-

1

Introduction

§1.1 SUMMARY

This report presents the first complete implementation of the Finite Element Method (FEM) using the *Mathematica* language. The report focuses on *data structures*, *data flow* and *programming modules* for linear structural mechanics.

The material collected herein directly supports most of the syllabus of the course **Finite Element Programming with Mathematica** (ASEN 5519). The programming modules are also used to support computer homework in the course **Introduction to Finite Element Methods** (ASEN 5007); however, the logic of those modules is not studied.

The present implementation has been designed with the following extensions in mind:

- Parallel computation
- Dynamic analysis, both transient and modal
- Nonlinear analysis
- Multiphysics problems

Extensibility is achieved by presenting data structures that are likely to change in *list* form. Lists are highly flexible because they can accomodate objects of any type in an arbitrary number of hierarchical levels. Some list structures can (and should) be readily implemented as arrays to increase processing efficiency in computational intensive tasks, whereas others may be implemented as derived data types in languages such as C, C++ or Fortran 90.

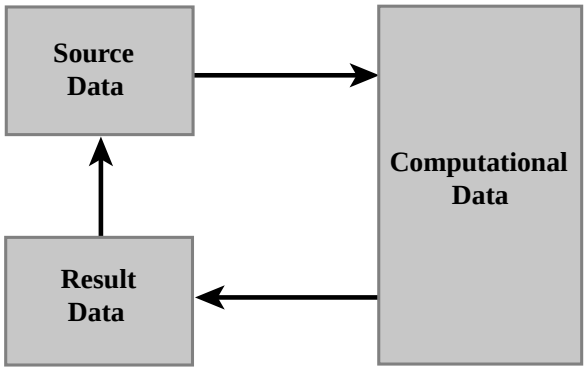


Figure 1.1. High level data flow in a Finite Element program.

§1.2 ORGANIZATION OF A FEM PROGRAM

§1.2.1 Data Flow

The high level data flow in any FEM program is schematized in Figure 1.1. This flows naturally suggests the grouping of data structures into three classes:

Source Data Structures. These bear a close relation to the FE model as the user defined it. For example, a table of node coordinates.

Computational Data Structures. As the name suggests, these are organized with processing efficiency in mind. The use of arrays is important for this goal. Example are sparsely stored coefficient matrices, and partitioned solution vectors.

Result Data Structures. These contain computed results again organized in a format that bears close relation to the FE model. Examples are completed node displacement vectors and element stress tables.

The feedback of *results into source* depicted in Figure 1.1, is one of the fundamental guides of the present study. The underlying philosophy is to view results as data that, on return from the computational phase, *completes the unknown portions* of source structures. This idea has unifying power because:

- It simplifies the merging of pre- and postprocessors so that the user deals with only one program.
- It is a natural way to organize saves and restarts in long remote computations.
- It allows a painless extension of linear into nonlinear analysis through a seamless cycle.

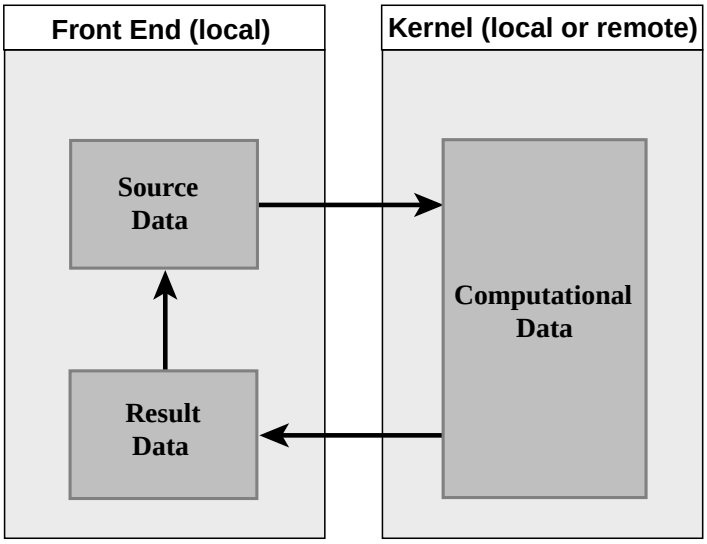


Figure 1.2. Separation of FEM system into Front End and Kernel.

§1.2.2 Front End and Kernel Programs

To simplify program operation it is convenient to separate the FEM system into the two parts diagrammed in Figure 1.2:

Front End. This program, or suite of programs, defines the model by direct input or generation of source data, prepares inputs and control information for the computational kernel, and handles visualization of input data as well as analysis results. It runs on a *local machine* such as a workstation or personal computer with adequate graphic facilities.

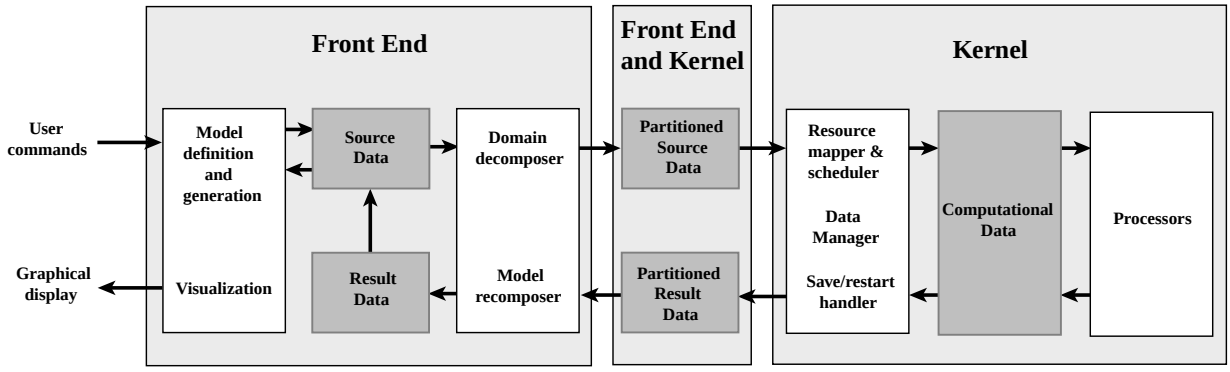


Figure 1.3. Further breakdown of data flow in FEM analysis system.

Computational Kernel. Also simply called *kernel*. This program, or suite of programs, handles the processing steps that require heavy number crunching. The kernel can run either on the local computer or on a remote one. Here "remote" is used in the sense of logically separated from the local, and does not necessarily imply physical distance. Physically remote processing is presently the rule, however, in the case of large-scale runs on massively parallel platforms.

Even if all runs could be done on the same machine, (for example a local, medium-level parallel computer such as a SGI Onyx), the front-end/kernel separation is strongly recommended for modularity reasons. If running on high-end parallel supercomputers is one of the main objectives the separation is mandatory since such systems are typically batch-oriented and not designed for time-shared local support.

A final consideration that supports separation is the ability to use different programming languages. The front end is best implemented with an object-oriented language such as C++. On the other hand that language may be overkill (or be unavailable on some massively parallel platforms) in the case of the kernel, for which C, or Fortran 90, or C mixed with Fortran 77, may suffice.

§1.2.3 Further Data Flow Breakdown

Figure 1.3 breaks down the data flow into additional steps and identifies the program components that perform specific functions. Some of these components deserve comment.

Commercial FEM systems usually distinguish between pre-processors, which define the problem and carry out mesh generation functions, from post-processors, which report and display results. This distinction has historical roots. The separation has been eliminated in the present organization by feeding back all results into source data structures. As a consequence there is no artificial distinction between inputs and outputs at the leftmost end of Figure 1.1.

The program component labeled "domain decomposer" is a fixture of task-parallel parallel processing. Its function is to break down the model into subdomains by element grouping. Usually each subdomain is assigned to a processor. Its output is called *partitioned source data*, which is supplied

as input to the kernel. The results delivered by the kernel are usually partitioned by subdomain, and must be reorganized through a reconstitution process before being merged back into the source data.

§1.3 REPORT ORGANIZATION AND TERMINOLOGY

§1.3.1 Coverage

The report is organized as follows. Chapter 1 is an overview of design principles, program organization, data classification, and nomenclature. Chapters 2 through 7 deal with source data structures associated with nodes, elements, degrees of freedom and loads. Chapters 8, 9 and 10 deal with auxiliary data structures constructed in preparation for the computational phases. Chapters 11, 12 and 13 deal with solution data structures. Chapter 14 offers conclusions and recommendations. Sections containing advanced material that may be omitted on first reading are identified by an asterisk.

The design of result data structures is omitted because of time constraints. It will be incorporated during the offering of the course.

§1.3.2 Objects, Lists, Table, DataSets

A *data structure* is an object or collection of objects that store related information, and is identified by a unique name.

Data structure identifiers are case sensitive: `force` is not the same as `Force`.

An *object* is any primitive or composite entity representable in computer memory and optionally identified by a name. Objects may be primitive items, such as the integer 3, a complicated composite item such as a complete PostScript graphic file, a function definition, or a composition of simpler objects.

A *list* structure is a named sequence of objects. It is identified enclosing the objects, separated by commas, with curly braces. For example:

```
A = { a,b,c,d }
```
(1.1)

Here list `A` is a defined as the sequence of four objects named `a`, `b`, `c` and `d`.

Lists may be embedded within lists through any depth. For example:

```
B = { a,b, { 1,2,3,cuatro }, d }
```
(1.2)

`B` is a two-level list if `a`, `b`, `cuatro` and `d` are not lists. In practice lists of more than three levels are rarely needed for the present study.

A list contained within a list is called a *sublist*.

A *table* is a list that is entered by a primitive key attribute, such as a node number or an element name, and returns a sublist associated with that key.

A *dataset* is a collection or grouping of data that pertains to a general or specific activity. For example, "node definition dataset" means all the data that pertains to the definition of nodal points. A dataset generally is a collection of related lists, and does not necessarily needs a name identifier.

§1.3.3 Arrays

A one-dimensional *array*, or simply *array*, is a list of objects of the same type, each of which uses exactly the same storage space. For example:

$$\texttt{Squares = \{ 1,4,9,16,25 \}} \tag{1.3}$$

is an integer array. This may be efficiently implemented as a primitive data type in most programming languages. A two-dimensional array is a list of one-dimensional arrays of identical length and type, and so on.

Arrays are often related to matrix objects. To emphasize the relationship matrix/vector notation may be used. In that case brackets are delimiters and comma separators are omitted. Thus (1.3) can be also displayed as

$$\texttt{Squares = [1 4 9 16 25]} \tag{1.4}$$

When array data structures are intimately connected to formulas, the name may be a matrix or vector symbol, which is always written in boldface. For example:

$$\mathbf{g}^T = [\, g_1 \quad g_2 \quad g_3 \,] \tag{1.5}$$

defines a 3-component column vector **g**.

§1.3.4 Naming Conventions

Three kind of names are associated with each of the major data structures presented here:

Complete names. For example, Master Node Definition Table. Such names are mnemonic but often too long for concise descriptions as well as programming.

Short names. For a list structure this is an acronym normally formed with the initials of the complete name. For example, MNDT for Master Node Definition Table. Letters are in upper or lower case following the conventions of Table 1.1. For array structures that correspond directly to vector or matrix objects, the matrix or vector symbol, such as $\mathbf{K}_b$ or **u**, may serve as short name.

Program names. These are used in the computer implementation. They are always shown in `typewriter` font. In this document, program names are usually taken to be the same as short names, but this is a matter of convenience. Programmers are of course free to make their own choices; see Remark below.

REMARK 1.1

A program-naming convention that greatly facilitates code maintenance is to use at least one `CAPITAL LETTER` for major data structures, while reserving all `lower case` names for less important entities such as indices,

temporary variables and the like. The style may of course vary with the implementation language. For example, here are some choices for the Master Node Definition Table:

```
MNDT      M_node_def_tab      MasterNodeDefinitionTable          (1.6)
```

The first would be accepted by any programming language, whereas the last one is most mnemonic.

In the present implementation the first choice (all-caps acronyms) is used because of its conciseness and portability. Longer identifiers are used in the naming of modules.

