and less popular. In most scientific and engineering applications, vertices are renumbered before a sparse matrix is determined. The next Chapter will illustrate how to renumber vertices without matrix [A].

2.2 A Bad Example

Let us use the previous 8×8 example to illustrate how to find error in input data.

The information of adjacent vertices is as follows:

Vertex	*Adjacent Vertices*	*Number of Adjacent Vertices*
1	5	1
2	3,6,8	3
3	2,5	2
4	7	1
5	1,3	2
6	2,8	2
7	4	1
8	2,6	2

The vertex 6 has adjacent vertices 2 and 8. Here on purpose input incorrect adjacent vertices 3 and 8. The example program is as follow:

```
   integer (4), parameter :: n = 8
   integer (4) :: nas(n)
   integer (4) :: output(n)
   integer (4) :: adj(14)
   integer (4), parameter :: limit = 13*n+14
   integer (4) :: work(limit)
   integer (4) :: flag

!
!  number of adjacent vertices
!
   nas(1) = 1
   nas(2) = 3
   nas(3) = 2
   nas(4) = 1
   nas(5) = 2
   nas(6) = 2
   nas(7) = 1
   nas(8) = 2

!
!  adjacent vertices
!
   !! vertex adjacent to vertex 1
   adj(1) = 5

   !! vertices adjacent to vertex 2
```

```
      adj(2) = 3
      adj(3) = 6
      adj(4) = 8

      !! vertices adjacent to vertex 3
      adj(5) = 2
      adj(6) = 5

      !! vertex adjacent to vertex 4
      adj(7) = 7

      !! vertices adjacent to vertex 5
      adj(8) = 1
      adj(9) = 3

      !! vertices adjacent to vertex 6
      adj(10) = 3 !! (error: It should be 2)
      adj(11) = 8

      !! vertex adjacent to vertex 7
      adj(12) = 4

      !! vertices adjacent to vertex 8
      adj(13) = 2
      adj(14) = 6

!
!  renumber by JCL
!
      call jcl(nas,n,adj,output,work,limit,flag)

!
!  output the new number
!
      write(*,*) "flag",flag
      if(flag.eq.1) write(*,*) output

!
!  end of program
!
      end
```

The output *flag* is *-2*. That gives a hint that an error may happen to the vertices adjacent to vertex 2 or happens to the vertices adjacent to vertices adjacent to vertex 2. User needs to check the following:

1) First, check vertices adjacent to vertex 2. Vertices adjacent to vertex 2 includes 3, 6, and 8. Data input to JCL is 3, 6, and 8. The input is correct. Then, check vertices adjacent to 3, 6, and 8.
2) Vertices adjacent to vertex 3 are 2 and 5. Data input to JCL is 2 and 5. It is correct.

3) Vertices adjacent to vertex 6 are 2 and 8. However, Data input to JCL is 3 and 8. We find the error.

The return *flag* gives a hint to find an input error.

Chapter 3. Renumbering Before the Determination of Sparse Matrix

This Chapter illustrates how to renumber vertices before the determination of matrix [A]. Renumbering is important in scientific and engineering computing. Most scientific and engineering problems can be formulated into a system of equations, i.e., $[A]\{X\}=\{B\}$ where matrix $[A]$ is symmetric and sparse. The unknowns can be renumbered before the determination of matrix $[A]$. For example, let us consider the engineering problem, as shown in Figure 1.

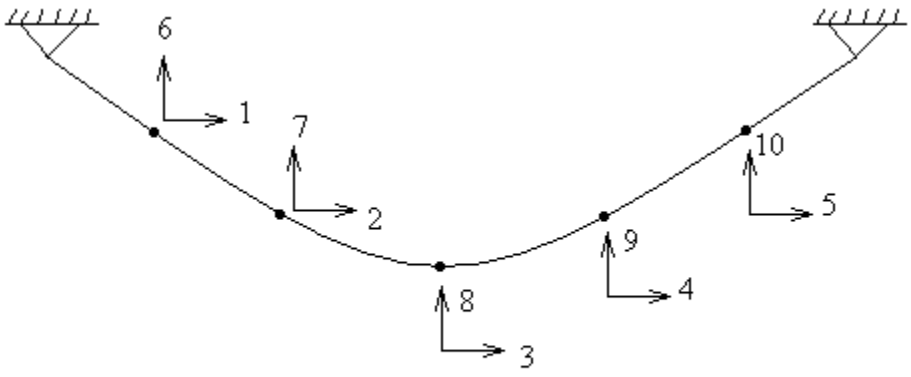


Figure 1

It is a cable. The example solves horizontal and vertical displacements at five joints, each of which has horizontal and vertical displacements. Each displacement is equivalent to a vertex in renumbering algorithms. This example has a total of 10 vertices, and are initially labeled from 1 to 10 in Figure 1. We won't use the initial number to solve displacements, but use the initial number to define adjacent vertices. The adjacent vertices are required by JCL. By Figure 1, we can define the adjacent vertices as:

