7. Matrix procedures for finite elements

7.1 Introduction: The extremely important assembly process (scattering) for combining element matrices to form the system matrices is independent of the physical application and the element type. The same process is used for fluid flow models and structural analysis, or any other fields of application. The topics covered in this chapter are automated and hidden within commercial finite element codes. The reader does not have to know this material if relying exclusively on such codes. However, if reader plans to solve small problems by hand then this information is critical.

The assembly process depends on how many unknowns are associated with each node in the mesh, and the element connection list which defines which nodes in the mesh are connected to each element. It is not necessary that every node have the same number of unknowns (degrees of freedom) but that common case is used here for simplicity. The assembly, the enforcement of boundary conditions and constraints, and the solution of the system matrices are easily automated and is hidden from the user of commercial software systems. However, to solve small problems by hand and to really understand the complete process the reader should master these concepts.

Finite element applications all lead to a system of linear algebraic equations to be solved for the set of unknowns. Often, a confusing aspect of this is how are the small element matrices assembled (scattered) into the much larger algebraic equations of the system? Unlike most examples included in a linear algebra course, most of the system equations resulting from a finite element study will usually be singular until the essential boundary conditions (EBC) have been imposed on the algebraic system.

The scattering of the element matrices is mainly a problem in matrix algebra. Earlier, it was noted in passing that the assembly could technically be done by using Boolean matrices built from the element connection list. However, that process is grossly inefficient from the
computational point of view. In this chapter, the critical process of assembling the system matrices is described in terms of a computationally efficient ‘direct assembly’ that is equivalent to the theoretical Boolean matrix process. The concepts, necessary Matlab scripts, and examples are presented on how to build the large algebraic system that will eventually yield the degrees of freedom at each node. Three methods for enforcing the essential boundary conditions on those system matrices are given.

Also, it is not uncommon for the system matrix equations to be subjected to one or more algebraic constraint equations that reduce the number of unknowns in the original set of equations. A “multipoint constraint” (MPC) imposes a linear relation between two or more of the unknowns. They occur often in real world finite element analysis and the process for including them is given and illustrated with simple examples. Both the EBC and any MPC must be imposed upon the system matrices before they can be solved for the unknown degrees of freedom, and any reactions of interest.

The system of equations is often very large and efficient ways for solving the linear equations must be used. A short discussion of factorization methods is also included here. Before any of these operations can be automated for all fields of application it is necessary to define how the degrees of freedom are to be numbered at each node.

### 7.2 Equation numbers for gather and scatter:

The finite element mesh defines a small number of quantities at the nodes of an element as a sub-set of the huge number of the same quantities at all the nodes in the assembled mesh of elements. There are different choices for numbering the items in such a list. The most common approach, followed here, is to sequentially number all the same kind of quantities at a node before going to the next node and repeating that process. Consider the common case where the same number of quantities, say \( n_g \) (for number of generalized items), occur at every node in the mesh. Then the local list of items at each node of any element can be defined as a sub-set of the same items at all the nodes in the mesh, say \( n_m \), by a simple equation derived by deduction.

At any node denote the number of the \( j \)-th item of interest as \( j = 1: n_g \), and let the \( i \)-th local node number of an element be \( i = 1: n_n \). The element node connection list, say \( e_{\text{nodes}}(i) \) (input by the user or a mesh generator) identifies the system node number in the full mesh that corresponds to that local node number. Then the local element item number, say \( n(i, j) \), is

\[
n(i, j) = n_g \times (i - 1) + j, \quad 1 \leq i \leq n_n, \quad 1 \leq j \leq n_g
\]

and defining corresponding system node number in the mesh as \( I \) the corresponding system item number is

\[
N(I, j) = n_g \times (I - 1) + j, \quad 1 \leq I \leq n_m, \quad 1 \leq j \leq n_g
\]

but since the corresponding system node number in the mesh is \( I = e_{\text{nodes}}(i) \) the corresponding system item number is

\[
N(i, j) = n_g \times (e_{\text{nodes}}(i) - 1) + j, \quad 1 \leq i \leq n_n, \quad 1 \leq j \leq n_g.
\]

Here, \( n_g \) is used to denote the number of unknowns at each node, also called the degrees of freedom (DOF) at each node. These local and system equation numberings are employed in
gathering from the system unknowns to local unknowns and for scattering (adding) local known arrays into locations within the corresponding system arrays.

Equation (7.2-3) is the major key to automating the essential process of assembling (scattering) the element arrays into the system arrays. It creates a list of all of the system equation numbers associated with a specific element. Here, that list is given the name \textit{rows} since it is initially used in examples to show how the rows of an element column matrix are scattered into the system column matrix. In other words, the list is used to convert the row subscript of an element array to the corresponding row in a system array. The same process is used to assemble square matrices which also each have a column subscript. The above list is also used to directly convert the column number of an element square matrix to the column number in the system square matrix where it is to be added. Since the list serves that dual purpose it is sometimes also called the element \textit{index} list.

Figure 7.2-1 shows how (7.2-3) is implemented as a Matlab script. The input arguments to the function \texttt{get_element_index.m} are the number of nodes connected to the element, $n_n \leftrightarrow n_n$, the number of generalized unknowns per node, $n_g \leftrightarrow n_g$, and the list of system node numbers connected to the element. The return from the function is the list of corresponding system equation numbers (which are used later as vector subscripts to achieve efficient programming). There is another aspect of the script that is related to the Matlab requirement that all lines in an input text file must have the same number of columns. The general finite element system provided herein allows the mesh to have a mixture (for example) of quadrilateral and triangular elements. In that case, the input line containing the connection list for the triangle must be appended (padded) with a zero so as to have the same number of columns as the quadrilaterals in the system connection list. Thus, the script has logic to skip zero node numbers due to a mesh containing different, but compatible, element types. The actual use of this script is shown later in Fig. 7.3-1.

```matlab
function [rows] = get_element_index (n_g, n_n, e_nodes) % ===
% calculate system DOF numbers of element, for gather, scatter
% e_nodes = maximum connectivity list for any element type
% n_g = number of DOF per node
% n_n = maximum number of nodes per element
% rows = vector subscript changing elem to system eq numbers
rows = zeros (1, n_g * n_n); % allow for node = 0
for k = 1:n_n; % loop over element nodes
  global_node = round (e_nodes (k)); % corresponding sys node
  for i = 1:n_g; % loop over DOF at node
    eq_global = i + n_g * (global_node - 1); % sys DOF, if any
    eq_element = i + n_g * (k - 1); % el DOF number
    if ( eq_global > 0 ); % check node=0 trick
      rows (1, eq_element) = eq_global; % valid DOF > 0
    end; % if allow for omitted nodes
  end; % for DOF i
end; % for each element node
% end get_element_index
```

Figure 7.2-1 The important calculation of system equation (DOF) numbers for an element
The two concepts of gathering and scattering element data from and to the system, respectively, are illustrated in Fig. 7.2-2. The major differences in the two processes are that the gather operation retrieves known items from the system and brings them to the element without changing the item. In comparison, the scatter operation takes known (recently calculated) element coefficients and adds them to specific locations in the system arrays.

The example mesh in Fig. 7.2-2 has the system node numbers that are not in sequential order. While that is unusual in a one-dimensional mesh it is common in two- and three-dimensional problems. Again, for simplicity that figure assumes just a single unknown per node. The script in Fig. 7.2-1 shows that scatter operations will work for any number of unknowns per node.

The mesh in Fig. 7.2-2 with a total of seven unknowns could be changed to three three-node line elements (L3, \( n_n = 3 \)) or two four-noded line elements (L4, \( n_n = 4 \)) and the number of system nodes (\( n_m = 7 \)) would be unchanged. Just the size of the typical element \( c_e \) matrices (and coordinate lists and connection lists) would increase from two rows to three or four rows, respectively. In other words, the element type (parametric space dimension) does not matter in the calculation of the element’s system equation numbers; just the number of nodes in the connection list matters.

In practice, all of the element array coefficients must be added (scattered) to rows and columns of the square matrix, \( S \), and source vector, \( c \), of the system equilibrium matrices:

\[
S \ u = c + c_{NBC}
\]

(7.2-4)

The system column matrix \( c \) is built up, from zero, by scattering (adding) the column matrices \( c_e \) from all of the elements in the mesh. Likewise, the system square matrix, \( S \), is built up, from zero, by scattering into it all of the element square matrices, \( S^e \).

The paired set of numbers can be used for other purposes also. For example, each node in the mesh and each local node on an element have \( n_s \) physical spatial coordinates. So, setting \( n_g = n_s \) yields the equations for gathering the coordinates of a single node (where I is constant) or the physical coordinates of all the local nodes on an element (for \( i = 1: n_n \)). Likewise, some applications have a list of properties assigned to every node. Then the above equations are easily converted to extract the nodal properties for an element.
Figure 7.2-2 Gathering and scattering data in a one-dimensional mesh ($n_g = 1$)

This numbering system is illustrated in a one-dimensional case in Fig. 7.2-2 where the system has six nodes ($n_m = 6$) and each element has two nodes ($n_n = 2$) each with a single generalized degree of freedom ($n_g = 1$) and where the nodes are numbered randomly (as they are by automatic mesh generators). The connection lists in Fig. 7.2-2 are used to gather the one-dimensional spatial coordinates ($n_s = 1$) for each element. But the line element in the figure could just as easily have been a planar curve where both coordinates are gathered. For the two-node line elements the gathered coordinates are used to calculate some $2 \times 1$ source column vector for each element, $c^e$. Each of the element column vectors are assembled (scattered) into the resultant $6 \times 1$ system column, $c$.