§1.3.5 Qualifiers

Some table data structures are prefixed by the qualifier *master*. For example, the Master Element Definition Table or `MEDT`. The qualifier means that the table contains sufficient information to reconstruct, by itself or with the help of other Master Tables, the properties of the indicated dataset. In the case of the MEDT, that table defines the elements of a complete FE model.

The "master" property is obviously critical as to deciding which data must be saved and moved *from one run to another*, or *from one computer to another*, without loss of information.

All data structures which are not qualified as master may be viewed as auxiliary or subordinate. Those data structures are derived, directly or indirectly, from master tables. But a qualifier such as "auxiliary" or "subordinate" need not be specifically given as part of the title. The absence of "master" is sufficient. Non-master data structures may normally be deleted without harm after they have served their purpose.

The term *state* appears in some data structures, and this has more of a technical connotation. A FE model is a discrete system. The state of a discrete system is a set of variables from which the internal behavior of the system at any point in space can be computed with the help of that and other information. Data structures that contain those variables are qualified by the term *state* in their title.

As a relevant example, the nodal displacements of a displacement-based FE model form a set of state variables. The stress at, say, an element center can be obtained from the state variables for that element, plus element definition data such as constitutive and fabrication properties obtained from Master Tables. The data structure that contains all node displacements (and other data such as node forces) is in fact called the Master Node State Table or `MNST`.

Source and results data structures organized according to the partition of the FEM model into subdomains are qualified by the terms *local* or *partitioned*. An example is the Local Element Definition Table, or `LEDT`.

The opposite of partitioned or local model is the *global* or *source* model. When there is need for explicit use of a qualifier to identify a global data structure, the term *global* is used and the name is prefixed by `G`.

Table 1.1 Conventions for Short Name Acronyms

Letter	Meaning
A	Activity, Address, Arrangement, Assembly
a	Acceleration (vector)
B	Bandwidth, Boundary, Built-in
b	Bodyforce (vector)
C	Configuration, Connection, Constitutive, Constraint
c	Constitutive (individual)
D	Definition
d	Deformation (vector)
E	Element
e	Element (individual)
F	Fabrication, Freedom, Flexibility, Fluid
f	Freedom (individual), Fabrication (individual)
G	Global, Generalized
g	Gradient (vector)
H	Heat, History
I	Identity, Initial, Individual, Interface
i	Internal
J	Energy**
j	Jump
K	Stiffness
k	Conductivity
L	Local, Load
M	Mass, Master, Matrix, Multiplier
m	Moment (vector), Multiplier (individual)
N	Node, Nonlinear
n	Node (individual)
O	Object
P	Partition, Potential, Property
p	Partition (individual), Momentum (vector), pointer
Q	Force as freedom conjugate*
q	Force (vector)
R	Response, Rigid
r	Rotation (vector)
S	Shock, Skyline, Spectrum, State, Structure, Subdomain
s	Subdomain (individual)
T	Table, Tag, Tangent, Temperature, Time
t	Translation (vector)
U	Unconstrained, Unified
u	Displacement (vector)
V	Valency, Vector, Vibration, Visualization
v	Velocity (vector)
W	Weight, Width
w	Frequencies*
X	Eigenvectors**
x	External
Y	Strain****
Z	Stress****

* To avoid clash with Freedom et al.
** To avoid clash with Element.
*** To avoid clash with Partition et al.
**** To avoid clash with Structure et al.

§1.3.6 Implementation Languages

The ease of implementation of flexible data structures depends on the implementation language. Generally there is a tradeoff effect between computational efficiency and human effort, and it is important to balance the two requirements. Two general requirements are:

- Many computational data structures can be implemented as arrays, but the size is only known at run time. Thus, any conventional programming language that supports dynamic storage management may be used.
- Source data structures may be best implemented as lists for maximum flexibility and to facilitate program evolution, because for the front end computational efficiency is not usually a major issue.

Following is a brief review of various programming languages for implementing FEM applications.

C, C++, Fortran 90. These languages directly support arrays as primitive objects as well as dynamic storage allocation. Lists are not directly supported as primitives and must be implemented as programmer-defined data types: structures in C, classes in C++, derived types in Fortran 90.

Matlab, Fortran 77. These languages do not support lists or the creation of derived data types, although Matlab handles dynamic storage allocation. Fortran 77 may be considered in kernel implementations if called from master C routines that handle dynamic resource allocation. This has the advantage of reuse of the large base of existing FE code. However, mixed language programming can run into serious transportability problems.

Mathematica. Because this language supports primitive and dynamic list operations at run time, the implementation of all data structures described here is straightforward. This simplicity is paid by run time penalties of order 100-10000. Hence *Mathematica* deserves consideration as an instructional and rapid prototyping tool, but should not be viewed as a production vehicle. It is in this spirit that the presently implementation is offered.

Java. This language deserves consideration as network programming becomes the dominant mode in the future. But it has not apparently been regarded as a contender in the scientific programming area because of its interpretive nature.

Automatic translation from *Mathematica* to Java may offer the ultimate solution to a combination of a rapid prototyping language for instruction and research exploration, with a highly portable and reasonably efficient language for numerical computation.

§1.4 WHAT'S UNIQUE ABOUT *MATHFET*

The present report discusses an implementation of the finite element method using *Mathematica*. The implementation is written as a set of modules collectively call *MathFET* which is an acronym for *Mathematica* implementation of a Finite Element Toolkit.

This is believed to be the first complete implementation of a general purpose finite element analysis in *Mathematica*. In addition, *MathFET* provides the following unique features:

1. A strict treatment of degrees of The minimum number of freedoms at each node needed to solve the problem is automatically used. For example, if the model contains only flat plate bending elements, three degrees of freedom are carried at each node. If some nodes of the model require an additional freedom, that freedom is carried there and nowhere else. This approach is made possible because of the use of list structures.
2. The toolkit approach. The user builds custom FEM program by calling toolkit functions. No unique closed program is provided; just examples. Source code is always available, and the

user is encouraged to write own contributions. This white-box approach ensures that programs can always contain the latest technology, avoiding the obsolescence typical of finite element black boxes.

3. Symbolic capabilities. Toolkit components may be used to conduct symbolic studies useful in certain applications.

-

2
Nodes

§2.1 GENERAL DESCRIPTION

Node points or *nodes* are selected space locations that serve two functions:

(i) To define the geometry of the elements and hence that of the finite element model.

(ii) To provide "resident locations" for the degrees of freedom. These freedoms specify the state of the finite element model.

Nodes are defined by giving their coordinates with respect to a rectangular Cartesian coordinate system (x, y, z) called the *global system*. See Figure 2.1.

Attached to each node `n` there is a local Cartesian coordinate system $\{\bar{x}_n, \bar{y}_n, \bar{z}_n\}$, which is used to specify freedom directions and is called the Freedom Coordinate System or FCS. The definition of the FCS is not part of the data structures introduced in this Chapter, because that information pertains to the freedom data structures. It is treated in §7.2.5.

Nodes are classified into *primary* and *auxiliary*. Primary nodes or P-nodes do double duty: specification of element geometry as well as residence for degrees of freedom. Auxiliary nodes or A-nodes are used only for geometric purposes and bear no degrees of freedom.

§2.1.1 Position and Direction Nodes

Another classification of nodes distinguishes between *position* and *orientation* or *direction* nodes. The former are points with finite coordinates. The latter are directions or "points at infinity" which are often used to define local coordinate systems. In the data structures described here *position and orientation nodes are placed in the same table and are treated by the same methods*. This unification is possible thanks to the use of homogeneous nodal coordinates: four numbers may be given instead of three.

Although orientation nodes are usually of auxiliary type, sometimes they carry degrees of freedom and thus are categorized as primary. This situation occurs in the definition of infinite elements (in topic treated in Advanced Finite Element Methods), which have node points at infinity.

§2.1.2 External Identification

Nodes are defined by the user. They are identified by unique positive numbers in the range

$$\texttt{1} \text{ through } \texttt{laxnod} \tag{2.1}$$

where `laxnod`, which stands for *largest external node*, is a problem parameter. These identifiers are called *external node numbers*. All communication with the user employs external numbers.

When it is important to distinguish external node numbers from the internal node numbers defined below, the former will be shown as **boldface** numbers, as in the examples (2.2) and (2.4).

External node numbers need not be consecutive. That is, "gaps" may appear. For example, the user may define 6 nodes numbered

$$\textbf{2, 4, 5, 8, 14, 18} \tag{2.2}$$

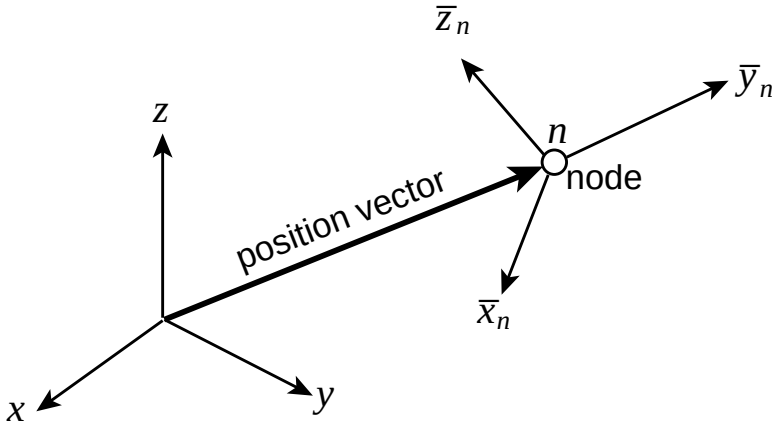


Figure 2.1. Location of a node `n` in 3D space x, y, z.

Then `laxnod` is **18** but the *number of defined nodes*, called `numnod`, is 6. In programming, external node numbers are consistently identified by symbols

`xnode` (as variable), `xn` (as loop or array index) (2.3)

§2.1.3 Internal Identification

External nodes are stored consecutively in tables for use by the program. The index for table retrieval is called *internal node number*. For the previous example:

External :	**2**	**4**	**5**	**8**	**14**	**16**
Internal :	1	2	3	4	5	6

(2.4)

Internal node numbers are usually identified in programs by

`inode` or `in` (as variable), `n` (as loop or array index) (2.5)

§2.1.4 *Node Types

Primary nodes can be further classified according to their relationship with element geometries:

Corner nodes or C-nodes. Located at corners of two- and three-dimensional elements. End nodes of one-dimensional elements and the single node of zero-dimensional elements are conventionally classified as corner nodes.

Side nodes or S-nodes. Located on edges of two- and three-dimensional elements. Midpoint nodes of one-dimensional elements are conventionally classified as side nodes. Zero-dimensional elements have no side nodes.

Face nodes or F-nodes. Located on faces of three-dimensional elements. Interior nodes of two-dimensional elements are conventionally classified as face nodes, although in two dimensional analysis they would be more logically classified as disconnected nodes. One- and zero-dimensional elements have no face nodes.

Table 2.1 Node type identifiers

Letter	Meaning
missing	Type is unknown
A	Auxiliary node
C	Corner node
S	Side node
F	Face node
D	Disconnected node

Disconnected nodes or D-nodes. Located in the interior of three-dimensional elements. Two-, one- and zero-dimensional elements have no disconnected nodes.

This classification assigns each node an attribute called *type*. Nodes are usually, but not always, defined before elements. The type classification cannot be completed until the element information is available. Until that is done, the node type is said to be *unknown*.

The node type is identified by a letter as shown in Table 2.1. The type is set to blank until the element information is available.

§2.2 THE MASTER NODE DEFINITION TABLE

§2.2.1 Configuration

The Master Node Definition Table or MNDT, defines the location of each node through the coordinates stored in the table. It also has slots for storing node types. This data structure is a list of numnod sublist items:

```
MNDT = { nDL(1), nDL(2) ... nDL(n) ... nDL(numnod) }
```

Each of the MNDT components is a list called the Individual Node Definition List or nDL. The nDL of internal node n is the n^{th} item in the MNDT.

§2.2.2 The Individual Node Definition Table

The nDL(n) is a three-item list:

$$\{\texttt{nx, clist, type}\} \tag{2.6}$$

The second and third items are lists with the configuration

$$\texttt{clist} = \{x_n, y_n, z_n\} \text{ or } \{x_n, y_n, z_n, w_n\} \tag{2.7}$$

$$\texttt{type} = \{\,\} \text{ or } \{\texttt{type}\} \text{ or } \{\texttt{type, pMSGT}\} \tag{2.8}$$

where the items shown are now primitive with the possible exception of pMSGT. The meaning of those items is as follows.

nx The external node number for the internal node n.