Vertex	***Adjacent Vertices***	***Number of Adjacent Vertices***
1	2,6,7	3
2	1,3,6,7,8	5
3	2,4,7,8,9	5

Vertex	*Adjacent Vertices*	*Number of Adjacent Vertices*
4	3,5,8,9,10	5
5	4,9,10	3
6	1,2,7	3
7	1,2,3,6,8	5
8	2,3,4,7,9	5
9	3,4,5,8,10	5
10	4,5,9	3

A vertex is equivalent to a degree of freedom (hereinafter "dof") in finite element method, and adjacent vertices are equivalent to coupled degrees of freedom. If user has a background with finite element method, it is easy to verify the coupled dof with respect to each dof can be summarized as:

dof	*coupled dof*
1	2,6,7
2	1,3,6,7,8
3	2,4,7,8,9
4	3,5,8,9,10
5	4,9,10
6	1,2,7
7	1,2,3,6,8
8	2,3,4,7,9
9	3,4,5,8,10
10	4,5,9

Adjacent vertices and coupled dof identify connectivity. This manual uses the term "adjacent vertices" for illustration. Applying the above adjacent vertices to JCL, the new number of vertices is as:

```
10  8  6  4  2  9  7  5  3  1
```

E. g., replace "number 1" in Figure 1 with "number 10", and replace "number 2" with

"number 8", and so on. The new number is labeled as:

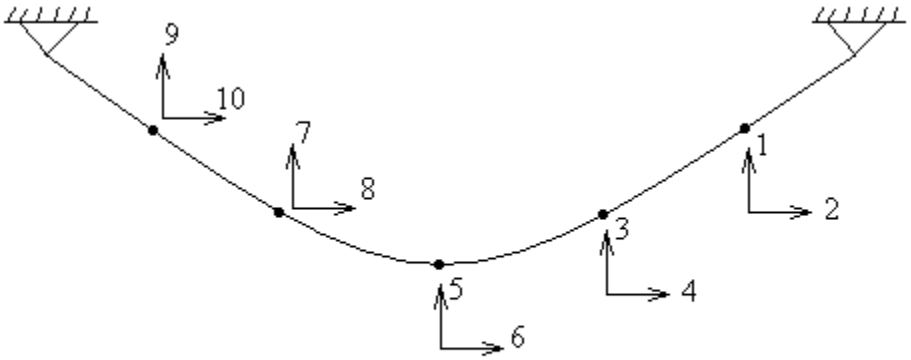


Figure 2

We apply the new number labeled in Figure 2 to solve displacements, e.g., by finite element method. The connectivity in Figure 2 shows the following:

1) At vertex 1, vertices 1, 2, 3, and 4 are connected (or coupled). That means, on the first row of [A], columns 1, 2, 3, and 4 have non-zero fill-ins.
2) At vertex 2, vertices 1, 2, 3, and 4 are connected. On the second row of [A], columns 1, 2, 3, and 4 have non-zero fill-ins.
3) At vertex 3, vertices 1, 2, 3, 4, 5, and 6 are connected. One the third row of [A], columns 1, 2, 3, 4, 5, and 6 have non-zero fill-ins.
4) .... and so on.

The new number of vertices leads the following matrix $[A]$:

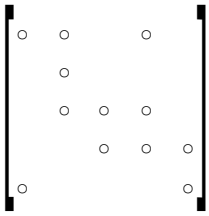
$$\begin{bmatrix}
A_{11} & A_{12} & A_{13} & A_{14} & & & & & & \\
A_{21} & A_{22} & A_{23} & A_{24} & & & & & & \\
A_{31} & A_{32} & A_{33} & A_{34} & A_{35} & A_{36} & & & & \\
A_{41} & A_{42} & A_{43} & A_{44} & A_{45} & A_{46} & & & & \\
 & & A_{53} & A_{54} & A_{55} & A_{56} & A_{57} & A_{58} & & \\
 & & A_{63} & A_{64} & A_{65} & A_{66} & A_{67} & A_{68} & & \\
 & & & & A_{75} & A_{76} & A_{77} & A_{78} & A_{79} & A_{7,10} \\
 & & & & A_{85} & A_{86} & A_{87} & A_{88} & A_{89} & A_{8,10} \\
 & & & & & & A_{97} & A_{98} & A_{99} & A_{9,10} \\
 & & & & & & A_{10,7} & A_{10,8} & A_{10,9} & A_{10,10}
\end{bmatrix}$$

User can use the new numbers to calculate matrix [A] without a rearrangement of rows and columns. The procedure to renumber vertices can be summarized as follows:

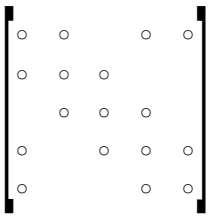
1) Arbitrarily number the vertices
2) Define adjacent vertices by the initial number
3) Apply JCL to get new number.