After the system square matrix has been assembled in a similar fashion (using real matrix values) and the system must be modified to include the essential boundary conditions the $6 \times 1$
system degrees of freedom, say \( \mathbf{u} \), are computed. Most studies do not end there because the degrees of freedom (DOF) almost always must be post-processed at the element level to calculate other items of importance. In such post-processing loops over every element the equation numbering system is used again to gather the now known \( 2 \times 1 \) element degrees of freedom, say \( \mathbf{u}_e \), from the appropriate system degrees of freedom for additional calculations in each element.

The scattering (assembly) process is the same for all finite element applications and needs to be clearly understood. Sometimes it is helpful to view the relation between the equation numbers for the element’s degrees of freedom (DOF) and the equation numbers for the system’s degrees of freedom in a tabular format. Table 7.2-1 gives such a table when the number of generalized DOF per node, \( n_g \), is constant. The process is only slightly changed if every node in the system is allowed to have a different number of DOFs.

**Table 7.2-1 Relating local and system equation numbers**

<table>
<thead>
<tr>
<th>Local node number</th>
<th>DOF number</th>
<th>System node number</th>
<th>Element degree of freedom (DOF) numbers:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( I_L )</td>
<td>( J )</td>
<td>( I_S = el_node(I_L) )</td>
<td>( n_g(I_L - 1) + J ) ( n_g(I_S - 1) + J )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>( el_node(1) )</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>( el_node(1) )</td>
<td>2</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>1</td>
<td>( n_g )</td>
<td>( el_node(1) )</td>
<td>( n_g )</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>( el_node(2) )</td>
<td>( n_g + 1 )</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>( el_node(2) )</td>
<td>( n_g + 2 )</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>( k )</td>
<td>( j )</td>
<td>( el_node(k) )</td>
<td>( n_g(k - 1) + j )</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>( n_n )</td>
<td>1</td>
<td>( el_node(n_n) )</td>
<td>( n_g(n_n - 1) + 1 )</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>( n_n )</td>
<td>( n_g )</td>
<td>( el_node(n_n) )</td>
<td>( n_n \cdot n_g )</td>
</tr>
</tbody>
</table>

\( n_g = \) number of degrees of freedom per node, \( n_n = \) number of nodes per element

\( el\_node(k) = \) node connection list for element number \( k \)

To provide a specific numerical example consider a large planar truss having two DOF per node: the horizontal and vertical displacement components (u and v). To be kinematically stable any truss mesh must be made up of triangular cells with three truss members pinned together at the vertices of the cell (sometime the earth serves as a third member of a cell). The must not be any quadrilateral cells enclosed by four truss members. Figure 7.2-3 shows a sample large truss with 400 nodes (800 DOF) along with segments of the system connectivity array \( (\text{nodes}) \) in which any row defines the connectivity of a single element (array \( \text{el\_nodes} \) ).
Each truss member has two nodes, and a planar truss has two displacements per node and thus contributes to four of the 800 system DOF. Consider truss element number 21 and how its element matrices would be scattered into the system matrices. The system equation numbers of the four truss DOFs are listed in Table 7.2-2. A planar truss member always has a $4 \times 4$ stiffness matrix, $S^e$, and usually a $4 \times 1$ gravity and/or wind load resultant, $c^e$. In order to add these element coefficients into the system equations one must identify the relation between the local degree of freedom numbers and the corresponding system degree of freedom numbers. The array index provides this information for any specific element. In practice, the assembly procedure is as follows. First the system matrices $S$ and $c$ are set equal to zero. Then a loop over all the elements is performed. For each element, the element matrices are generated in terms of the local degrees of freedom. Those coefficients of the element matrices are to be added to the corresponding coefficients in the system matrices. Before the addition is carried out, the element array index is used to directly convert the local subscripts of the coefficient to the system subscripts of the term in the system equations to which the coefficient is to be added.

### Table 7.2-2 Equation numbers for truss element 21, $n_i \equiv n_g \ast n_n = 4$

<table>
<thead>
<tr>
<th>Node Local I_L</th>
<th>Node System I_S</th>
<th>DOF J</th>
<th>Equation Local DOF</th>
<th>Equation System DOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>261</td>
<td>1</td>
<td>1</td>
<td>521</td>
</tr>
<tr>
<td>1</td>
<td>261</td>
<td>2</td>
<td>2</td>
<td>522</td>
</tr>
<tr>
<td>2</td>
<td>270</td>
<td>1</td>
<td>3</td>
<td>539</td>
</tr>
<tr>
<td>2</td>
<td>270</td>
<td>2</td>
<td>4</td>
<td>540</td>
</tr>
</tbody>
</table>

Consider how all of the 20 element coefficients ($4 \times 4$ and $4 \times 1$) for member 21 will be directly added into the system matrices. Here the symbol `+→` is used to indicate directly converting an element equation number to a system equation number (row number and/or column number) and then adding the element term onto the system term at that location:

$$c^e_1 + \rightarrow c_{521}, \ S^e_{1,1} + \rightarrow S_{521,521}, \ S^e_{1,2} + \rightarrow S_{521,522}, \text{ etc.}$$
\[ c_2^e \rightarrow c_{522}, \ S_2^e \rightarrow S_{522,522}, \ S_2^e, \ S_3^e \rightarrow S_{522,539}, \ \text{etc.} \]
\[ c_3^e \rightarrow c_{539}, \ S_3^e \rightarrow S_{539,539}, \ S_3^e, \ 539, \ S_4^e \rightarrow S_{539,540}, \ \text{etc.} \]
\[ c_4^e \rightarrow c_{540}, \ S_4^e \rightarrow S_{540,521}, \ S_4^e, \ 540, \ \text{etc.} \]

**Example 7.2-1 Given:** Using the line element data in Fig. 7.2-2 start with fictitious element column matrices where the two entries are simply the element number itself, \( c^e \equiv [e \ e] \) and then assemble the system column matrix, \( c \), starting with a null vector. **Solution:** Adding each of the two numbers to the rows of \( c \) defined by the two numbers in the element connection list gives:

\[
\begin{align*}
&\begin{array}{cccccccc}
+e & 1 & 2 & 3 & 4 & 5 & 6 & \text{All}
\end{array} \\
\text{Eq} & 1,3 & 3,5 & 4,7 & 5,6 & 4,2
\end{align*}
\]

\[
c = \begin{bmatrix}
1 \\
1 & 0 & 1 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
1 \\
0 & (1+2) & 3 \\
1
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 1 \\
0 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
1 \\
3 & 0 & 3 \\
0
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 1 \\
0 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 1 \\
3 & 0 & 3
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 1 \\
5 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 1 \\
5 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 1 \\
6 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 1 \\
6 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 1 \\
6 & 0 & 0
\end{bmatrix}
\]

Note in this assembly the first two system nodes \((1, 2)\) received only one number each. That is because those system nodes are only connected to one element. A general rule is that the system row in a column vector, like \( c \), will receive as many contributions to that row as there are elements connected to that system node. The entries in a real element column matrix are usually different and the first number goes to the row in \( c \) given by the first number in an element's connection list. Likewise, the second number goes to the row in \( c \) given by the second number in an element’s connection list.

**Example 7.2-2 Given:** Assume that the sequence of nodes in Fig. 7.2-2 correspond to three three-node triangular elements each sharing one corner node with its neighbor, or two four-node quadrilaterals sharing one corner. Develop the connection list for those two meshes. **Solution:** Here the triangle and quadrilateral corners are numbered counter-clockwise. The lists can be created beginning at any corner without affecting the analysis. Thus, one set of triangle connection lists becomes:

\[
\begin{array}{cccc}
& (1) & (2) & (3) \\
3 & 1 & 5 & \\
5 & 2 & 7 & (3) \\
7 & 3 & 2 & (3) \\
\end{array}
\]

and a set of quadrilateral connections becomes

\[
\begin{array}{cccc}
& (1) & (2) & (3) \\
1 & 1 & 5 & \\
2 & 7 & 6 & 5 \\
3 & 2 & 4 & 6 \\
\end{array}
\]

\[
\begin{array}{cccc}
& (1) & (2) & (3) \\
1 & 1 & 5 & 7 \\
2 & 6 & 4 & 7 \\
\end{array}
\]
7.3 Vector subscripts: A programming notation for arrays that is related to the use of vector subscripts is the colon, :, notation. In both Matlab and Fortran using the colon as an array subscript means it refers to the full range of that subscript. For example, referring to Fig. 7.2-3 and citing \( \mathbf{u}(:) \) means \( \mathbf{u}(1) \) through \( \mathbf{u}(6) \). An array subscript, and a loop, can also be ranged in an incremental fashion from a beginning integer, B, to and ending integer, E, with an integer increment of I. In Matlab that ranging is denoted by B:I:E; and in Fortran as B:E:I. For example, \( \mathbf{u}(1:2:6) \) above refers to the odd number of locations: \( \mathbf{u}(1) \) through \( \mathbf{u}(6) \).