Table 2.2 Node Type Configuration

`type`	Meaning
{ }	Type is undefined
{ `type` }	Type as per Table 2.1, coordinates given
{ `type,pMSGT` }	Type as per Table 2.1, coordinates acquired from MSGT*

* `pMSGT` is a pointer to a Master-Slave Geometric Table, see Section 2.3.2

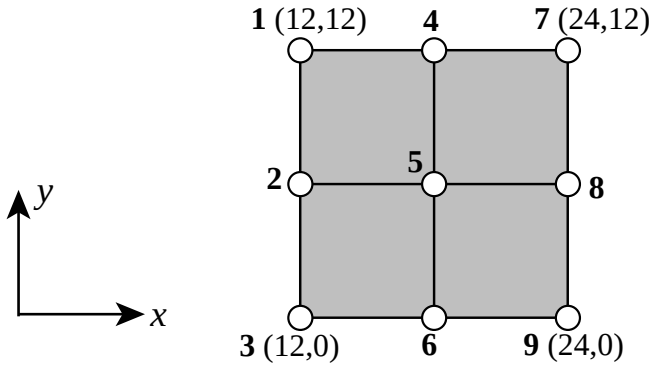


Figure 2.2. Mesh for node definition example.

x_n, y_n, z_n, w_n Homogeneous position coordinates of internal node `n`. If w_n is given and is nonzero, the node coordinates are x_n/w_n, y_n/w_n and z_n/w_n. If w_n is omitted, as in the form $\{ x_n, y_n, z_n \}$, $w_n = 1$ is assumed. If w_n appears and is zero the node is located at infinity and x_n, y_n, z_n specify its direction coordinates.

`type` Configuration defined in Table 2.2.

EXAMPLE 2.1

Consider the 9-node regular 2D mesh shown in Figure 2.2. Before the node type is known, the `MNDT` has the form

```
MNDT = {{1,{12,12,0},{}}, {2,{12,6,0},{}}, {3,{12,0,0},{}},
        {4,{18,12,0},{}}, {5,{18,6,0},{}}, {6,{18,0,0},{}},
        {7,{24,12,0},{}}, {8,{24,6,0},{}}, {9,{24,0,0},{}}}
```
(2.9)

After the element data is processed all nodes are known to be of corner type. Consequently the `MNDT` becomes

```
MNDT = {{1,{12,12,0},{"C"}}, {2,{12,6,0},{"C"}}, {3,{12,0,0},{"C"}},
        {4,{18,12,0},{"C"}}, {5,{18,6,0},{"C"}}, {6,{18,0,0},{"C"}},
        {7,{24,12,0},{"C"}}, {8,{24,6,0},{"C"}}, {9,{24,0,0},{"C"}}}
```
(2.10)

§2.3 AUXILIARY DATA STRUCTURES

§2.3.1 The External To Internal Node Mapping Table

Given an internal node number `n`, the external number xn may be immediately extracted from `nDL(n)`. Often it is necessary to perform the inverse operation: get `n` given `nx`. That is: given an external node number, find its slot in the `MNDT`.

Searching the `MNDT` would be inefficient, because the user numbers therein are not necessarily ordered. Thus a binary search is excluded and a time-consuming linear search would be required. It is therefore convenient to prepare a separate table that directly maps external to internal nodes. This table is an array called `Nx2i`, which contains `laxnod` integer entries:

```
Nx2i = { Nx2i(1) ... Nx2i(laxnod) }
```
(2.11)

such that `n = Nx2i(nx)`. If external node number `nx` is undefined, a zero is returned.

This table is normally kept separate from the `MNDT`. In languages that support dynamic storage allocation, `Nx2i` may be constructed "on the fly" on entry to a subroutine or module where such mapping is necessary. On exit the storage is released.

EXAMPLE 2.2

For the correspondence (2.8):

```
Nx2i = { 0, 1, 0, 2, 3, 0, 0, 4, 0, 0, 0, 0, 0, 5, 0, 6 }
```
(2.12)

§2.3.2 *The Master-Slave Geometric Table

In advanced FEM implementations nodes may be classified into *master* or *slaves* as regard their geometrical definition. Coordinates of master nodes are specified directly. Coordinates of slave nodes are computed indirectly, as functions of the coordinates of master nodes.

To give an example, suppose that nodes 2 and 3 are at the thirdpoints of the line segment defined by end nodes 1 and 4. Once 1 and 4 are defined, the coordinates of 2 and 3 can be computed according to the rules

$$\begin{aligned} x_2 &= \tfrac{1}{3}x_1 + \tfrac{2}{3}x_4, \quad y_2 = \tfrac{1}{3}y_1 + \tfrac{2}{3}y_4, \quad z_2 = \tfrac{1}{3}z_1 + \tfrac{2}{3}z_4, \quad w_2 = \tfrac{1}{3}w_1 + \tfrac{2}{3}w_4, \\ x_3 &= \tfrac{2}{3}x_1 + \tfrac{1}{3}x_4, \quad y_3 = \tfrac{2}{3}y_1 + \tfrac{1}{3}y_4, \quad z_3 = \tfrac{2}{3}z_1 + \tfrac{1}{3}z_4, \quad w_3 = \tfrac{2}{3}w_1 + \tfrac{1}{3}w_4, \end{aligned} \tag{2.13}$$

Such rules are specified in the *Master Slave Geometric Table* or `MSGT`, and are activated through the pointer(s) in `type`. The configuration of the `MSGT` is not defined in this document because it is an advanced feature not required for an initial FEM implementation.

Cell 2.1 Definition of Individual Node

```
DefineIndividualNode[MNDT_,nDL_]:= Module [
  {ndt=MNDT,n,nn,numnod,xn},
   numnod=Length[ndt]; xn=nDL[[1]]; nn=0;
   Do [ If [MNDT[[n,1]]==xn, nn=n; Break[] ], {n,1,numnod}];
   If [nn==0, AppendTo[ndt,nDL], ndt[[nn]]=nDL];
   Return[ndt];
 ];

MNDT= {};
MNDT= DefineIndividualNode[MNDT, {1,{1,2,-3.,1.},{}} ];
MNDT= DefineIndividualNode[MNDT, {4,{64,4,-8.,1},{}} ];
MNDT= DefineIndividualNode[MNDT, {12,{1,1,1,1},  {}} ];
MNDT= DefineIndividualNode[MNDT, {4,{x4,4,-8.,1},{}} ];
MNDT= DefineIndividualNode[MNDT, {2,{1,2,1,0},   {}} ];
Print["MNDT=",MNDT//InputForm];
PrintMasterNodeDefinitionTable[MNDT];

(*
nmax=100; MNDT={}; Nx2i=Table[0,{nmax}];
Print[Timing[Do [
   MNDT= DefineIndividualNode[MNDT, {n,{N[n-1],0,0},{}}],
  {n,1,nmax}]]];
PrintMasterNodeDefinitionTable[MNDT];*)
```

Cell 2.2 Output from the Program of Cell 2.1

```
MNDT={{1, {1, 2, -3., 1.}, {}}, {4, {x4, 4, -8., 1}, {}},

   {12, {1, 1, 1, 1}, {}}, {2, {1, 2, 1, 0}, {}}}
Xnode        x         y         z         w    typ  pMSGT
    1   1.00000   2.00000  -3.00000
    4        x4   4.00000  -8.00000
   12   1.00000   1.00000   1.00000
    2   1.00000   2.00000   1.00000   0.00000
```

§2.4 IMPLEMENTATION OF MNDT OPERATIONS

This section presents *Mathematica* modules that implement basic operations pertaining to finite element nodes and the MNDT.

§2.4.1 Defining an Individual Node

Cell 2.3 External to Internal Node Mapping

```
MakeNodeExternalToInternal[MNDT_]:= Module[
  {laxnod,n,Nx2i,numnod=Length[MNDT],xn},
   If [numnod==0, Return[{}]];
   laxnod=0; Do [laxnod=Max[laxnod,MNDT[[n,1]]], {n,1,numnod}];
   Nx2i=Table[0,{laxnod}];
   Do [xn=MNDT[[n,1]]; Nx2i[[xn]]=n, {n,1,numnod}];
   Return[Nx2i]
 ];

 MNDT={{17,{1./81,3,4,1},{" "}}, {6,{1234,69.5678,-134.8,0},{"D"}},
       {12,{.1234,.2345,.4567},{"C"}}};
 Print[MakeNodeExternalToInternal[MNDT]];
```

Cell 2.4 Output from the Program of Cell 2.3

```
{0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 1}
```

Module `DefineIndividualNode` is displayed in Cell 2.1. It defines a new node by inserting its `nDL` in the `MNDT`. If the external node number is already in that table, the information is replaced by the new one; else it is appended to the table. The input arguments are the existing `MNDT` and the `eNL`. The module returns the updated `MNDT`.

The statements following `DefineIndividualNode` in Cell 2.1 test the module by building a table with several nodes, which is then printed with the module `PrintMasterNodeDefinitionTable` described below. The output is shown in Cell 2.2.

§2.4.2 External to Internal Node Mapping

Module `MakeNodeExternalToInternal`, listed in Cell 2.3, constructs the External To Internal Node Mapping Table `Nx2i` described in §2.3.2. It receives the `MNDT` as input and returns `Nx2i`.

The code is tested by the statements that follow the module, and the results of running the test program are shown in Cell 2.4.

In practice this module is rarely used as an individual entity, as is just as easy to "inline" the code into the modules that need to construct `Nx2i` on the fly. It is shown here for instructional purposes only.

Cell 2.5 Printing the Master Node Definition Table

```
PrintMasterNodeDefinitionTable[MNDT_]:= Module[
  {numnod=Length[MNDT],t,n,xn,xnd,c,typlist,type},
   t=Table["    ",{numnod+1},{7}];
   Do [xn=MNDT[[n,1]];
        c=MNDT[[n,2]]; If [Length[c]==3, c=AppendTo[c,1]];
       typlist=MNDT[[n,3]];
       t[[n+1,1]]=ToString[xn];
       t[[n+1,2]]=PaddedForm[c[[1]]//FortranForm,{6,5}];
       t[[n+1,3]]=PaddedForm[c[[2]]//FortranForm,{6,5}];
       t[[n+1,4]]=PaddedForm[c[[3]]//FortranForm,{6,5}];
       If [c[[4]]!=1,t[[n+1,5]]=PaddedForm[c[[4]]//FortranForm,{6,5}]];
       If [Length[typlist]>0,t[[n+1,6]]=typlist[[1]]];
       If [Length[typlist]>1,t[[n+1,7]]=ToString[typlist[[2]] ]],
   {n,1,numnod}];
   t[[1]] = {"Xnode","x ","y ","z ","w ","typ","pMSGT"};
   Print[TableForm[t,TableAlignments->{Right,Right},
           TableDirections->{Column,Row},TableSpacing->{0,2}]];
   ];

 MNDT={{17,{1./81,3,4,1},{"?"}}, {2,{123456,69345.5678,-134567.8,0},
         {"D"}},
       {1513,{.1234,.2345,.4567},{"C"}}};
 PrintMasterNodeDefinitionTable[MNDT];
```

Cell 2.6 Output from the Program of Cell 2.5

```
Xnode             x             y             z         w   typ  pMSGT
   17       0.01235       3.00000       4.00000             ?
    2   123456.00000   69345.60000  -134568.00000   0.00000    D
 1513       0.12340       0.23450       0.45670             C
```

§2.4.3 Printing the MNDT

Module `PrintMasterNodeDefinitionTable`, listed in Cell 2.5, prints the `MNDT` in a tabular format. Its only input argument is the `MNDT`. The module is tested by the statements that follow it, which build an MNDT directly and print it.

Unlike conventional languages such as Fortran or C, printing data structures in a tabular format is quite involved in *Mathematica* because of the tendency of the language to print in free-field. Thus if one attempts to print the `MNDT` node by node, non-aligned display of coordinates is likely to happen.