Chapter 4. Renumbering Asymmetric Sparse Matrix

When renumbering a matrix with asymmetric non-zero fill-ins, it is necessary to expand the asymmetric figure into a symmetric figure. For example, the following has an asymmetric non-zero fill-ins:



The asymmetric figure needs to be expanded into a symmetric figure by copying the transpose of upper triangular part into the lower triangular part, and copying the transpose of lower triangular part into the upper triangular part. The expanded symmetric figure becomes:



Then, JCL renumbers the expanded sparse matrix that has a symmetric non-zero fill-ins.

CHAPTER 5. Data Storage Schemes

Sparse matrix can be efficiently handled by a proper data storage schemes. Among computer languages, Fortran is most suitable for sparse matrix operations. This chapter will introduces data storage schemes for constant-bandwidth and variable-bandwidth sparse symmetric matrices. For sparse asymmetric matrices, please refer to LAIPE manual.

5.1 Constant Bandwidth Sparse Symmetric Matrix

The data storage scheme is applicable in subroutine (or function). Sparse matrix is passed as a dummy argument. Since it is symmetric, we handle the lower triangular part. For example,

$$\begin{bmatrix} 1 & & & & & & \\ 4 & 25 & & & & \textit{sym.} & \\ 2 & 29 & 88 & & & & \\ & 9 & 34 & 89 & & & \\ & & 3 & 23 & 45 & & \\ & & & 11 & 7 & 22 & \\ & & & & 3 & 2 & 9 \end{bmatrix}$$

Data storage scheme for constant bandwidth sparse symmetric matrix is declared in a Fortran program, for example,

```
Integer :: m
Real :: a(m,1)
```

where *m* is the half bandwidth and *a* is the sparse matrix. The total length of matrix *a* is *(n-1)*m+n*. Then, the coefficient A_{ij} is programmed in fortran as *a(i,j)*. Fortran provides a very convenient way to handle sparse matrix.

In the above example, *m=2* and *n=7*, the total required length for matrix *a* is *(7-1)*2+7=19*. A subroutine to input the above example is as follow.

```
Subroutine input(a,m)
integer :: m
real :: a(m,1)
a(1,1) =  1.0
a(2,1) =  4.0
a(3,1) =  2.0
a(2,2) = 25.0
a(3,2) = 29.0
a(4,2) =  9.0
a(3,3) = 88.0
a(4,3) = 34.0
a(5,3) =  3.0
a(4,4) = 89.0
a(5,4) = 23.0
a(6,4) = 11.0
a(5,5) = 45.0
a(6,5) =  7.0
a(7,5) =  3.0
a(6,6) = 22.0
a(7,6) =  2.0
a(7,7) =  9.0
return
end
```

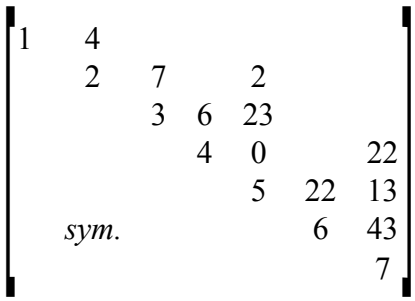
Matrix *a* requires a length of *19*. In the main program (or caller) needs to declare as:

```
real :: a(19)
```

and also need to define *m=2*.

5.2 Variable Bandwidth Sparse Symmetric Matrix

We handle the upper triangular part for variable bandwidth sparse symmetric matrix. For example,



Variable-bandwidth sparse symmetric solver is also called *skyline solver*. Date storage scheme includes an address label, *label*, whose definition is as:

```
Set label (1) = 1
For i = 2 to n, do the following
    label(i) = label(i-1)+[number of non-zero
                           fill-ins above the
                           diagonal in the
                           i-th column]
```

The address labels for the above example are *1, 2, 3, 4, 7, 8,* and *11*. The data storage scheme declares the sparse matrix in Fortran program, for example, as

```
real :: a(1,1)
```

Then, coefficient A_{ij} is programmed in Fortran program as *A(i,label(j))*. Fortran provides a very convenient way to handle sparse matrix. The total required length of matrix *a* is defined as <u>*lable(n)-1+n*</u> where *n* is matrix order. In the above example, *n=7* and *label(7)=11*, the required length is *11-1+7 = 17*. It can be verified there are 17 non-zero fill-ins. A fortran subroutine to input the above example is as:

```
subroutine input (a,label)
real :: a(1,1)
integer :: label(1)
a(1,label(1)) =  1.0
a(1,label(2)) =  4.0
a(2,label(2)) =  2.0
a(2,label(3)) =  7.0
a(3,label(3)) =  3.0
a(3,label(4)) =  6.0
a(4,label(4)) =  4.0
a(3,label(5)) =  2.0
```

```
a(4,label(5)) = 23.0
a(5,label(5)) =  0.0
a(4,label(5)) =  5.0
a(5,label(6)) = 22.0
a(6,label(6)) =  6.0
a(4,label(7)) = 22.0
a(5,label(7)) = 13.0
a(6,label(7)) = 43.0
a(7,label(7)) =  7.0
return
end
```