Vector subscripts are efficient programming tools in FEA. They are standard in the Fortran language and in the Matlab environment. A vector subscript is a named integer vector (one-dimensional array) of values which do not have to be in sequential order and can contain duplicate values. When the named array is placed in the row and/or column locators of any other array the software treats them as an implied “for loop” or “do loop” that uses each integer in the vector subscript, in sequential order, as the range values for that loop over the rows and/or columns of the second array.

For example, consider a \( 4 \times 5 \) array, \( \mathbf{A} \), from which four specific non-sequential coefficients, included in angle brackets, are to be copied:

\[
\mathbf{A} = \begin{bmatrix}
1 & 2 & 3 & 4 & 5 \\
6 & 7 & 8 & (9) & (10) \\
11 & 12 & 13 & (14) & (15) \\
16 & 17 & 18 & 19 & 20 \\
\end{bmatrix}
\]

The row and column locations of those terms in matrix \( \mathbf{A} \) can be identified with vector subscripts as \( \text{Row} = [2 \ 3], \text{Col} = [4 \ 5] \). In Fortran and Matlab a new array containing those four terms is

\[
\mathbf{B} = \mathbf{A(\text{Row},\text{Col})} = \begin{bmatrix} 9 & 10 \\ 14 & 15 \end{bmatrix}
\]

Now assume those four terms are to be added (scattered) into the original matrix \( \mathbf{A} \) in rows 3 and 4 and columns 1 and 3. A new set of vector subscripts are formed: \( \text{Row} = [3 \ 4], \text{Col} = [1 \ 3] \), and used carry out the addition with a pair of implied loops:

\[
\mathbf{A(\text{Row},\text{Col})} = \mathbf{A(\text{Row},\text{Col})} + \mathbf{B( :, :) } = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\
6 & 7 & 8 & (9) & (10) \\
(20) & 12 & (23) & 14 & 15 \\
(30) & 17 & (33) & 19 & 20 \end{bmatrix}
\]

As shown in the last section, the data needed for scattering the element arrays are in the connection lists for each element. Those numbers are the system mesh node numbers. Usually, the system node numbers are randomly assigned by an automatic mesh generator (and possibly re-ordered for more efficient storage and solution operations. That list of nodes defines were each local element row is to be added into a row in the system equations. For the special case of
a single unknown per node \((n_g = 1)\) such a direct equation subscript change defines the scatter destination row as

\[
\text{Scatter}_\text{row} (k) = \text{Element}_\text{connection} (k), k = 1, \ldots, n_n. 
\]  

(7.3-1)

For example, to find where the single \(k\)-th row term in an element source, \(c^e\), is to be added into the total system source vector, \(c\), the procedure is as follows. From the connection list for the system mesh extract the row of node connections for the current element. That row contains the list of integer node numbers that are attached to the element. From that list extract the \(k\)-th system node number, say \(K\). The increment the system source sum by adding the \(k\)-th of \(c^e\) to the \(K\)-th row of \(c\):

\[
c(K) = c(K) + c^e(k) 
\]  

(7.3-2)

That would add the effect of a single degree of freedom of that element into the system matrix. But, all of the element coefficients must be added into the system equations. As (7.3-1) shows, that is accomplished by looping over all of the node numbers for the current element. That would be done with a ‘for-loop’ in Matlab or a ‘do-loop’ in Fortran. However, the programming a loop is not efficient; especially for parallel computers. Therefore, Fortran and Matlab both offer a more efficient programming option known as a “vector subscript”. The appearance of a vector subscript is an implied loop calculation to be executed in the most efficient way available in the computer being utilized.

Therefore, rather than write a loop over all the nodes in the element connection row that row of integers is extracted from the input data and employed as a vector subscript. Here, that list is simply given name the array “row”. For example, scattering (adding) the element column matrix, \(c^e\), to the system column matrix, \(c\), is written in code as:

\[
c(\text{row}) = c(\text{row}) + c^e 
\]  

(7.3-3)

But this is actually executed as the implied loop:

\begin{verbatim}
% Scatter the element column matrix into the system column matrix
% Array ‘row’ contains the system DOF numbers connected to this element
n_DOF = size (row) % determine number of element equations
for k = 1:n_DOF % loop over all element DOF rows
c (row (k)) = c (row (k)) + c_e (k) % add element term to system term
end % for k element column matrix terms
\end{verbatim} 

(7.3-4)

In a similar fashion, the row subscripts and column subscripts are converted using the element connection lists. The column numbers are reordered exactly like the row numbers \((\text{col} \equiv \text{row})\). In vector subscript notation the assembly (scattering) of the element square matrix into the system square matrix is

\[
S(\text{row}, \text{row}) = S(\text{row}, \text{row}) + S^e 
\]  

(7.3-5)

Again, this is executed as two implied loops:

\begin{verbatim}
% Scatter the element square matrix into the system square matrix
% \(n\_\text{DOF}\) contains the system DOF numbers connected to this element
n_DOF = size (row) % determine number of element equations
for k = 1:n_DOF % loop over all element DOF rows
    c (row (k)) = c (row (k)) + c_e (k) % add element term to system term
end % for k element column matrix terms
\end{verbatim} 

(7.3-6)
The above assembly process is very simple to program in the Fortran language and in the Matlab environment by using vector subscripts. As illustrated in Fig. 7.3-1 the process is repeated for every element in the mesh.

### Example 7.3-1

**Given:** A single quadratic (three-noded) bar element has a load vector of

\[
c^e = \begin{bmatrix} \frac{w^e L^e}{6} \\ \frac{4}{1} \end{bmatrix}
\]

where \( L^e \) is the bar length and \( w^e \) is the axial load per unit length on the bar. Due to a non-sequential system node numbering the connection list for that element is \([3 \ 1 \ 2] = \text{row} \). Determine the system load vector resulting from scattering the coefficients to the system level.

**Solution:** For this single element, the scatter (assembly) of the element column matrix gives

\[
c^e(:, \text{row}) \rightarrow c(\text{row})
\]

Looping through each of the three numbers in the vector subscripts gives:

\[
c^e(1) + c(3) \rightarrow c(3) = \frac{w^e L^e}{6} + 0 = \frac{w^e L^e}{6}
\]

\[
c^e(2) + c(1) \rightarrow c(1) = \frac{4 w^e L^e}{6} + 0 = \frac{4 w^e L^e}{6}
\]
\[ c^e(3) + c(2) \rightarrow c(2) = \frac{w^e L^e}{6} + 0 = \frac{w^e L^e}{6} \]

So, after the assembly the system source vector is

\[ c = \frac{w^e L^e}{6} \begin{pmatrix} 4 \\ 1 \\ 1 \end{pmatrix} \]

### 7.4 Partitioning the system equations:

In most courses on matrix algebra the linear systems to be solved usually involve non-singular square matrices. That is because any constraints on the unknowns were satisfied in advance. In the finite element applications to be covered here the constraints are satisfied last, which results in an initially singular square matrix because the reactions due to the constraints appear as unknowns on the column vector side of the equations. In other words, because the essential boundary conditions will be enforced last and there is not a unique solution until that is done. A matrix partition can be used to allow for the solution constraints and to yield a smaller and non-singular system of equations to be solved. The algebraic system can be written in a general matrix form that more clearly defines what must be done to reduce the system to a solvable form by utilizing known essential boundary condition values. The system degrees of freedom, \( u \), and the full equations could always be re-arranged in the following partitioned matrix form

\[
\begin{bmatrix}
S_{uu} & S_{uk} \\
S_{ku} & S_{kk}
\end{bmatrix}
\begin{bmatrix}
u_u \\
u_k
\end{bmatrix} =
\begin{bmatrix}
c_u \\
c_k + r_k
\end{bmatrix}
\] (7.4-1)

where \( u_u \) represents the unknown nodal parameters, and \( u_k \) represents the known essential boundary values of the other parameters. The \( r_k \) term represents that there are usually unknown generalized reactions associated with essential boundary conditions. The only unknowns in this matrix system are the vectors \( u_u \) and \( u_k \). The net number of unknowns corresponds to the number of equations, but they must be re-arranged before all the remaining unknowns can be computed. Moving the unknown reactions, \( r_k \), to the left side would destroy the very useful symmetry of the square matrix. So instead they are obtained as a post-solution calculation.

The sub-matrices \( S_{uu} \) and \( S_{kk} \) are square, whereas \( S_{uk} \) and \( S_{ku} \) are rectangular, in general. In a finite element formulation all of the coefficients in the \( S \) and \( c \) matrices are known. This means that in general after the essential boundary conditions \( (u_k) \) are prescribed the remaining unknowns are \( u_u \) and \( r_k \). The unknown degrees of freedom are always calculated, but the calculation of the reaction unknowns is optional.