Cell 2.7 Setting the Type Attribute in the MNDT

```
SetTypeInMasterNodeDefinitionTable[MNDT_,MEDT_]:= Module[
  {ndt=MNDT,e,i,k,knl,lenenl,lenknl,n,nodtyp,numele,numnod,nx2i,xn},
   numnod=Length[ndt];  k=0;
   Do [k=Max[k,ndt[[n,1]]]; If [Length[ndt[[n,3]]]==0, ndt[[n,3]]={" "}],
   {n,1,numnod}];
   nx2i=Table[0,{k}]; Do [xn=ndt[[n,1]]; nx2i[[xn]]=n, {n,1,numnod}];
   nodtyp={"C","M","F","I"}; numele=Length[MEDT];
   Do [If [MEDT[[e,2,1]]=="MFC", Continue[]];
       enl=MEDT[[e,3]]; lenenl=Length[enl];
      Do [knl=enl[[k]]; lenknl=Length[knl];
         Do [xn=knl[[i]]; If [xn<=0, Continue[]]; n=nx2i[[xn]];
           If [n>0, ndt[[n,3,1]]=nodtyp[[k]]],
         {i,1,lenknl}],
      {k,1,lenenl}],
   {e,1,numele}];
   Return[ndt];
 ];
 MNDT={{1,{x1,y1,z1},{"?"}}, {3,{x3,y3,z3},{" "}},
       {6,{x6,y6,z6},{" "}}, {8,{x8,y8,z8},{" "}},
       {2,{x2,y2,z2},{" "}}, {11,{x11,y11,z11},{"U"}},
       {5,{x5,y5,z5},{" "}}, {7,{x7,y7,z7},{" "}}};
 PrintMasterNodeDefinitionTable[MNDT];
 MEDT={ {"S.1",{"QUAD.4"},{{1,3,8,6},{2,11,5,7},{},{}},{5,2}}};
 MNDT=SetTypeInMasterNodeDefinitionTable[MNDT,MEDT];
 PrintMasterNodeDefinitionTable[MNDT];
```

Cell 2.8 Output from the Program of Cell 2.7

```
Xnode   x    y    z     w   typ  pMSGT
    1   x1   y1   z1          ?
    3   x3   y3   z3
    6   x6   y6   z6
    8   x8   y8   z8
    2   x2   y2   z2
   11  x11  y11  z11          U
    5   x5   y5   z5
    7   x7   y7   z7
Xnode   x    y    z     w   typ  pMSGT
    1   x1   y1   z1          C
    3   x3   y3   z3          C
    6   x6   y6   z6          C
    8   x8   y8   z8          C
    2   x2   y2   z2          M
   11  x11  y11  z11          M
    5   x5   y5   z5          M
    7   x7   y7   z7          M
```

This accounts for the rather elaborate code shown in Cell 2.5. Essentially a character copy of the complete `MNDT` is built in array `t`, which is then displayed in *Mathematica*'s TableForm.

§2.4.4 Setting Node Types

The last module presented here is `SetTypeInMasterNodeDefinitionTable`, which is listed in Cell 2.7. It stores the node type attribute discussed in §2.3.2, in the MNDT. Setting up that attribute requires element definition information; specifically the Master Element Definition Table or `MEDT`, which is supplied as second argument. Thus, execution of this module cannot be done until both the `MNDT` and `MEDT` are concurrently available.

The module is tested by the statements shown in Cell 2.6, which built a MNDT and a MEDT, execute the module and print the MNDT before and after execution. The outputs are shown in Cell 2.8.

-

3

Individual Elements

Elements are the fundamental entities in the Finite Element Method. They specify the topological, constitutive and fabrication properties of a finite element model.

Unlike nodes, there is a great variety of data associated with elements. Because of that variety, the coverage of element data structures is spread over three Chapters: 3, 4and 5. This Chapter deals with the identification of *individual* elements. This data can be grouped into five pieces: identifier, type, nodes, material, and properties. These are described in the following subsections.

§3.1 ELEMENT IDENTIFICATION

Each element has an external identifier, which is assigned by the user, and an internal identifier, which is assigned by the program.

§3.1.1 External Identifiers

Elements are externally identified by a (name,number) pair: an object name, and an external number. The latter is required while the former may be blank.

The *object name* is a character string that specifies the structural object to which the element belongs. For example:

```
"RightWing"      "RightWing.Flap4"                    (3.1)
```

This character string should have a convenient maximum length to accomodate readable descriptions; for example 80 characters. It is case dependent: `"wing"` is not the same as `"Wing"`. Sometimes it is useful to name objects within objects. This is done by writing strings separated by periods. In the second example, `"RightWing.Flap4"` is a composite object in which `"Flap4"` is a component, in the substructure sense discussed below, of `"RightWing"`.

For simple structures a separation into objects may not be necessary. If so this character string may be left empty and only an element external number given.

The *external element number* is an integer in the range

```
1  through  maxele                                    (3.2)
```

where `maxele` is a program parameter that specifies the maximum element number which may be assigned to an object. This number is denoted by `xe` or `xele` in programming. The element sequence may have gaps. For example, suppose that six elements for object name `"RightWing.Flap4"` are defined with numbers

```
1, 4, 5, 6, 11, 12                                    (3.3)
```

There are two gaps: 2–3, and 7–10, in the sequence for the `"RightWing.Flap4"` object. The last element number defined, here 12, is called the *last external element* for a particular object and is usually called `laxele` in programming.

§3.1.2 Internal Element Numbers

As each element is defined, it is assigned an *internal element number*. This is an integer in the range

```
1 through numele                                      (3.4)
```

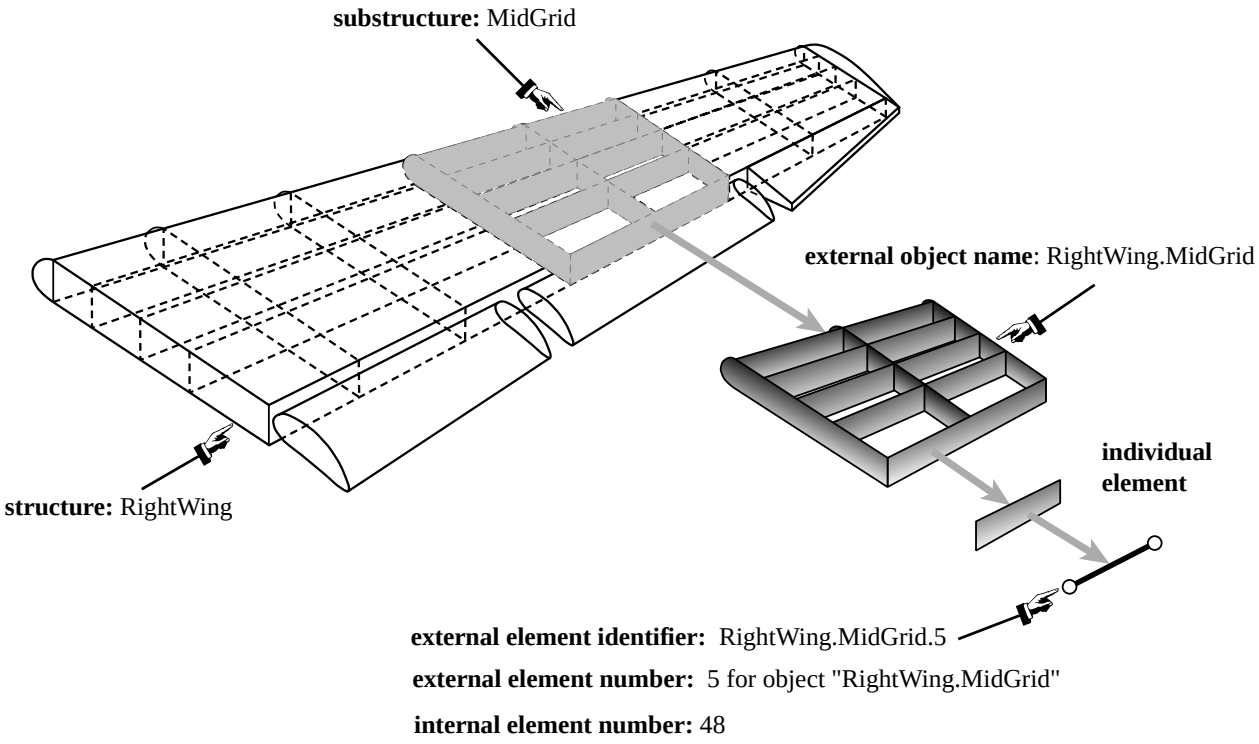


Figure 3.1. Example illustrating the components of an external element identification. The structure of the right wing of an airplane is, not surprisingly, called `RightWing`. The shaded substructure of ribs and spars is `MidGrid`. From it an individual spar element, numbered 5, is distinguished. Its internal number is taken to be 48. That is, `RightWing.MidGrid.5` happens to be the 48th element defined by the user. Note that dots may be replaced by commas in an actual implementation to simplify parsing.

where `numele` is the total number of defined elements. There are no gaps. *Internal element numbers are assigned once and for all and never change*. They are usually denoted by `e`, `ie` or `iele` in programming.

External and internal element numbers must be carefully distinguished. They only coalesce when there is only one object name (which is then often empty) and external element numbers are assigned in ascending sequence.

§3.1.3 Substructures

An *external substructure* or simply *substructure* is the set of elements that pertains to the same object name. Multilevel structuring can be easily implemented with the dot notation. For many structures one level is sufficient. The ability to use multiple levels is convenient, however, for mesh generation and visualization purposes.

See Figure 3.1 for a two-level substructure example.

§3.1.4 *Subdomains

To facilitate efficient processing on parallel computers, elements are grouped into sets called *subdomains*, which are created by software tools called *domain decomposers*. Subdomains may span objects and also be empty, that is, devoid of elements. Subdomains may overlap; that is, an element may belong to more than one subdomain.

A subdomain is identified by a positive integer in the range

$$\texttt{1} \text{ through } \texttt{numsub} \tag{3.5}$$

where `numsub` is the number of subdomains. This sequence has no gaps. The subdomain number is usually carried in variables

$$\texttt{s} \text{ (in subscripts or loops), } \texttt{sub} \text{ (otherwise)} \tag{3.6}$$

Individual elements within a non-empty subdomain are identified by an *subdomain element number* in the range

$$\texttt{1} \text{ through } \texttt{nelsub(s)} \tag{3.7}$$

where `nelsub(s)` is the number of elements in subdomain `s`. Again this sequence has no gaps. If the $\texttt{s}^{th}$ subdomain is empty, `nelsub(s)=0`.

The subdomain element number is usually carried in variables

$$\texttt{se} \text{ in arrays or loops, } \texttt{sele} \text{ otherwise} \tag{3.8}$$

The connection between external and subdomain identifiers is effected through the internal element number, as described in Chapter 3.

REMARK 3.1

The concepts of *subdomains* and *substructures* has had historically many points in common, and most authors regard them as identical. In the present exposition a logical distinction is established: *a substructure is an external concept* whereas *a subdomain is an internal concept*. More specifically:

1. Substructures are defined by the user through specification of a common object name. This kind of grouping is useful for reporting and visualization.
2. Subdomains are defined by domain decomposers on the basis of efficiency for parallel processing. This breakdown may have to obey certain rules dictated by the solution procedure. For example, FETI solution methods may require that subdomains cannot have zero-energy modes other than rigid-body motions. A necessary (but not sufficient) condition to fulfill this requirement is that subdomain elements must be connected. On the other hand, substructures can contain any combination of elements whatsoever, whether connected or not.
3. Domain decomposers may be forced to respect object boundaries. This is common in coupled field analysis, since decompositions of different field, such as fluid and structure, may be directed to different solvers.

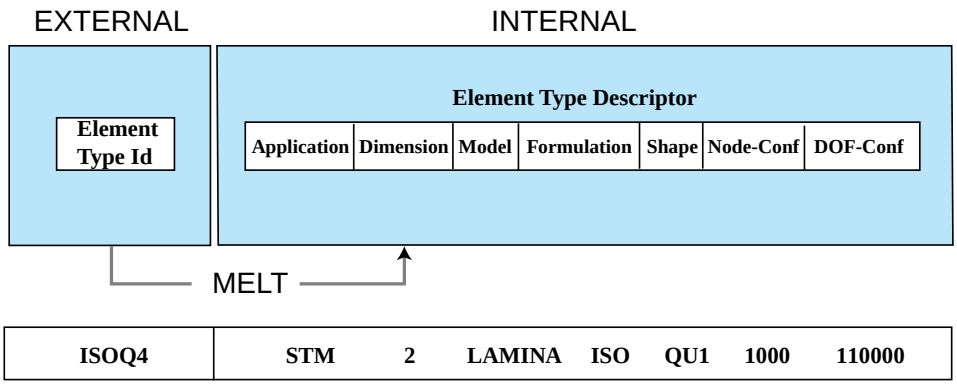


Figure 3.2. Element type name and descriptor: (a) the name is external and specified by the user whereas the descriptor is internal and only seen by the program; (b) an example.

§3.2 ELEMENT TYPE

The element *type* describes its function in sufficient detail to the FE program so that it can process the element through the appropriate modules or subroutines.

The element type is concisely defined by the user through an *element type name*. A name may be viewed as an external representation of the type. Legal type names are paired trough an internal table called the Master Element Library Table or `MELT`, with with an internal list of attributes collectively called the *element type descriptor*.

§3.2.1 Element Type Name

The *element type name*, or simply *element name*, is a character string through which the user specifies the function of the element, as well as some distinguishing characteristics.

The complexity of this name depends on the level of generality of the underlying FE code. As an extreme case, consider a simple code that uses one and only one element type. No name is then required.

Most finite element codes, however, implement several element types and names appear. Some ancient programs use numeric codes such as `103` or `410`. More common nowadays is the use of character strings with some mnemonic touches, such as `"QUAD9"` or `"BEAM2"`.

In the present scheme an element type name is assumed. The choice of names is up to the program developer. The name implicitly define the element type descriptor, as illustrated in Figure 3.1.

The connection of the type name in the `MEDT` and and the descriptor in the `MELT` is effected through a *type index*, which is a pointer to the appropriate entry of the `MELT`.

§3.2.2 Element Type Descriptor

The element *type descriptor* or simply *element descriptor* is a list that contains seven components

Table 3.1 Application specification in element type descriptor (examples)

Identifier	Application
`"ACF"`	Acoustic fluid
`"EMM"`	Electromagnetic medium
`"EUF"`	Euler fluid
`"NSF"`	Navier-Stokes fluid
`"RBM"`	Rigid body mechanics
`"STM"`	Structural mechanics
`"THM"`	Thermomechanical

that collectively classify the element in sufficient detail to be routed, during the processing phases, to the appropriate element formation routines. The descriptor does *not* include information processed within the routine, such as node coordinates, materials and fabrication data.

The seven components are either items or sublists, and appear in the following order:

```
descriptor = {application, dimensionality, model, formulation,
              geometric-shape, node-configuration, freedom-configuration }
```
(3.9)

See Figure 3.2. In programming, these may be identified by names such as:

```
eDL = {eDapp, eDdim, eDmod, eDfrm, eDsha, eDnod, eDdof }
```
(3.10)

or similar conventions.

As noted above, the connection between element type names and descriptor is done through a built-in data structure called the Master Element Library Table or `MELT`. This data structure is updated by the program developer as elements are added to the library. The organization of the `MELT` is described in Chapter 4.

§*3.2.2.1 Application*

The *application* item is a 3-character identification tag that explains what kind of physical problem the element is used for. The most important ones are listed in Table 3.1. The data structures described here emphasize Structural Mechanics elements, with tag `"STM"`. However, many of the attributes apply equally to the other applications of Table 3.1, if modeled by the finite element method.

§*3.2.2.2 Dimensionality*

The *dimensionality* item is an integer that defines the number of intrinsic space dimensions of a mechanical element: 0, 1, 2 or 3. Multipoint contraint and multifreedom constraint elements are conventionally assigned a dimensionality of 0.

Table 3.2 Structural Mechanics model specification in element type descriptor

Appl	Dim	Identifier	Mathematical model
"STM"	0	"POINT"	Point
"STM"	0	"CON"	Multipoint constraint
"STM"	0	"CON"	Multifreedom constraint
"STM"	1	"BAR"	Bar
"STM"	1	"PBEAM0"	C^0 plane beam
"STM"	1	"PBEAM1"	C^1 plane beam
"STM"	1	"SBEAM0"	C^0 space beam
"STM"	1	"SBEAM1"	C^1 space beam
"STM"	1	"CABLE"	Cable
"STM"	1	"ARCH"	Arch
"STM"	1	"FRUST0"	C^0 axisymmetric shell
"STM"	1	"FRUST1"	C^1 axisymmetric shell
"STM"	2	"LAMIN"	Plane stress (membrane)
"STM"	2	"SLICE"	Plane strain slice
"STM"	2	"RING"	Axisymmetric solid
"STM"	2	"PLATE0"	C^0 (Reissner-Mindlin) plate
"STM"	2	"PLATE1"	C^1 (Kirchhoff) plate
"STM"	2	"SHELL0"	C^0 shell
"STM"	2	"SHELL1"	C^1 shell
"STM"	3	"SOLID"	Solid

This attribute should not be confused with spatial dimensionality, which is 2 or 3 for 2D or 3D analysis, respectively. Spatial dimensionality is the same for all elements. For example a bar or beam element has intrinsic dimensionality of 1 but may be used in a 2D or 3D context.

§*3.2.2.3 Model*

The *model* item is linked to the application and dimensionality attributes to define the mathematical model used in the element derivation. It is specified by a character string containing up to 6 characters. Some common models used in Structural Mechanics applications are alphabetically listed in Table 3.2.

In that Table, 0/1 after bending models identifies the so-called C^0 and C^1 formulations, respectively. For example "PLATE1" is another moniker for Kirchhoff plate or thin plate. "LAMINA" and "SLICE" are names for 2D elements in plane stress and plane strain, respectively. "FRUS0" and "FRUS1" are axisymmetric shell elements obtained by rotating C^0 and C^1 beams, respectively, 360 degrees. "RING" is an axisymmetric solid element.

Other mathematical model identifiers can of course be added to the list as program capabilities expand. One common expansion is through the addition of *composite elements*; for example a stiffened shell or a girder box.

Table 3.3 Structural Mechanics formulation specification in element type descriptor

Identifier	Formulation
"ANS"	Assumed Natural Strain
"AND"	Assumed Natural Deviatoric Strain
"EFF"	Extended Free Formulation
"EXT"	Externally supplied: used for MFCs
"FRF"	Free Formulation
"FLE"	Flexibility (assumed force)
"HET"	Heterosis
"HYB"	Hybrid
"HYP"	Hyperparametric
"ISO"	Isoparametric
"MOM"	Mechanics of Materials
"SEM"	Semi-Loof
"TEM"	Template (most general of all)
"TMX"	Transfer Matrix

§*3.2.2.4 Formulation*

The *formulation* item indicates, through a three-letter string, the methodology used for deriving the element. Some of the most common formulation identifiers are listed in Table 3.3.

§*3.2.2.5 Geometric Shape*

The geometric shape, in conjunction with the intrinsic dimensionality attribute, identifies the element geometry by a 3-character string. Allowable identifiers, paired with element dimensionalities, are listed in Table 3.4.

Not much geometrical variety can be expected of course in 0 and 1 dimensions. Actually there are two possibility for zero dimension: "DOT" identifies a point element (for example, a concentrated mass or a spring-to-ground) whereas "CON" applies to MPC (MultiPoint Constraint) and MFC (MultiFreedom Constraint) elements.

In two dimensions triangles and quadrilaterals are possible, with identifiers "TR" and "QU" respectively, followed by a number. In three dimensions the choice is between tetrahedra, wedges and bricks, which are identified by "TE", "WE" and "BR", respectively, also followed by a number. The number following the shape identifier specifies whether the element is constructed as a unit, or as a macroelement assembly.

The "ARB" identifier is intended as a catch-all for shapes that are too complicated to be described by one word.

Table 3.4 Shape specification in element type descriptor

Dim	Identifier	Geometric shape
0	"DOT"	Point; e.g. a concentrated mass
0	"CON"	Constraint element (conventionally)
1	"SEG"	Segment
2	"TR1"	Triangle
2	"QU1"	Quadrilateral
2	"QU2"	Quadrilateral built of 2 triangles
2	"QU4"	Quadrilateral built of 4 triangles
3	"TE1"	Tetrahedron
3	"WE1"	Wedge
3	"WE3"	Wedge built of 3 tetrahedra
3	"BR1"	Brick
3	"BR5"	Brick built of 5 tetrahedra
3	"BR6"	Brick built of 6 tetrahedra
2-3	"VOR"	Voronoi cell
1-3	"ARB"	Arbitrary shape: defined by other attributes

§*3.2.2.6 Element Nodal Configuration*

The node configuration of the element is specified by a list of four integer items:

```
node-configuration := { eCNX, eSNX, eFNX, eINX }
```
(3.11)

Here `eCNX`, `eSNX`, `eFNX` and `eINX`, are abbreviations for element-corner-node-index, element-side-node-index, element-face-node-index and element-internal-node-index, respectively. Their value ranges from 0 through 2 with the meaning explained in Table 3.5. For storage and display convenience, the four indices are often decimally packed as one integer:

```
eCSFIX = 1000*eCNX + 100*eSNX + 10*eFNX + eINX
```
(3.12)

Note that the full set of indices only applies to elements with intrinsic dimensionality of 3. If dimensionality is 2 or less, `eINX` is ignored. If dimensionality is 1 or less, `eFNX` and `eINX` are ignored. Finally if dimensionality is 0, all except `eCNX` are ignored. Some specific examples:

20-node triquadratic brick	`ECSFI=1100`
27-node triquadratic brick	`ECSFI=1111`
64-node tricubic brick	`ECSFI=1222 (64=8*1+12*2+6*4+8)`
8-node serendipity quad	`ECSFI=1100`
9-node biquadratic quad	`ECSFI=1110`
10-node cubic lamina (plane stress) triangle	`ECSFI=1210 (10=3*1+3*2+1)`

Table 3.5 Node configuration in element type descriptor

Dim	Indices	Meaning/remarks
1–3	`eCNX=0,1`	element has `eCNX` nodes per corner
1–3	`eSNX=0,1,2`	element has `eSNX` nodes per side
2–3	`eFNX=0,1,2`	if `eFNX=0`, no face nodes
		if `eFNX=1`, one node per face
		if `eFNX=2`, as many face nodes as face corners
3	`eFNX=0,1,2`	if `eINX=0`, no face nodes
		if `eINX=1`, one node per face
		if `eINX=2`, as many face nodes as face corners
0	`eCSFIX=1000`	all nodes are treated as corners
Some exotic node configurations are omitted; cf. Remark 3.2		

REMARK 3.2

Note that the meaning of ”interior nodes” here does not agree with the usual FEM definition except for 3D elements. A plane stress element with one center node has `eFNX=1` and `eINX=0` instead of `eFNX=0` and `eINX=1`. In other words, that *center node is considered a face node*. Similarly a bar or beam element with two nodes at its third-points has `eSNX=2`, `eFNX=eINX=0`; that is, those nodes are considered side nodes. As explained next, the reason for this convention is mixability.

Why use indices instead of actual node counts? Mixability tests (connecting different elements together) are considerably simplified using this item. More specifically, elements with same application, model, formulation, `eCNX`, `eSNX`, `eFNX`, (in 3D) and `eCNX`, `eSNX` (in 2D) can be mixed if, in addition, some geometric compatibility conditions are met, while dimensionality and geometric shape attributes may be different. [Note that `eINX` and `eFNX` are irrelevant in 3D and 2D, respectively.] Two examples:

(i) Two solid elements with ECSFI=1220, 1221 and 1222 are candidates for mixing because interior nodes are irrelevant to such mating. This is true regardless of whether those elements are bricks, wedges or tetrahedra. Of course the common faces must have the same geometric shape; this represents another part of the mixability test.

(ii) Mixability across different dimensionalities is often of interest. For example, a 3-node bar with ECSFI=1100 attached along the edge of a quadrilateral or triangular lamina with `ECSFI=1100` or `1110`. Similarly a 2-node beam, which has `ECSFI=1000`, may be attached to a plate with `ECSFI=1000` but not to one with `ECSFI=1100`.