Here, for simplicity, it has been assumed that the equations have been numbered in a manner that places the prescribed parameters (essential boundary conditions) at the end of the system equations. The above matrix equations can be re-written in expanded form as:

\[
S_{uu} u_u + S_{uk} u_k = c_u
\]
\[
S_{ku} u_u + S_{kk} u_k = c_k + r_k
\] (7.4-2)
so that the unknown nodal parameters are obtained by inverting the non-singular square matrix $S_{uu}$ in the top partitioned rows. That is,

$$u = S_{uu}^{-1} (c - S_{uk}u_k). \quad (7.4-3)$$

Most books on numerical analysis assume that you have reduced the system to the above non-singular form where the essential conditions, $u_k$, have already been moved to the right hand side. Many authors use examples with null conditions $u_k = 0$ so the solution is the simplest form, $u = S_{uu}^{-1}c$. If desired, the values of the necessary reactions, $r_k$, can now be determined from

$$r_k = S_{ku}u + S_{kk}u_k - c_k \quad (7.4-4)$$

In structural applications using line elements like bars, beam, trusses, and frames the reactions on each member is sometimes needed in the design of the member. They are the forces and/or moments that each member transmits at its interface nodes. For the individual element to be in equilibrium with its now known stiffness, $S^e$, known displacement vector, $u^e$, and known element source resultant vector, $c^e$, and the unknown external reaction point forces and/or moments, $r^e$, requires

$$S^e u^e = c^e + r^e \rightarrow r^e = S^e u^e - c^e. \quad (7.4-5)$$

That element level relation is probably easiest to envision for an element in the mesh that has no distributed external sources, $c^e \equiv 0$, and then the member reactions are simply $r^e = S^e u^e$ since, like with a linear spring, external forces are required to deform an elastic body.

In nonlinear and time dependent applications the reactions can be found from similar calculations. In most applications the reaction data have physical meanings that are important in their own right, or useful in validating the solution. However, this part of the calculations is optional. If one formulates a finite element model that satisfies the essential boundary conditions in advance, then the second row of the partitioned system $S$ matrix is usually not generated and one cannot recover the reaction data directly from the matrix system.

The above matrix partitioning can be accomplished easily with languages that allow vector subscripts. In several examples given later the Matlab scripts will employ vector subscripts. Fig. 7.4-1 illustrates how vector subscripts can extract the above partitions in a coupled matrix system and use them to solve for the unknowns and the reactions from the EBC. In that example, the nodal degrees of freedom with essential boundary conditions occur at the first (1) and last (5) nodes, so those two numbers are defined as an integer vector subscript called $Fixed$. All other degrees of freedom are unknown and are placed in the vector subscript $Free$. The Matlab script is listed on the left side of that figure and the results of that script are on the right side.
7.5 Numerically equivalent process: The actual partitioning of the full matrix is not actually necessary and neither is the use of vector subscripts. Alternate ways to assign a known value to a single degree of freedom, say \( d_j \), can be done with numerical manipulations. Since the number of essential boundary conditions are often a very small percentage of the number of unknowns the enforcement of the EBC can be completed by column and row operations that modify both the square matrix and the column vector for each assigned EBC for vector \( d \), say \( d_j = b \).

The operation first subtracts the assigned value, \( b \) times the \( j \)-th column of \( S \), and then zeros out that column in \( S \) along with the \( j \)-th row of both \( S \) and \( c \). At this point the system is singular (non-unique) because the EBC is still not included in the square matrix. To enforce the EBC, include the EBC as a scalar identity, \( d_j = b \), in \( S \) and \( c \). That is, place a 1 in the matrix diagonal location \( S_{jj} \) and in the column vector set \( d_j = b \) so the size of the equations remains the same but the square matrix is rendered non-singular.

For the complete details see each step illustrated in Table 7.5-1. This approach is stored in the function `enforce_essential_BC.m` and/or the related penalty method in the Matlab function `enforce_MPC_equations.m`. If the reaction data, \( r_j \), need to be recovered in post-processing, then the original \( j \)-th row of \( S \) and \( c \) must be saved for later use.
The above series of steps are repeated for each and every EBC in the problem. The prior matrix partition approach is correct for any size problem but is more commonly used on very small problems, while the identity manipulation (or the penalty method, below) is used when there are thousands or millions of equations in the problem.
Table 7.5-1 Steps to enforce EBCs, without renumbering equations

Original matrix system before EBC: \( \mathbf{d}_j = \mathbf{b} \)

\[
\begin{bmatrix}
1 & \ldots & j & \ldots & k & \ldots & p \\
1 & S_{11} & S_{1j} & S_{1k} & S_{1p} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
j & S_{j1} & S_{jj} & S_{jk} & S_{jp} & d_j \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
k & S_{k1} & S_{kj} & S_{kk} & S_{kp} & d_k \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
p & S_{p1} & S_{pj} & S_{pk} & S_{pp} & d_p \\
\end{bmatrix}
\begin{bmatrix}
d_1 \\
d_j \\
d_k \\
d_p \\
\end{bmatrix}
= 
\begin{bmatrix}
c_1 \\
c_j + r_j \\
c_k \\
c_p \\
\end{bmatrix}
\]

**Step 1**, move \( \mathbf{b} \) column \( j \) to right side

\[
\begin{bmatrix}
1 & \ldots & j & \ldots & k & \ldots & p \\
1 & S_{11} & 0 & S_{1k} & S_{1p} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
j & 0 & \ldots & 0 & S_{jp} & d_j \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
k & S_{k1} & 0 & S_{kk} & S_{kp} & d_k \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
p & S_{p1} & 0 & S_{pk} & S_{pp} & d_p \\
\end{bmatrix}
\begin{bmatrix}
d_1 \\
d_j \\
d_k \\
d_p \\
\end{bmatrix}
= 
\begin{bmatrix}
c_1 \cdot S_{1j} \\
c_j \\
c_k - b \cdot S_{kj} \\
c_p - b \cdot S_{pj} \\
\end{bmatrix}
\]

**Step 2**, zero row \( j \) (equations now singular)

\[
\begin{bmatrix}
1 & \ldots & j & \ldots & k & \ldots & p \\
1 & S_{11} & 0 & S_{1k} & S_{1p} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
j & 0 & \ldots & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
k & S_{k1} & 0 & S_{kk} & S_{kp} & d_k \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
p & S_{p1} & 0 & S_{pk} & S_{pp} & d_p \\
\end{bmatrix}
\begin{bmatrix}
d_1 \\
d_j \\
d_k \\
d_p \\
\end{bmatrix}
= 
\begin{bmatrix}
c_1 \cdot S_{1j} \\
0 \\
c_k - b \cdot S_{kj} \\
c_p - b \cdot S_{pj} \\
\end{bmatrix}
\]

**Step 3** insert identity \( \text{Big} \times \mathbf{D}_j = \text{Big} \times \mathbf{b} \), \( \text{Big} = \max(S) \)

\[
\begin{bmatrix}
1 & \ldots & j & \ldots & k & \ldots & p \\
1 & S_{11} & 0 & S_{1k} & S_{1p} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
j & 0 & \ldots & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
k & S_{k1} & 0 & S_{kk} & S_{kp} & d_k \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
p & S_{p1} & 0 & S_{pk} & S_{pp} & d_p \\
\end{bmatrix}
\begin{bmatrix}
d_1 \\
d_j \\
d_k \\
d_p \\
\end{bmatrix}
= 
\begin{bmatrix}
c_1 - b \cdot S_{1j} \\
\text{Big} \cdot b \\
c_k - b \cdot S_{kj} \\
c_p - b \cdot S_{pj} \\
\end{bmatrix}
\]

(Equations now non-singular)

Solving gives \( \mathbf{d}_j = \mathbf{b} \) and remaining unknowns

**7.6 EBC by a penalty method**: The essential boundary conditions can be satisfied, to a selected number of significant figures, by a penalty method that also leaves the original size and
equation order unchanged (but which does change the column heights of a sparse storage approach). Let \( d_j \) be the DOF to be assigned an essential boundary condition, say \( d_j = b \).

A numerical trick to avoid partitions or zeroing out rows and columns is known as the penalty method. For an EBC it changes only two terms in the governing matrix system. The \( j \)-th row of the resultant source vector is changed to

\[
 c_j = \mathbb{R} b S_{j,j} 
\]

and its diagonal term is changed to

\[
 S_{j,j} = \mathbb{R} S_{j,j} 
\]

where \( \mathbb{R} \) is a really really big number. Then that row product of the system matrices becomes

\[
 S_{j,1} d_1 + \cdots + \mathbb{R} S_{j,j} d_j + S_{j,k} d_k + \cdots + S_{j,n} d_n = \mathbb{R} b S_{j,j} 
\]  

(7.6-1)

As the penalty term approaches infinity, \( \mathbb{R} \to \infty \), the numerical solution approaches \( d_j = b \) to a certain number of significant figures. The actual value of \( \mathbb{R} \) to be assigned depends on the hardware word length (which engineering programming languages can display). Values of \( \mathbb{R} \) as high as \( \mathbb{R} = 10^{23} \) have been used. (The author uses a scaling of the maximum diagonal element of the square matrix, say \( \mathbb{R} = 10^5 \times \max[\mathbf{S}] \)).

### 7.7 Multiple point constraints* (MPC)

It is not uncommon for structures to be supported on rollers that move against a surface that is not parallel to any of the global axes. Then, what would normally be an essential boundary condition becomes a constraint involving the displacement vector and the normal vector at that surface. The above penalty approach is also the easiest way to treat cases where two or more degrees of freedom are related by a linear constraint such as \( A d_j + B d_k = C \) and \( A d_j + B d_k + C d_m = D \) coupling three DOFs, which occur frequently. Let the first DOF be the master and the following ones are dependent DOFs. Such a constraint equation between multiple rows (points) in the solution vector has long been called a “Multiple Point Constraint” or MPC. Clearly, the first coefficient is not zero, so the equation is divided through by it to store as many coefficients as there are DOF in the constraint.

As an example, the constraint between two DOF is:

\[
 A d_j + B d_k = C \\
 A/A d_j + B/A d_k = C/A \\
 d_j + b d_k = c 
\]  

(7.7-1)

So, its input equations become \( d_j + b d_k = c \), where the new coefficients are \( b = B/A, c = C/A \). Likewise, for a linear constraint between three DOF, \( A d_j + B d_k + C d_m = D \) has an input form of \( d_j + b d_k + c d_m = d \) where \( b = B/A, c = C/A, d = D/A \). To input such a constraint equation you would enter the integer degree of freedom numbers followed by the real values of the coefficients (one more than the number of DOF in the constraint).

Note that there is more than one DOF in each equation. That is, there are more unknowns than equations. An approximate solution can be obtained from a least squares penalty model (or a singular-decomposition algorithm). Either of the example constraints can be written as a null matrix dot product.
\[
[1 \ b] \{d_j\}_{d_k} - c = 0, \quad \text{or} \quad [1 \ b \ c] \begin{bmatrix} d_j \\ d_k \\ d_m \end{bmatrix} - d = 0
\]

For the first form, if the equation is not satisfied the error is:
\[
[1 \ b] \{d_j\}_{d_k} - c \neq 0 = Err.
\]

The error squared is
\[
Err^T Err = \left( \begin{bmatrix} d_j \\ d_k \end{bmatrix}^T [1 \ b] - c^T \right) \left( [1 \ b] \begin{bmatrix} d_j \\ d_k \end{bmatrix} - c \right)
\]
\[
Err^2 = [d_j \ d_k] \begin{bmatrix} 1 \\ b \end{bmatrix} [1 \ b] \begin{bmatrix} d_j \\ d_k \end{bmatrix} - [d_j \ d_k] \begin{bmatrix} 1 \\ b \end{bmatrix} c - c^T [1 \ b] \begin{bmatrix} d_j \\ d_k \end{bmatrix} + c^T c
\]
\[
Err^2 = \frac{1}{2} [d_j \ d_k] \left( \begin{bmatrix} 1 \\ b \end{bmatrix} [1 \ b] \begin{bmatrix} c \\ b c \end{bmatrix} - \begin{bmatrix} c \\ b c \end{bmatrix} \right) + \frac{1}{2} c^T c
\]

To find the constrained DOFs, \(d^c\), that minimize the error solve for
\[
\frac{\partial Err^2}{\partial \{d^c\}} = \{0\}, \quad \frac{2}{2} \left( [1 \ b] \begin{bmatrix} d_j \\ d_k \end{bmatrix} - \begin{bmatrix} c \\ b c \end{bmatrix} \right) = \{0\}
\]

when multiplied a really big number, \(\mathbb{R}\), these define a pseudo-element stiffness and load relation:
\[
S^c d^c = c^c. \quad (7.7-2)
\]

Such constraint matrices defined for the first form are
\[
S^c = \mathbb{R} \begin{bmatrix} 1 & b \\ b & b^2 \end{bmatrix}, \quad c^c = \mathbb{R} \begin{bmatrix} c \\ b c \end{bmatrix}
\]

These matrix identities are treated as pseudo-element equations that are assembled into the system equations like actual elements (including when the nodes or elements are renumbered to improve the sparseness of the system square matrix).

Sometimes you need two DOF to be identical, then \(d_j + (-1) d_k = 0\). Often that is called a rigid link and that constraint system is
\[
S^c = \mathbb{R} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}, \quad c^c = \begin{bmatrix} 0 \\ 0 \end{bmatrix}
\]

That square matrix is similar to that of a bar or linear spring: \(S^e = \frac{EA}{L} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}\).

The term \(EA/L\) is known as the axial stiffness of a bar. Thus, the constraint equation is thought of as a pseudo-element which acts as a very stiff spring connecting the two equal degrees of freedom. This is sometimes called a ‘rigid link’.
The elastic modulus of a material, \( E \), is usually a very large number. Without the introduction of \( \mathbb{R} \), that almost always would lead to \( S^p \gg S^c \) and the numerical constraint would not be accurately satisfied in the numerical solution of the assembled system matrix. The large penalty number scaling is needed such that the constrain equation pseudo-element has the proper units and is much larger that a physical element, \( S^c \gg S^p \). That can be done by multiplying the constraint pseudo-element matrices by the really really big number \( \mathbb{R} \), and by arbitrarily defining \( \mathbb{R} = 10^5 \times \max[S] \) it is assured that the constraints have the proper units. Likewise, the constraint equation for the three degree of freedom example is

\[
S^c = \mathbb{R} \begin{bmatrix} 1 & b & c \\
 b & b^2 & bc \\
 c & cb & c^2 \end{bmatrix}, \quad c^c = \mathbb{R} \begin{bmatrix} d \\
 bd \\
 cd \end{bmatrix}
\]

All MPC pseudo-element arrays have a connectivity list, and are scattered into the system arrays. That scatter is done last so that the maximum physical diagonal term, \( \max[S] \), is known.

In the supplied library of finite element scripts the MPC connectivity list (DOF numbers) and the list of user defined coefficients are read by script \texttt{get_constraint_eqs.m}. Those data are supplied by the user in the text file \texttt{msh_mpc.txt}. After all constraint equation data are read the system counts the number of each type of constraint so that memory can be allocated for the pseudo-elements of each type. Herein the constraint type is the number of coupled degrees of freedom in the constraint equation. The script \texttt{count_MPC_eqs.m} counts the number of constraint equations of each type. The penalty method of enforcing the constraints (after complete assembly of all physical elements) on the system \( S \) and \( c \) arrays is carried out by the script \texttt{enforce_MPC_equations.m}. The most common MPC occurs when a node in a structure is required to move tangent to an inclined support surface, as illustrated in Ex. 7.7-1. For a surface in three-dimensions the components of its unit vector are simply the direction cosines of the line perpendicular to the surface.

In the supplied library of finite element scripts the nodal boundary condition flag also plays a part when any MPC is present in the data. Recall that the flag (present in file \texttt{msh_bc_xyz.txt}) is a packed integer consisting of \( n_g \) digits packed into a single integer. The first (leftmost) digit refers to any EBC or MPC that is active for the first degree of freedom at that node, and so on. If a digit value is 0 that means that there is no EBC or MPC active for that DOF at that node. If a digit value is 1 that means that there is an EBC (a type 1 constraint equation) to be enforced for that DOF at that node. If a digit value is a number greater than one \( (1 < m \leq 9) \) that means that there is a MPC of type \( m \) to be enforced for that DOF at that node. Those integer flags are internally split back into their \( n_g \) individual integers for the purpose of allocating memory and for checking for consistency between the user input files for a single application. Those operations are completed in script \texttt{count_EBC_MPC_flags.m}.

There are exact matrix manipulation methods for enforcing MPC relations. The script library can handle full matrix systems having the common type 2 MPC with the matrix manipulator \texttt{apply_mpc_type_2.m}. It gives more accurate results than the penalty approach, but for sparse storage systems it tends to reduce the sparseness more than the penalty approach. That is not a problem when the system square matrix, \( S \), is fully populated as in most of the examples herein.
Figure 7.7-1 A truss supported by an inclined roller

% mshRemarks.txt: remarks about the application
A three bar planar truss with inclined support
Type 1 is a line truss element, n_n = 2
n_g = 2, n_e = 3, n_m = 3, n_d = 6
% msh_bc_xyz.txt: n_g digit BC flag, x-coord, y-, z-coord
11 0.0 0.0
00 0.0 400.0
22 300.0 400.0
% msh_typ_nodes.txt: element-type, connection list
1 1, 2
1 2, 3
1 1, 3
% msh_load_pt.txt: node, direction, source_value
2 1 1e5
% msh_ebc.txt: node-number, dof-number, assigned value
1 1 0.0
1 2 0.0
% msh_mpc.txt: one line per constraint equation
% group of node-dof pairs, group of coefficients
3 1 3 2 -0.75 0.0
% msh_properties.txt: one line per element OR element type
% for Type 1: Area, E_modulus, Spec_wt, Alpha, Del_T
0.001 209e9 77e3 17e-6 0.

Figure 7.7-2 Numerical text files to define the truss mesh and restraints

Figure 7.7-3 Torsional rotations of coupled shafts
Example 7.7-1 Given: the third node of a planar truss is supported by an inclined roller that makes an angle of $\alpha$ with respect to the positive x-axis and has a slope of 3 horizontal to 4 vertical. Determine the constraint equation between the components of the displacement vector at the support point. Solution: A roller is defined as a support that allows motion tangent to its surface only. In other words, the displacement of a structure normal to the support surface is zero. The displacement vector is $\delta = u \mathbf{i} + v \mathbf{j}$ and the surface normal vector is $\mathbf{n} = -\sin \alpha \mathbf{i} + \cos \alpha \mathbf{j}$. The constrain equation for zero normal displacement is $\delta \cdot \mathbf{n} \equiv 0$. From the definition of the dot product of two vectors: $-u \sin \alpha + v \cos \alpha = 0$. For the given slope $\cos \alpha = 3/5$ and $\sin \alpha = 4/5$. The system equation number for the $x$-displacement at node 3 is $2(3 - 1) + 1 = 5$ while the equation number for the $y$-displacement is 6. Therefore, the constraint equation needed to enforce this support condition is $\delta_5(-4/5) + \delta_6(3/5) = 0$ which would be simplified for input as $\delta_5 - (3/4)\delta_6 = 0$.