Values other than those listed in Table 3.5 are reserved for exotic configurations. For example, elements with ”double corner nodes” (`eCNX=2`) are occassionally useful to represent singularities in fracture mechanics.

§*3.2.2.7 Element Freedom Configuration*

The element freedom configuration defines the *default freedom assignation* at element nodes. Here “default” means that the assignation may be ovewriden on a node by node basis by the Freedom Definition data studied in Chapter 7. A brief introduction to the concept of freedom assignment is needed here. More details are given in that chapter. The discussion below is limited to Structural Mechanics elements.

Table 3.6 Freedom Signatures in element descriptor

Signature	Meaning
0	Freedom is not assigned ("off")
1	Freedom is assigned ("on")

At each node n a local Cartesian reference system $(\bar{x}_n, \bar{y}_n, \bar{z}_n)$ may be used to define freedom directions. See Figure 2.1. If these directions are not explicitly specified, they are assumed to coincide with the global system (x, y, z). (This is in fact the case in the present implementation).

In principle up to six degrees of freedom can be assigned at a node. These are identified by the symbols

$$\texttt{tx, ty, tz, rx, ry, rz} \tag{3.13}$$

Symbols `tx`, `ty` and `tz` denote translations along axes $\bar{x}_n$, $\bar{y}_n$ and $\bar{z}_n$, respectively, whereas `rx`, `ry` and `rz` denotes the rotations about those axes. If a particular freedom, say `rz`, appears in the finite element equations of the element it is said to be *assigned*. Otherwise it is *unassigned*. For example, consider a flat thin plate element located in the global (x, y) plane, with all local reference systems aligned with that system. Then `tz`, `rx`, and `ry` are assigned whereas `tx`, `ty` and `rz` are not.

The freedom configuration at corner, side, face and interior nodes of an element is defined by four integers denoted by

$$\texttt{eFS = \{ eFSC, eFSS, eFSF, eFSI \}} \tag{3.14}$$

which are called *element freedom signatures*. Omitted values are assumed zero.

Each of the integers in (3.14) is formed by decimally packing six *individual freedom signature* values:

$$\begin{aligned}
\texttt{eFSC} &= \texttt{100000*txC + 10000*tyC + 1000*tzC + 100*rxC + 10*ryC + rzC} \\
\texttt{eFSS} &= \texttt{100000*txS + 10000*tyS + 1000*tzS + 100*rxS + 10*ryC + rzS} \\
\texttt{eFSF} &= \texttt{100000*txF + 10000*tyF + 1000*tzF + 100*rxF + 10*ryC + rzF} \\
\texttt{eFSI} &= \texttt{100000*txI + 10000*tyI + 1000*tzI + 100*rxI + 10*ryC + rzI}
\end{aligned} \tag{3.15}$$

The freedom signature `txC` is associated with the translation `tx` at corner nodes. Similarly, `rxS` is associated with the rotation `rx` at side nodes. And so on. The signature value specifies whether the associated freedom is on or off, as indicated in Table 3.6. Signatures other than 0 or 1 are used in the Master Freedom Definition Table described in Chapter 7 to incorporate the attribute called *freedom activity*.

REMARK 3.3

If the element lacks node of a certain type the corresponding signature is zero. For example, if the element has only corner nodes, `eFSS = eFSF = eFSI = 0`, and these may be omitted from the list (3.14).

REMARK 3.4

For an MFC element the signatures, if given, are conventionally set to zero, since the node configuration is obtained from other information.

§3.3 ELEMENT NODE LIST

The list of node numbers of the element is supplied by the user. This information is grouped into four sublists, at least one of which is non empty:

```
nodelist = { cornernodes, sidenodes, facenodes, internalnodes }
```
(3.16)

The first list, `cornernodes`, lists the corner nodes of the element by external node number. If the element has only corner nodes all other lists are empty. If the element has side nodes, they are listed in `sidenodes`, and so on.

In programming the following variable names are often used for these lists:

```
eXNL = { eFXNL, eSXNL, eFXNL, eILNL }
```
(3.17)

in which the letter `X` emphasizes that these are *external* node numbers. If the `X` is dropped, as in `eNL`, it implies that *internal* node numbers appear.

Corner nodes must be ordered according to the conventions applicable to each geometric element shape. All corner node numbers must be nonzero and positive.

Negative or zero node numbers are acceptable for the last three lists. A zero number identifies a "hierarchically constrained" node or H-node used in certain transition elements; such freedoms do not appear in the assembled equations. A negative number identifies an unconnected node or U-node, whose degrees of freedom are not connected to those of another element. In linear static analysis those freedoms may be statically condensed at the element level. I-nodes are always U-nodes.

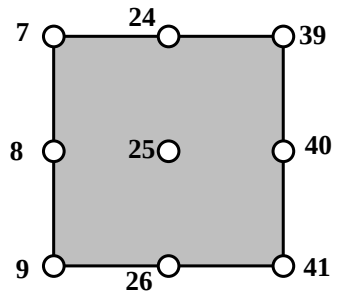


Figure 3.3. Example element to illustrate the configuration of the element node list `eXNL`.

EXAMPLE 3.1

The element nodelist of the 9-node quadrilateral element shown in Figure 3.3 following the usual counterclockwise-numbering conventions of most FEM codes, is

```
eXNL = { {7,9,41, 39}, {8,26,40,24 }, {25 }}
```
(3.18)

REMARK 3.5

There are 2D "flux elements" that contain no corner nodes. They are connected through midside nodes in 2D and face nodes in 3D. The specification of their node list is a bit unusual in that the first list is empty in 2D whereas the first two lists are empty in 3D. Such elements are not commonly used in Structural Mechanics, however.

§3.4 ELEMENT CODES

The element codes are integers that indirectly specify individual element properties by pointing to constitutive and fabrication tables. Up to three codes may be listed:

```
element-codes = { constitutive, fabrication, template }
```
(3.19)

The variable names often used for these items are

```
eCL = { cc, fc, tc }  or  { ccod, fcod, tcod }
```
(3.20)

Of these the first two codes must appear explicitly, although values of zero are acceptable to indicate a "null" or void pointer. The last one is optional.

§3.4.1 Constitutive Code

This is a pointer to the Constitutive tables that specify material properties. The format of these tables is presented in Chapter 4.

Some elements (for example, constraint elements) may not require constitutive properties, in which case the code is zero.

§3.4.2 Fabrication Code

This is a pointer to the Fabrication tables that specify fabrication properties. These properties include geometric information such as thicknesses or areas. For constraint elements fabrication tables are used to specify coefficients in multipoint or multifreedom constraints. The format of these tables is presented in Chapter 5.

Some elements (for example, solid elements) may not require fabrication properties, in which case the code is zero.

§3.4.3 Template Code

This is a pointer to the Template tables that specify numerical coefficients applicable to the construction of certain classes of elements that fall under the `TEM` Formulation. If the element does not pertain to that class, this code can be left out or set to zero.

EXAMPLE 3.2

If the constitutive code is 3 and the fabrication code is 4, the element code list is

```
eCL = { 3,4 }
```
(3.21)

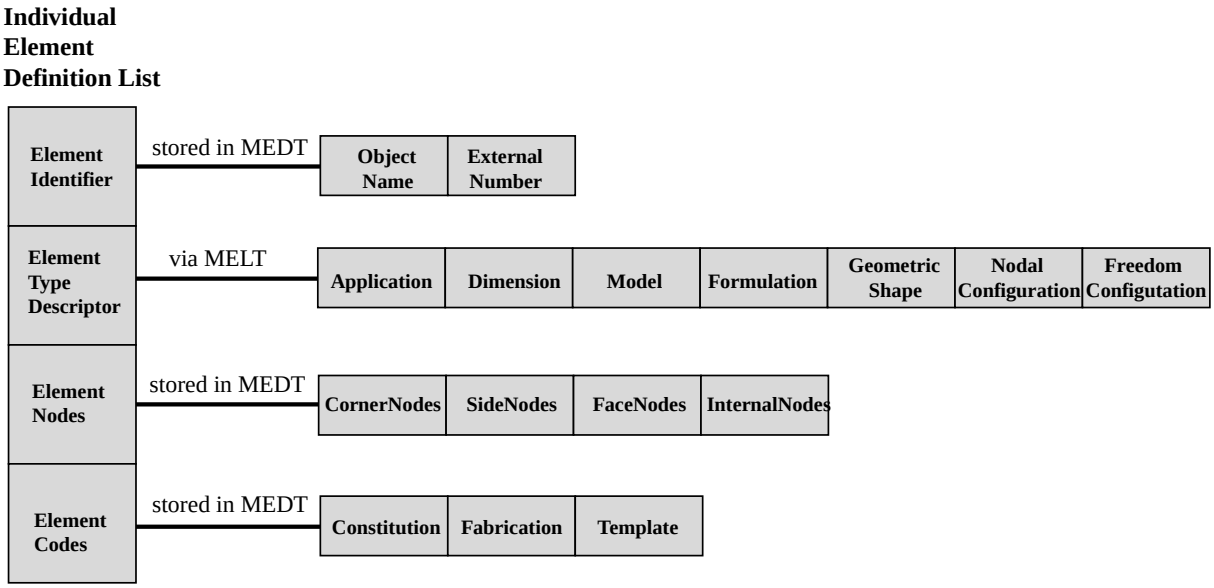


Figure 3.4. Configuration of the Individual Element Definition List.

§3.5 INDIVIDUAL ELEMENT DEFINITION LIST

The Individual Element Definition List or `eDL` defines the properties of an individual element This `eDL` is a list of four components that have been previously described in §3.1 through §3.4:

$$\texttt{eDL := \{identifier, type, nodes, codes \}} \tag{3.22}$$

The variable names often used for these items are

$$\texttt{eDL = \{eleident, eletype, elenodes, elecodes \}} \tag{3.23}$$

As described above, each of the eDT components is a list, or a list of lists. The complete eDL configuration is summarized in Figure 3.4.

The `eDL`s of all elements are stacked to built the Master Element Definition Table described in Chapter 4.

-

4

Element Grouping

Chapter 3 discussed the data that defines an individual element. This Chapter shows how that data is grouped to form a Master Element Definition Table.

Unlike nodes, the element data can be arranged in several ways to meet the needs of the user or program. Three important arrangements, qualified as Master, Visual and Local, are described here. Although these certainly do not exhaust all possibilities, they illustrate the three most important arrangements of element data.

Familiarity with the concepts and terminology of Chapter 3 is essential.

§4.1 THE MASTER ELEMENT DEFINITION TABLE

§4.1.1 General Description

The Master Element Definition Table or `MEDT` is a flat data structure that list all individual finite elements of a discrete model. The qualifier "flat" means that for example, the element with internal number 42 is the 42^{th} object in the table.

The qualifier "Master" means that the other element data structures described in this Chapter: the Visual Element Definition Table or `VEDT`, and the Local Subdomain to Element Connection Table or `LSECT` can be built from the `MEDT`. Consequently `VEDT` and `LSECT` are referred to as *subordinat* data structures.

On the other hand, the `MEDT` cannot be reconstructed from its subordinates. Hence the proper way to transfer the finite element model from a computer to another is to move the `MEDT`.

§4.1.2 Configuration

The `MEDT` is simply the collection of all Individual Element Definition Lists or `eDL` defined in §3.5:

```
MEDT = { eDL(1), eDL(2), ... eDL(e), ... eDL(numele)}          (4.1)
```

where `numele` is the total number of defined elements. List `eDL(e)` defines properties of the $\texttt{e}^{th}$ individual element, where `e` is the internal element number.

This configuration of the `MEDT` has advantages and shortcomings. If the internal element number is known, extraction of the data for that element is immediate. See Figure 4.1(a). Access by any other attribute is less efficient in that it will generally require a table search.

§4.2 THE VISUALIZATION ELEMENT DEFINITION DATA

The Visualization Element Definition Data or `VEDT` is organized according to object name. It thus provide rapid access to all elements that pertain to one substructure. It is also well suited to extract the `eDL` given the external element identifier; see Figure 4.1(b).