Example 7.7-2 Given: For the supplied FEA Matlab library prepare the user data files for the truss in Fig. 7.7-1 (except for the properties). Solution: The only new operation for the truss is to note that it has a type 2 MPC. Thus, the digit 2 must appear in the boundary condition flag for each of the degrees of freedom involved in the constraint (in text file msh_bc_xyz.txt), and the new MPC data file must give the node number and degree of freedom number for each DOF in the constraint equation, and the corresponding constraint coefficients (like $b$, $c$, $d$ above). Those data go in text file msh_mpc.txt. The required user data are listed in Fig. 7.7-2. This truss is solved later in Ex. 8.5-4.

Example 7.7-3 Given: The torsional shafts in Fig. 7.7-3 are coupled by two gears. The finite element unknowns are rotations at nodes 1 through 4. The shaft and gear rotation at node 2 is $\theta_2$ and its gear has a radius of 5 inches. That gear engages a smaller gear with a 1 inch radius. The smaller gear and shaft node 3 have a rotation of $\theta_3$. Determine the constraint equation between the shaft element rotations at nodes 2 and 3. Solution: A point on the circumference of the large gear moves through an arc length of $L = \theta_2 R_2$. A point on the circumference of the smaller gear moves through the same arc length $L = \theta_3 R_3$ but with the rotation being in the opposite direction. Therefore, the constraint equation between the two rotations at the nodes coupled by those gears is $5\theta_2 + 1\theta_3 = 0$. (Shaft elements are derived in section 8.5.)

7.8 Wilson’s static condensation algorithm: The static condensation algorithm of Professor Ed Wilson is well known. The static matrix equilibrium equation for an element having internal degrees of freedom, $K\delta = F$, can be written in a partitioned form to separate the internal and external degrees of freedom:

$$\begin{bmatrix} K_{aa} & K_{ab} \\ K_{ba} & K_{bb} \end{bmatrix} \begin{bmatrix} \delta_a \\ \delta_b \end{bmatrix} = \begin{bmatrix} F_a \\ F_b \end{bmatrix}$$

(7.8-1)

where $\delta_a$ indicates the internal degrees of freedom to be eliminated and $\delta_b$ indicates the external degrees of freedom that are associated with the reduced stiffness matrix $(K^*)$. The upper partition gives

$$\delta_a = K_{aa}^{-1}(F_a - K_{ab}\delta_b).$$
so the lower partition becomes
\[ K_{bb} \delta_b = F_b - K_{ba} \delta_a = F_b - K_{ba} K_{aa}^{-1} (F_a - K_{ab} \delta_b) \]
or
\[ \left[ K_{bb} - K_{ba} K_{aa}^{-1} K_{ab} \right] \delta_b = \left\{ F_b - K_{ba} K_{aa}^{-1} F_a \right\} \]
this defines the reduced element equilibrium equation as
\[ K^* \delta_b = F^*. \quad (7.8-2) \]

This process defines a square work matrix, \( W \), a rectangular transformation matrix, \( T_k \), and a column transformation matrix, \( T_F \), respectively:
\[ W \equiv K_{aa}^{-1}, \quad T_k \equiv W K_{ab}, \quad T_F \equiv W F_a, \quad (7.8-3) \]
to denote the reduced system matrices as
\[ K^* = K_{bb} - K_{ba} T_k, \quad F^* = F_b - K_{ba} T_F. \quad (7.8-4) \]
Also, the first partition is expressed in terms of the transformations as
\[ \delta_a = T_F - T_k \delta_b. \quad (7.8-5) \]
which allows for additional accurate post-processing of related internal quantities, like stresses.

These equations are well suited for both symbolic and numerical calculation of the reduced element matrices. A numerical static condensation algorithm is given in Fig. 7.8-1. It returns (or preferably stores) the two transformation matrices as well as the reduced element matrices. After the solution of the assembled system equations the element external degrees of freedom (e.g. displacements), \( \delta_b \), are gathered and the previously eliminated internal degrees of freedom, \( \delta_a \), are recovered from (7.8-5) using the two element transformation matrices. An algorithm to numerically recover the internal degrees of freedom is shown in Fig. 7.8-2.
function [Kr, Pr, Tk, Tp] = remove_internal_dof (K, P, out)
% static condensation algorithm, removes internal dof
% from K*U=P, to give Kr*Ur=Pr at the element level
% see Wilson IJNME 8,1,198-203,1974 & recover_internal_dof
% K = static stiffness, or conduction, matrix
% keep = vector subscripts of dof to be kept
% P = resultant force, or source, vector
% out = vector subscripts of dof to be condensed out
% Kr = condensed version of K, without internal dof
% Pr = condensed version of P, without internal dof
% Tk = rectangular transformation matrix for K
% Tp = column transformation matrix for P
% W = work matrix to form Tk, and Tp

n_k = size (K, 1) ; % number of original dof
n_i = size (out) ; % number of dof to remove
[keep] = get_free_from_fixed (n_k, out) ; % dof to keep
Tk = zeros (n_i, n_k); Tp = zeros (n_i, 1) ; % allocate
W = zeros (n_i, n_i) ; % allocate
W = inv (K(out, out)) ; % invert diagonal partition
Tk = W * K (out, keep) ; % transform for K
Tp = W * P (out) ; % transform for P and U
Kr = K (keep, keep) - K (keep, out) * Tk ; % reduced K
Pr = P (keep) - K (keep, out) * Tp ; % reduced P
% NOTE: For post-processing, get U(out) = Tp - Tk * U(keep)
% end remove_internal_dof

Figure 7.8-1 Condensation algorithm to form reduce element matrices

function [U] recover_internal_dof (Tk, Tp, out, Ur) %======
% recover internal dof to form full U for element
% post-processing. See Wilson IJNME 8,1,198-203,1974
% keep = vector subscripts of dof to be kept
% Tk = column transformation matrix for K, and
% Tp = column transformation matrix for P, were returned
% from [Kr, Pr, Tk, Tp] = remove_internal_dof (K, P, out)
% Ur = gathered external element dof after system solution
% U = full set of element degrees of freedom

n_in = max(size(out)); % number of internal dof to recover
n_ex = max(size(Ur)); % number of external dof, in Ur
n_Tk = max(size(Tk)); % number of external dof, from Tk
n_i = n_in + n_ex ; % number of element dof, in U
if ( n_ex ~= n_Tk ) ; % if inconsistent arguments
  error ('Invalid argument sizes in recover_internal_dof')
end

U = zeros (n_i, 1) ; % allocate all element dof
[kept] = get_free_from_fixed (n_i, out) ; % dof kept in Ur
U(kept) = Ur(:) ; % insert external dof into full dof
U(out) = Tp(:) - Tk*Ur(:) ; % insert internal dof into full
% end recover_internal_dof

Figure 7.8-2 Recovering internal degrees of freedom for post-processing
7.9 Equation Factorization: Most of the examples given herein imply that the solution of the equilibrium equations (after enforcing the boundary conditions), \( S \mathbf{u} = \mathbf{c} \), will be obtained by inverting the system square matrix and multiplying that inverted matrix times the solution source vector, \( \mathbf{c} \), to obtain the system solution vector, \( \mathbf{u} = S^{-1} \mathbf{c} \). While that is theoretically a correct matrix algebra procedure it is not practical for more than three unknowns. A more efficient solution process is to factor the system square matrix into the product of a lower triangular matrix, \( L \), and an upper triangular matrix, \( U \), as \( S \equiv LU \), or as a triple matrix product using a diagonal matrix, \( S \equiv LDL^T \), where usually \( U = L^T \). A factorization solution algorithm greatly reduces the operations count and the required storage.

One important issue is that the finite element square matrix is sparse and diagonally dominated but its inverse matrix is fully populated. That contrasts with a factorization process where the matrix factors of \( S \) have the same sparsity and can be stored within the original matrix and thus require no additional storage (when the factorizations are programmed efficiently, unlike the Matlab \( \text{lu} \) function).

Example 7.9-1: Given: Consider the sparse matrix \( S \) below. Use the Matlab function \( \text{inv}(S) \) to form the inverse and the function \( \text{sul}(S) \) the two triangular factors and compare the storage requirements. Verify that \( S \equiv LU \).

\[
S = \begin{bmatrix}
2 & 1 & 0 & 0 & 0 \\
1 & 2 & 1 & 0 & 0 \\
0 & 1 & 2 & 1 & 0 \\
0 & 0 & 1 & 2 & 1 \\
0 & 0 & 0 & 1 & 2
\end{bmatrix}
\]

Solution: its inverse matrix is full populated

\[
S^{-1} = \frac{1}{6} \begin{bmatrix}
5 & -4 & 3 & -2 & 1 \\
-4 & 8 & -6 & 4 & -2 \\
3 & -6 & 9 & -6 & 3 \\
-2 & 4 & -6 & 8 & -4 \\
1 & -2 & 3 & -4 & 5
\end{bmatrix}
\]

but the two factors are sparse, and in some cases can be stored in (overwrite) the original matrix:

\[
L = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
1/2 & 1 & 0 & 0 & 0 \\
0 & 2/3 & 1 & 0 & 0 \\
0 & 0 & 3/4 & 1 & 0 \\
0 & 0 & 0 & 4/5 & 1
\end{bmatrix}, \quad U = \begin{bmatrix}
2 & 1 & 0 & 0 & 0 \\
0 & 3/2 & 1 & 0 & 0 \\
0 & 0 & 4/3 & 1 & 0 \\
0 & 0 & 0 & 5/4 & 1 \\
0 & 0 & 0 & 0 & 6/5
\end{bmatrix}
\]

Evaluating \( S - LU \) yields a null matrix as expected.