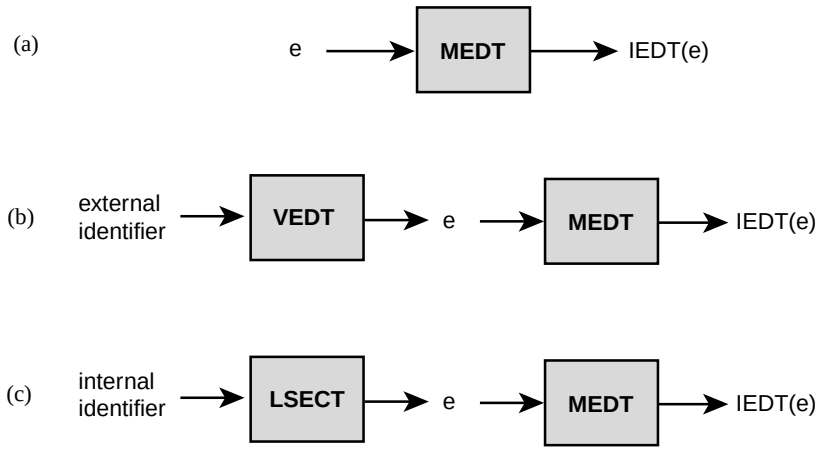


Figure 4.1. Element access operations using `MEDT`, `VEDT`, and `LSECT`.

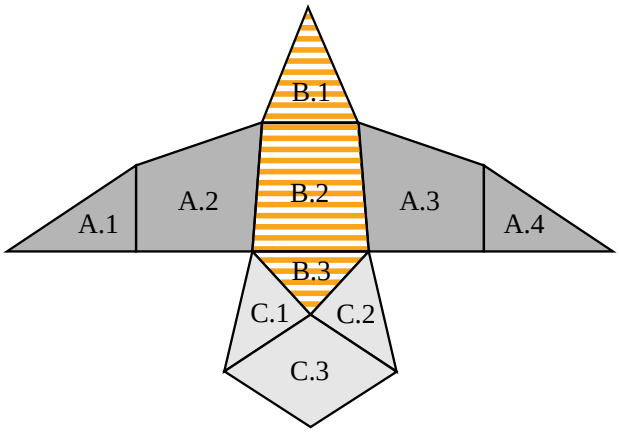


Figure 4.2. Mesh configuration for example (4.1).

§4.2.1 One Level Substructures

The organization of the `VEDT` for one-level substructures is best illustrated by an example. Suppose that ten elements pertaining to substructures `A`, `B` and `C` of the 2D mesh of Figure 4.2 have been defined in the following sequence:

A.1	B.1	A.4	C.3	B.2	A.3	C.2	A.2	C.1	B.3
1	2	3	4	5	6	7	8	9	10

(4.2)

where the number in the second row is the internal element number. For this case the VEDT may be organized as a two-level data structure that lists substructure names followed by internal element

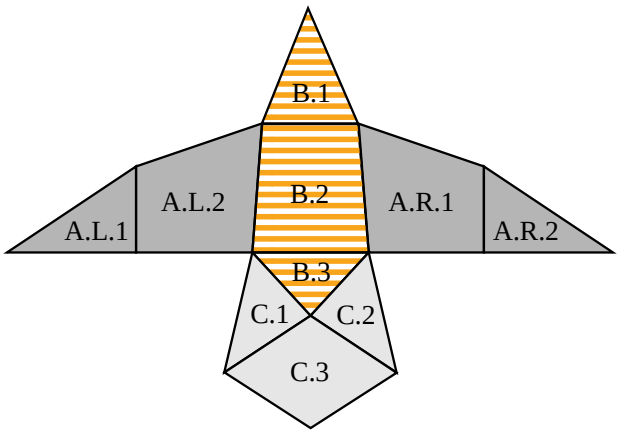


Figure 4.3. Mesh configuration for example (4.3).

numbers:

```
VEDT= {{A,{1,3,6,8}}, {B,{2,5,10}}, {C,{4,7,9}}}
```
(4.3)

Note that substructure names have been alphabetically ordered. This permits fast access to a particular name by binary search of the VEDT. This version of the VEDT is called *object sorted* or *name sorted*.

Internal element numbers are also ordered in sequence but that does not guarantee like ordering of user numbers. For example, C.1 and C.2 have internal numbers 9 and 7, respectively. Thus the arrangement illustrated by (4.3) is suitable when access to all elements of a substructure, in no particular sequence, is desirable. This is common in graphic operations.

If fast access to a named individual elements, say C.3, is important, the internal element number lists should be reordered in the sense of increasing external element numbers. This alternative VEDT form, called *fully sorted*, is

```
VEDT = {{A,{1,8,6,3}}, {B,{2,5,10}}, {C,{9,7,4}}}
```
(4.4)

Now binary search can be used for element numbers too.

§4.2.2 Multiple Level Substructures

For multiple level substructuring, the VEDT acquires additional list levels. Figure 4.3 illustrates a simple modification of the previous example, in which A contains sub-substructures L and R ("left" and "right"), whereas B and C remain the same, conveys the idea:

A.L.1	B.1	A.R.2	C.3	B.2	A.L.2	C.2	A.R.1	C.1	B.3
1	2	3	4	5	6	7	8	9	10

(4.5)

The object-sorted VEDT is

```
VEDT = {{A,{L,{1,6}},{R,{3,8}}}, {B,{2,5,10}}, {C,{4,7,9}}}
```
(4.6)

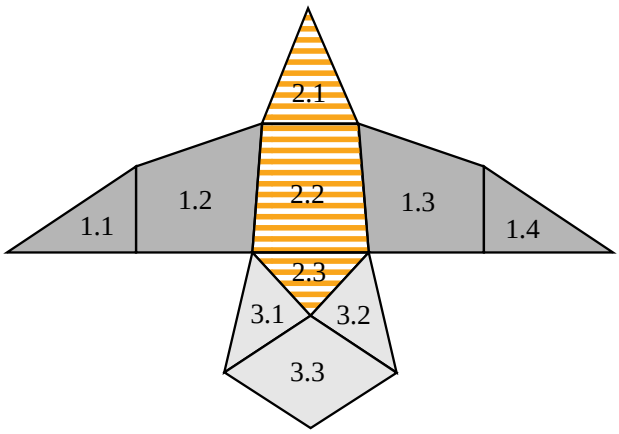


Figure 4.4. Mesh configuration for example (4.5).

whereas the fully-sorted form is

```
VEDT = {{A,{L,{1,6}},{R,{8,3}}}, {B,{2,5,10}}, {C,{9,7,4}}}        (4.7)
```

Many variations of these basic arrangements that sort elements by type or other attributes can be imagined. Since the number of such possibilities is very great, and many arrangements are likely to be customized to make optimal use of underlying hardware, there is little point in describing more variants here.

§4.3 *LOCAL SUBDOMAIN TO ELEMENT CONNECTION TABLE

The Local Subdomain to Element Connection Table or `LSECT`, is a data structure that defines the partition of the finite element model into *subdomains* to facilitate parallel processing. The `LSECT` is the primary data structure produced by a Domain Decomposer. It is well suited to extract the `eDL` given the subdomain identifier; see Figure 4.1.

EXAMPLE 4.1

Let us reuse the example (4.2) with one change. Suppose that substructures `A`, `B` and `C` become *subdomains* `1`, `2` and `3`, respectively, as illustrated in Figure 4.4:

1.1	2.1	1.4	1.3	2.2	1.3	3.2	1.2	3.1	2.3
1	2	3	4	5	6	7	8	9	10

(4.8)

where the notation `s.se` identifies the $\mathtt{se}^{th}$ element of the $\mathtt{s}^{th}$ subdomain. The configuration of the `LSECT` is

```
LSECT = {{1,3,6,8}, {2,5,10}, {4,7,9}}        (4.9)
```

Conparing this to the VEDT (4.5) it is can observed that (4.9) is simpler as only a two-level list is necessary. This is because subdomains are identified by number and not by name. Thus to access all elements that belong to, say subdomain 2, one simply extracts the second sublist: `{2,5,10}`, from (4.9). From the internal element numbers then one can immediately get the `eDL`, as schematized in Figure 4.1(c). The external element identifier is not involved in these manipulations.

Cell 4.1 Definition of Individual Element

```
DefineIndividualElement[MEDT_,eDL_]:= Module [
  {edt=MEDT,name,numele,m},
   name=eDL[[1]]; numele=Length[edt]; m=0;
   Do [If [name==edt[[e,1]], m=e; Break[]], {e,1,numele}];
   If [m==0, AppendTo[edt,eDL], edt[[m]]=eDL];
   Return[edt];
 ];

 eDL1={"RightWing.Flap.67", {"QLAM.4"},{{2,6,5,4},{1,3,5,8}},{3,5}};
 eDL2={"RightWing.Spar.211",{"SPAR.2"},{{65,79}}, {4,12}};
 eDL3={"RightWing.Spar.211",{"SPAR.2"},{{65,101}},{4,12}};
 MEDT={};
 MEDT=DefineIndividualElement[MEDT,eDL1];
 MEDT=DefineIndividualElement[MEDT,eDL2];
 PrintMasterElementDefinitionTable[MEDT];
 MEDT=DefineIndividualElement[MEDT,eDL3];
 PrintMasterElementDefinitionTable[MEDT];
```

Cell 4.2 Output from the Program of Cell 4.1

```
ElemId              Type    Nodelist                      Codes
RightWing.Flap.67   QLAM.4  {{2, 6, 5, 4}, {1, 3, 5, 8}}  {3, 5}
RightWing.Spar.211  SPAR.2  {{65, 79}}                    {4, 12}
ElemId              Type    Nodelist                      Codes
RightWing.Flap.67   QLAM.4  {{2, 6, 5, 4}, {1, 3, 5, 8}}  {3, 5}
RightWing.Spar.211  SPAR.2  {{65, 101}}                   {4, 12}
```

REMARK 4.1

The data structures presented in §4.1 and §4.2 are called Master and Visualization Element Definition Tables, respectively. To maintain consistency, the `LSECT` should have been christened Local Element Definition Table. However, this particular data structure turns out to more properly belong to the class of *connectivity data structures* discussed in Chapter 9. This fact motivates its more convoluted name.

§4.4 THE MASTER ELEMENT LIBRARY TABLE

The Master Element Library Table or `MELT` was described in Chapter 3as an *internal* data structure that links element type names with element type descriptors. The purpose of the `MELT` is to buffer the user from element definition details. For example, to define a 2-node bar element the user simply gives the type `"BAR2"` without bothering about the fine print contained in the descriptor.

"Internal" means that `MELT` is not defined by the user but by the code developer. The contents of `MELT` provide an inventory of what elements are available to the user. There is one entry in the

Cell 4.3 External to Internal Node Mapping

```
PrintIndividualElementDefinitionList[eDL_]:= Module[
  {t},
   t=Table["          ",{4}];
   t[[1]]=eDL[[1]]; t[[2]]=eDL[[2,1]];
   t[[3]]=ToString[eDL[[3]]]; t[[4]]=ToString[eDL[[4]]];
   Print[TableForm[t,TableAlignments->{Left,Right},
           TableDirections->Row,TableSpacing->{2}]];
];

eDL={"RightWing.Flap.67",{"QLAM.4"},{{2,6,5,4},{}},{3,5}};
PrintIndividualElementDefinitionList[eDL];
```

Cell 4.4 Output from the Program of Cell 4.3

```
RightWing.Flap.67  QLAM.4  {{2, 6, 5, 4}, {}}  {3, 5}
```

table for each element in the program. The table is not modified unless new elements are added or existing elements removed.

Section 4.5 below provides a specific example of the configuration of this table.

§4.5 IMPLEMENTATION OF MEDT OPERATIONS

This section lists modules pertaining to the manipulation and display of the Master Element Definition Table (MEDT). They are listed in alphabetic order.

§4.5.1 Defining an Individual Element

Module `DefineIndividualElement` is displayed in Cell 4.1. It defines a new element by inserting its `nDL` in the `MEDT`. If the external element name is already in that table, the information is replaced by the new one; else it is appended. The input arguments are the existing `MEDT` and the `eDL`. The module returns the updated `MEDT`.

The statements following `DefineIndividualElement` in Cell 4.1 test the module by building a table with several, which is then printed with the module `PrintMasterElementDefinitionTable` described below. The output is shown in Cell 4.2.

§4.5.2 Printing Individual Node Definition List

Module `PrintIndividualNodeDefinitionList`, listed in Cell 4.3, print the `eDL` of an individual element.