In practical FEA the size of the equations is so large that a matrix inversion is not practical and a factorization of the equations or an iterative solution is used. Factorization is especially efficient when the square matrix is symmetric and positive definite. Then the Crout Factorization, \( S \equiv LDL^T \), is very popular for large finite element systems. The Crout factorization is used herein with the skyline sparse matrix storage method discussed on the next section. The process for factoring a square matrix into an upper and lower triangle product is sketched in Fig. 7.9-1. After the factorization, a forward substitution loop and a backward
substitution loop are run. They each involve only one equation and one unknown when solving for the intermediate dummy unknown and the actual unknown in each row of the matrix system. When the matrix size gets extremely large an iterative solver, like a pre-conditioned conjugant gradient method, is utilized.

Figure 7.9-1 Utilizing matrix factorization instead of matrix inversion
7.10 Skyline sparse storage: Up to this point, and in the majority of the supplied scripts, it has been assumed that the system square matrix will be assembled, stored and factored as a full matrix. There are other more efficient sparse storage and solution techniques. A sparse storage technique for the system square matrix that has proved very efficient in finite element solutions is the “skyline storage mode”. In that mode, the mainly non-zero terms in the full matrix are stored as one long vector. To use these procedures it is necessary to determine the top of the non-zero coefficients in every column of the system square matrix. The number of non-zero coefficients above, and including the diagonal coefficient, is called the height of the column. By using the element connection list, the degrees of freedom associated with the element are easily determined.

As shown in Fig. 7.10-1, those system equation numbers define the system column heights for that element. Retaining the highest height in each column defines how many coefficients will have to be stored in the system square matrix once the assembly of all elements is completed. If the system matrix is symmetric then only the coefficients in the column heights need to be stored, as suggested by Fig. 7.10-2. If the application leads to a non-symmetric system matrix then a second (lower triangle) skyline matrix of the same shape and size is used to store the other half of the coefficients (yes, that duplicates the storage of the diagonal terms).

![Skyline storage for a five degree of freedom mesh](image)

*Figure 7.10-1 Skyline storage for a five degree of freedom mesh*
There are sparse storage modes that avoid storing and operating on any zero coefficients. They do that by storing the row and column number, and value, of every non-zero coefficient in the system square matrix. For finite element applications the author has found that the skyline mode yields a much faster run time than the more general sparse storage and solver techniques. Figure 7.10-2 illustrates that the stored coefficients in each column are numbered from the top down from the first column to the last column. Note that the skyline storage mode can store some zero coefficients in the columns, like S (2, 4) in Fig. 7.10-1, but their number is vastly smaller that the number of zeros stored, and operated upon, in a full matrix storage mode. There are re-sequencing algorithms (like the Cuthill-McGee algorithm) that can greatly reduce the column heights by re-numbering the nodes in the mesh. That results in reduced memory requirements, and drastically reduced solution times. Such algorithms are not utilized here.

**Example 7.10-1 Given:** In Ex. 12.14-1 the assembled sparse system conduction matrix is

\[
S = \begin{bmatrix}
4 & -4 & 0 & 0 & 0 & 0 \\
-4 & 16 & -8 & -4 & 0 & 0 \\
0 & -8 & 16 & 0 & -8 & 0 \\
0 & -4 & 0 & 8 & -4 & 0 \\
0 & 0 & -8 & -4 & 16 & -4 \\
0 & 0 & 0 & 0 & -4 & 4
\end{bmatrix}
\]

which requires \(6 \times 6 = 36\) storage locations. Develop the corresponding skyline storage matrix.  

**Solution:** Since the system matrix is symmetric it could have been stored and assembled as an upper triangular matrix as

\[
S \leftrightarrow U = \begin{bmatrix}
4 & -4 & 0 & 0 & 0 & 0 \\
0 & 16 & -8 & -4 & 0 & 0 \\
0 & 0 & 16 & 0 & -8 & 0 \\
0 & 0 & 0 & 8 & -4 & 0 \\
0 & 0 & 0 & 0 & 16 & -4 \\
0 & 0 & 0 & 0 & 0 & 4
\end{bmatrix}
\]

which requires only 21 storage locations, which is an 18% reduction in storage. But it still contains nine zeros with eight of those zeros in the uppermost columns portions. Those eight zeros would be filled with non-zeros during a factorization on inversion process. Thus those (and any) zeros wastefully increase the number of operations needed to solve the system. The skyline mode stores only the columns from the first non-zero entry to the diagonal. In this case it is a \(13 \times 1\) vector (which must be allocated before use):

\[
S_{sky} = [ 4; -4; 16; -8; 16; -4; 0; 8; -8; -4; 16; -4; 4 ]^T
\]

which corresponds to about a 64% reduction in the storage requirement. It also contains only one zero term and will reduce wasted operations on zeros by about 89%.  

---

27
Figure 7.10-2 System square matrix stored as a vector

Since a typical coefficient in the system square matrix can no longer be located by inspection if it is stored in skyline, or any other sparse, mode all of the operations for gathering, scattering, applying essential boundary conditions, printing, or multiplying the coefficients by a vector must be programmed in a different way. All of those operations can be hidden from the user. An additional source library directory, called Akin_Sky_Lib.m, has been supplied for users that wish to extend the basic application library to solve practical problems with meshes introducing thousands of unknowns. Switching to the skyline mode can be quite simple by including the Matlab “addpath” command to automatically search that directory. In addition, a key control word can be used to allow the user to run in either the full matrix mode or in the skyline mode and is discussed in Appendix B.

There are two important integer arrays that are required in skyline mode to: 1. Determine the amount of storage required, 2. Locate where a typical coefficient, $S(j, k)$, is stored in the $S_{sky}$ vector. That information is required for any operation that is normally applied to the full mode $S$ array, and is extracted from the element connection list. Those two vectors are named $i_{dof_hi}$, and $i_{diag}$, respectively, which are shown in Fig. 10.7-3.
function [i_dof_hi] = sky_hi (n_d, n_g, nodes)  %==
% find column heights of system equations in skyline
% storage mode (zero height is inactive eq)
% i_dof_hi(i) = col height of sys dof i
% n_d  = total no of system dof
% n_g  = number of parameters per node
% nodes = nodal incidences of all elements
n_e = size(nodes, 1) ; % number of elements
n_n = size(nodes, 2) ; % number of nodes per element
n_i = n_n * n_g  ; % max number of element dof

i_dof_hi = zeros (n_d, 1) ; % zero column heights
l_high = zeros (n_i, 1) ; % zero work vector
lt_nodes = zeros (n_n, 1) ; % zero work vector
lt_index = zeros (n_i, 1) ; % zero work vector

% loop over elements
for ie = 1:n_e ; % loop over elements ====> ====> ====> ====>
    lt_n = sum (nodes (ie, :) > 0) ; % number of type nodes
    lt_nodes (1:lt_n) = nodes (ie, 1:lt_n) ; % get type nodes
    lt_free = lt_n*n_g ; % get element types number of DOF
    lt_index (1:lt_free) = get_elem_index (lt_n, n_g, ...
        lt_nodes) ; % get elem DOF numbers
    [l_high] = el_high (lt_free, lt_index) ; % this col height
    for j = 1:lt_free ; % compare with current maxmb hi
        ndx = lt_index (j) ; % get system column number
        if ( ndx > 0 ) ; % a valid column number ?
            if ( i_dof_hi (ndx) < l_high (j)) ; % exceeds max ?
                i_dof_hi (ndx) = l_high (j) ; % keep the new max hi
            end ; % if
        end ; % for element type dof
end ; % for all elements <=<<< <=<<< <=<<< <=<<< <=<<<
% end function sky_hi

function [i_diag] = get_sky_diag (n_d, i_dof_hi)
%============================================================================
% use column heights to find diagonal locations for
% skyline storage mode. columns stored from top down
%============================================================================
% n_d  = total no of system equations
% i_dof_hi(i) = column height of eq i, with diag
% i_diag(i)  = locate diag of i-th eq in upper tri
% Total number of sq matrix terms = i_diag(n_d)

i_diag = zeros (n_d, 1) ; % allocate diagonal items

ipoint = 0 ; % initialize the diagonal location
for k = 1:n_d ; % loop over all degrees of freedom
    ipoint = ipoint + i_dof_hi (k) ; % add col hi
    i_diag (k) = ipoint ; % update diagonal locations
end ; % for k over all dof
% end get_sky_diag

Fig. 10.7-3 Extracting the system skyline from the element connection list
Example values of those arrays for a small matrix are given in Fig. 7.10-2. The relation between the full storage row and column locations \((j, k)\) and the corresponding row, \(j_k.v\), in \(S_{\text{sky}}\) is:

\[
j_k.v = i_{\text{diag}}(\max([j,k])) - \abs(j - k)
\]

(7.10-1)

if, and only if, the \(S(j, k)\) coefficient falls within the skyline. Otherwise, using additional logic, the location is set to \(j_k.v \equiv 0\) to flag that it is a zero coefficient outside the skyline and is not required for any operation on the square matrix. These operations are frequently needed and are executed in function \textit{get\_sky\_subscript.m} given in Fig. 10.7-4.

```matlab
function [i_j_v] = get_sky_subscript (n_d, i_diag, i, j)
% convert (i,j) full symmetric matrix subscripts to
% (i_j_v) subscript of vector skyline storage mode.
% i_j_v = 0 if outside skyline and thus has 0 value.

id = max ([i,j]); % biggest of row or column
i_j_v = i_diag (id) - abs(i-j); % sky vector location

% Test for zero entry outside skyline
if ( i_j_v <= i_diag (id - 1) ); % then is not needed
    i_j_v = 0 ; % flag full term not in vector storage
end
% if is location re-set needed
end
% if full term is off the diagonal
end get_sky_subscript
```

Figure 10.7-4 Locating a full matrix term in the skyline vector

\textbf{Example 7.10-2 Given:} Referring to Fig. 7.10-2, find where the full array coefficients \(S(3, 6)\) and \(S(3, 7)\) are stored in the skyline vector. \textbf{Solution:} For row 3 and column 7 Eq. 7.10-1 gives the corresponding possible single subscript location in the skyline vector is

\[
j_k.v = i_{\text{diag}}(\max(3,6)) - \abs(3 - 6)
\]

\[
= i_{\text{diag}}(6) - 3 = 15 - 3 = 12
\]

But, is that a valid location above the diagonal? Checking; is

\[
12 > i_{\text{diag}}(\max(3,6) - 1) ?
\]

\[
12 > i_{\text{diag}}(6 - 1) ?
\]

\[
12 > i_{\text{diag}}(5) = 11 ?
\]
But, the coefficient $S(3, 7)$ has a potential location calculation of

$$j_k v = i_{\text{diag}}(\max(3, 7)) - \text{abs}(3 - 7)$$

$$= i_{\text{diag}}(7) - 4 = 17 - 4 = 13$$

But is

$$13 > i_{\text{diag}}(\max(3, 7) - 1) ?$$

$$13 > i_{\text{diag}}(7 - 1) ?$$

$$13 > i_{\text{diag}}(6) = 15 ?$$

No! So that term will not be stored in the skyline vector. It falls outside the skyline and is not stored so it not needed for any future operation to be applied to the full $S$ matrix. Therefore, set $j_k v = 0$ to flag that coefficient $S(3, 7)$ is a zero term that is not stored in the skyline vector.

The change to any sparse storage mode requires a complete library of other functions that do the many operations that are automatic in a full storage mode. For example, multiplying a square matrix by a vector is a common operation, but its sparse implementation requires locating the non-zero entries in each row and multiplying only them by the corresponding terms in the vector. The sequential changes that need to be made to the main program in order to use the sparse skyline storage mode to assemble and solve the system equations are listed in Appendix A. That appendix also shows the main new called functions stored at the Akin_Sky_Lib path. The main changes are to set a flag to use skyline storage, use the connection list to determine the number of terms in the sky vector, call alternate functions to: assemble the element square matrices; copy the data for calculating the reactions; to apply the essential boundary conditions; to solve the system equations; and recover the system reactions. Thereafter no changes are required.

### 7.10 Summary

- $n_b \equiv$ Number of boundary segments
- $n_e \equiv$ Number of elements
- $n_i \equiv$ Number of unknowns per element
- $n_n \equiv$ Number of nodes per element
- $n_q \equiv$ Number of total quadrature points
- $b =$ boundary segment number
- $\subset =$ subset of a larger set

Boundary, element, and system unknowns:

Boolean extraction arrays:

- $\delta^b \subset_b \delta^e \subset_e \delta$
- $\delta^b \equiv \beta^b \delta$, $\delta^e \equiv \beta^e \delta$
- $\Omega^e \equiv$ Element domain
- $\Gamma = \cup_b \Gamma^b \equiv$ Domain boundary

Geometry: $\Gamma^b \subset \Omega^e \equiv$ Boundary segment

Solution domain $\Omega = \cup_e \Omega^e$

Local equation (degree of freedom) number at element node $i$ for parameter $j$

$$n(i, j) = n_g * (i - 1) + j, \quad 1 \leq i \leq n_n, \quad 1 \leq j \leq n_g$$

System equation (degree of freedom) number at system node $I$ for parameter $j$

$$N(I, j) = n_g * (I - 1) + j, \quad 1 \leq I \leq n_m, \quad 1 \leq j \leq n_g$$
System mesh node number, $I$, at local element node $i$: $I = e_{\text{nodes}}(i)$

Matrix equilibrium equations (partitioned); $u_u = \text{remaining unknowns}$, $u_k = \text{set by EBCs}$, $r_k = \text{reactions needed to maintain EBCs}$, $S = \text{known stiffnesses}$, $c = \text{known sources}$:

$$
\begin{bmatrix}
S_{uu} & S_{uk} \\
S_{ku} & S_{kk}
\end{bmatrix}
\begin{bmatrix}
u_u \\
u_k
\end{bmatrix}
= \begin{bmatrix}
c_u \\
c_k + r_k
\end{bmatrix}
$$

$$
u_u = S_{uu}^{-1}(c_u - S_{uk} u_k), \quad r_k = S_{ku} u_u + u - c_k$$

Multipoint constraint (MPC), Type 2: $A u_j + B u_k = C \rightarrow u_j + b u_k = c$

Element static condensation:

$$
\begin{bmatrix}
K_{aa} & K_{ab} \\
K_{ba} & K_{bb}
\end{bmatrix}
\begin{bmatrix}
\delta_a \\
\delta_b
\end{bmatrix}
= \begin{bmatrix}
F_a \\
F_b
\end{bmatrix}, \quad W \equiv K_{aa}^{-1}, \quad T_k \equiv WK_{ab}, \quad T_F \equiv WF_a,
$$

$$
K^* = K_{bb} - K_{ba} T_k, \quad F^* = F_b - K_{ba} T_F, \quad K^* \delta_b = F^*, \quad \delta_a = T_F - T_k \delta_b
$$

Numerical factorization of a symmetric, positive definite square matrix (replaces $u = S^{-1}c$)

$L = \text{lower triangle matrix}$, $U = \text{upper triangle matrix}$:

$$
S u = c, \quad S \Rightarrow L \times U, \quad L \times (U \times u) = c, \quad U \times u = g, \quad L \times g = c
$$

Forward substitution: $\rightarrow g$, $U \times u = g$, Backward substitution: $\rightarrow u$

### 7.11 Exercises

#### 7.12 List of Examples

- Ex 7.2-1 Assembly (scatter) of six column source vectors, L2
- Ex 7.2-2 Form connection lists for seven nodes connected in two ways, T3, Q4
- Ex 7.3-1 Assembly (scatter) with a non-sequential connection list, L3
- Ex 7.7-1 Write the constraint equation for an inclined 2D roller support
- Ex 7.7-2 Prepare sample data for Matlab truss analysis
- Ex 7.7-3 Write the constraint equation for two geared torsional shafts
- Ex 7.10-1 Factor a 5x5 symmetric matrix into upper and lower triangles
- Ex 7.10-2 Store a symmetric 6x6 matrix in sparse skyline storage format

### 7.13 Index

- addpath, 24
- apply_mpc_type_2.m, 17
- assembly, 1, 6, 10
- axial stiffness, 16
- backward substitution, 20, 26
- Boolean matrix, 1
- column heights, 23
- connection list, 1, 3
- connectivity list, 17
- constraint, 10
- constraint equation, 19
- count_EBC_MPC_flags.m, 17
- count_MPC_eqs.m, 17
- Cuthill-McGee algorithm, 24
degree of freedom numbers, 26
degrees of freedom, 1, 2, 11
direct assembly, 8
DOF, 2, 5
DOF numbering, 2
do-loop, 8
EBC, 11
element static condensation, 26
enforce EBC, 11, 12
enforce_MPC_equations.m, 17
enforcing EBC, 2
essential boundary condition, 12
essential boundary conditions, 1
exercises, 28
factorization, 20, 22
for, 3
for-loop, 8
Fortran, 7
forward substitution, 20, 26
gather, 4, 5
gear, 19
get_constraint_eqs.m, 17
get_element_index.m, 3
implied loop, 9
inclined roller, 18
L3_C0, 10
linear algebraic equations, 1
Matlab colon, 7
matrix equilibrium equations, 26
matrix factorization, 26
matrix partition, 11
MPC, 2, 15, 17
msh_bc_xyz.txt, 17
msh_ebc.txt, 18
msh_load_pt.txt, 18
msh_mpc.txt, 18
msh_properties.txt, 18
msh_typ_nodes.txt, 18
multipoint constraint, 2, 15, 26
numerical manipulations, 12
padded connection list, 3
padded list, 3
penalty method, 13, 14
penalty number, 17
post-processing, 12
pseudo-element, 16, 17
reaction, 2, 11, 12
reactions, 12
rigid link, 16
scatter, 1, 4, 5, 6, 9
shaft, 19
skyline storage mode, 23
sparse storage, 22
sub-set, 2
summary, 26
Summary, 26
system equations, 1
system equilibrium matrices, 4
torsion, 19
triangle matrix, 26
truss, 18
vector subscript, 3, 7
vector subscripts, 13