The code is tested by the statements that follow the module, and the results of running the test program are shown in Cell 4.4.

§4.5.3 Printing the MEDT

Module `PrintMasterElementDefinitionTable`, listed in Cell 4.5, prints the `MEDT` in a tabular format. Its only input argument is the `MEDT`. The module is tested by the statements that follow it, which build an MEDT directly and print it.

Cell 4.5 Printing the Master Node Definition Table

```
PrintMasterElementDefinitionTable[MEDT_]:= Module[
  {e,numele,t},
   numele=Length[MEDT]; t=Table["  ",{numele+1},{4}];
   Do [
     t[[e+1,1]]=MEDT[[e,1]];
     t[[e+1,2]]=MEDT[[e,2,1]];
     t[[e+1,3]]=ToString[MEDT[[e,3]]];
     t[[e+1,4]]=ToString[MEDT[[e,4]]],
   {e,1,numele}];
   t[[1]] = {"ElemId","Type","Nodelist","Codes"};
   Print[TableForm[t,TableAlignments->{Left},
         TableDirections->{Column,Row},TableSpacing->{0,2}]];
];
MEDT={{"RightWing.Flap.67",{"QLAM.4"},{{2,6,5,4},{1,3,5,8}},{3,5}},
      {"RightWing.Spar.211",{"SPAR.2"},{{65,79}},{4,12}}};
PrintMasterElementDefinitionTable[MEDT];
```

Cell 4.6 Output from the Program of Cell 4.5

```
ElemId              Type    Nodelist                       Codes
RightWing.Flap.67   QLAM.4  {{2, 6, 5, 4}, {1, 3, 5, 8}}  {3, 5}
RightWing.Spar.211  SPAR.2  {{65, 79}}                    {4, 12}
```

§4.5.4 Printing the Type Indexed MEDT

Module `PrintIndexedMasterElementDefinitionTable` is listed in Cell 4.7. This module receives as argument the MEDT.

It prints the table like the module `PrintMasterElementDefinitionTable` with one difference: the type index is printed instead of the type name. This is useful for carrying out tests after the `MELT` has been modified, but is of little conecrn to the user.

The module is tested by the statements shown in Cell 4.6, which built a `MEDT`, store type indices, print the `MEDT` before and after execution. The outputs are shown in Cell 4.8.

Cell 4.7 Printing a Type Indexed MEDT

```
PrintTypeIndexedMasterElementDefinitionTable[MEDT_]:= Module[
  {e,numele,t,type},
   numele=Length[MEDT]; t=Table["  ",{numele+1},{4}];
   Do [
     t[[e+1,1]]=MEDT[[e,1]]; type=MEDT[[e,2]];
     If [Length[type]>1, t[[e+1,2]]=ToString[type[[2]]]];
     t[[e+1,3]]=ToString[MEDT[[e,3]]];
     t[[e+1,4]]=ToString[MEDT[[e,4]]],
   {e,1,numele}];
   t[[1]] = {"ElemId","Tix","Nodelist","Codes"};
   Print[TableForm[t,TableAlignments->{Left},
         TableDirections->{Column,Row},TableSpacing->{0,2}]];
];

MEDT={{"RightWing.Flap.67",{"QLAM.4",91},{{2,6,5,4},{1,3,5,8}},{3,5}},
      {"RightWing.Spar.211",{"SPAR.2"},{{65,79}},{4,12}}};
PrintTypeIndexedMasterElementDefinitionTable[MEDT];
```

Cell 4.8 Output from the Program of Cell 4.7

```
ElemId              Tix  Nodelist                      Codes
RightWing.Flap.67   91   {{2, 6, 5, 4}, {1, 3, 5, 8}}  {3, 5}
RightWing.Spar.211       {{65, 79}}                    {4, 12}
```

§4.5.5 Setting the Type Index in the MEDT

Module `SetTypeIndexInMasterElementDefinitionTable` is listed in Cell 4.9. This module receives as input arguments the `MEDT` and the `MELT`. For each element found in the `MEDT` it scans the `MELT` to verify that a linkage exists and if so it stores the type index. If no linkage is found the index field remains empty.

The module is tested by the statements shown in Cell 4.9, which directly built a `MEDT` and `MELT`, store type indices, print the `MEDT` before and after execution. The outputs are shown in Cell 4.10.

Cell 4.9 Setting Up Type Indices in MEDT

```
SetTypeIndexInMasterElementDefinitionTable[MEDT_,MELT_]:= Module[
  {edt=MEDT,e,i,itype,jtype,numele,type,typnam},
   numele=Length[edt]; numtyp=Length[MELT]; itype=jtype=1;
   Do [type=MEDT[[e,2]]; typnam=type[[1]];
      If [typnam!=MELT[[jtype,1]], itype=0;
         Do [If[typnam==MELT[[i,1]], itype=i; Break[]], {i,1,numtyp}]
         ];
      jtype=itype; edt[[e,2]]={typnam,jtype}; jtype=Max[jtype,1],
   {e,1,numele}];
   Return[edt];
];

MELT={{"BAR2D.2","STM",2,"BAR",   "MOM","SEG",1000,{110000}},
      {"BAR3D.2","STM",3,"BAR",   "MOM","SEG",1000,{111000}},
      {"QLAM.4", "STM",2,"LAMINA","ISO","QU1",1000,{110000}},
      {"SPAR.2", "STM",2,"SPAR",  "MOM","SEG",1000,{111000}},
      {"MFC",    "STM",2,"CON",   "EXT","ARB",1000,{0,0,0,0}}};
 eDL1={"RightWing.Flap.67",{"QLAM.4"},{{2,6,5,4},{1,3,5,8}},{3,5}};
 eDL2={"RightWing.Spar.211",{"SPAR.2"},{{65,79}}, {4,12}};
 eDL3={"RightWing.Spar.214",{"BAR2D.2"},{{65,101}},{4,12}};
 eDL4={"RightWing.MFC.20",{"MFC"},{{2,65}},{0,1}};
 MEDT={};
 MEDT=DefineIndividualElement[MEDT,eDL1];
 MEDT=DefineIndividualElement[MEDT,eDL2];
 MEDT=DefineIndividualElement[MEDT,eDL3];
 MEDT=DefineIndividualElement[MEDT,eDL4];
 PrintMasterElementDefinitionTable[MEDT];
 MEDT=SetTypeIndexInMasterElementDefinitionTable[MEDT,MELT];
 PrintTypeIndexedMasterElementDefinitionTable[MEDT];
```

Cell 4.10 Output from the Program of Cell 4.9

```
ElemId              Type     Nodelist                      Codes
RightWing.Flap.67   QLAM.4   {{2, 6, 5, 4}, {1, 3, 5, 8}}  {3, 5}
RightWing.Spar.211  SPAR.2   {{65, 79}}                    {4, 12}
RightWing.Spar.214  BAR2D.2  {{65, 101}}                   {4, 12}
RightWing.MFC.20    MFC      {{2, 65}}                     {0, 1}
ElemId              Tix  Nodelist                      Codes
RightWing.Flap.67   3    {{2, 6, 5, 4}, {1, 3, 5, 8}}  {3, 5}
RightWing.Spar.211  4    {{65, 79}}                    {4, 12}
RightWing.Spar.214  1    {{65, 101}}                   {4, 12}
RightWing.MFC.20    5    {{2, 65}}                     {0, 1}
```

Cell 4.11 Constructing the Master Element Library Table

```
MakeMasterElementLibraryTable[]:= Module [ {MELT},
   MELT ={
          {"Bar2D.2","STM",2,"BAR", "MOM","SEG",1000,{110000}},
          {"MFC",    "STM",2,"CON", "EXT","ARB",1000,{0,0,0,0}}
         };
   Return[MELT];
];

PrintMasterElementLibraryTable[MakeMasterElementLibraryTable[]];
```

Cell 4.12 Output from the Program of Cell 4.11

```
TypeId   App   Dim   Model  Form   Sha  Nod-con  DOF-con
Bar2D.2  STM   2     BAR    MOM    SEG  1000     {110000}
MFC      STM   2     CON    EXT    ARB  1000     {0, 0, 0, 0}
```

§4.6 IMPLEMENTATION OF MELT OPERATIONS

This section presents an implementation of the Master Element Library Table or tt MELT. The material belongs more properly with that in the Chapter dealing with element implementations. The make and print operations are presented here, however, because they are also interweaved with the element definition.

§4.6.1 Constructing the MELT

Module `MakeElementLibraryTable`, listed in Cell 4.11, constructs the `MELT`. This module is changed only by the program developer and must be called at program start to initialize that table.

Testing statements follow to print the contents using the module described below. The output is shown in Cell 4.12.

§4.6.2 Printing the MELT

Cell 4.13 lists module `PrintMasterElementLibraryTable`, which lists a complete `MELT` supplied as argument.

The output from the test statements is shown in Cell 4.14.

Cell 4.13 Printing the MELT

```
PrintMasterElementLibraryTable[MELT_]:= Module[
  {numtyp=Length[MELT],t,n},
   t=Table["    ",{numtyp+1},{8}];
   Do [
       t[[n+1,1]]=MELT[[n,1]];
       t[[n+1,2]]=MELT[[n,2]];
       t[[n+1,3]]=ToString[MELT[[n,3]] ];
       t[[n+1,4]]=MELT[[n,4]];
       t[[n+1,5]]=MELT[[n,5]];
       t[[n+1,6]]=MELT[[n,6]];
       t[[n+1,7]]=ToString[MELT[[n,7]] ];
       t[[n+1,8]]=ToString[MELT[[n,8]] ],
   {n,1,numtyp}];
   t[[1]] = {"TypeId","App ","Dim ","Model","Form ","Sha",
             "Nod-con","DOF-con"};
   Print[TableForm[t,TableAlignments->{Left,Left},
           TableDirections->{Column,Row},TableSpacing->{0,2}]];
   ];

MELT={{"BAR2D.2","STM",2,"BAR",  "MOM","SEG",1000,{110000}},
      {"BAR3D.2","STM",3,"BAR",  "MOM","SEG",1000,{111000}},
      {"QLAM.4","STM",2,"LAMINA","ISO","QU1",1000,{110000}}};
PrintMasterElementLibraryTable[MELT];
```

Cell 4.14 Output from the Program of Cell 4.13

```
TypeId   App   Dim   Model   Form   Sha   Nod-con   DOF-con
BAR2D.2  STM   2     BAR     MOM    SEG   1000      {110000}
BAR3D.2  STM   3     BAR     MOM    SEG   1000      {111000}
QLAM.4   STM   2     LAMINA  ISO    QU1   1000      {110000}
```

-

5

Element Constitutive Properties

This Chapter continues the study of element-level information by explaining how constitutive properties are assigned to individual elements.

§5.1 GENERAL DESCRIPTION

Constitutive properties define the element material behavior of an individual element as specified by its constitutive equations. These properties are used by element processing routines for the computation of stiffness matrices, mass matrices, internal loads and stresses.

There is a wider spectrum of complexity in constitutive property computations than of any aspect of finite element computations, This reflects the huge range of constitutive models proposed and validated over the past 130 years by engineers and material scientists.

In the sequel the term *homogeneous element* identifies one whose macroscopic constitutive law is the same throughout its domain. On the other hand, the term *heterogeneous element* identifies one that is fabricated from with different constitutive laws. Fiber-reinforced layered composites and sandwich plates are common sources of heterogeneous elements in aerospace engineering structures.

The term "heterogeneous", however, does not apply if an effective or integrated constitutive model is specified from the outset. For example, the linear stiffness constitutive behavior of a laminated plate or a reinforced concrete beam may be described by moment/curvature and axial-force/stretch relations obtained by pre-integration through the thickness or cross section area, respectively. Such elements are called homogeneous from the standpoint of stiffness and stress-resultant calculations.

§5.2 *EFFECT OF CONSTITUTIVE MODEL ON MODULARITY

This section discusses the gradual loss of program modularity in accessing material properties as the constitutive law increases in complexity. Because this phenomenon can be the cause of extreme grief as analysis capabilities are expanded, it must be well understood by FEM program architects.

Consider first a simple FEM program restricted to homogeneous elements and *isotropic linear elastic* materials. Elasticity is described by two coefficients: E and ν, thermoelasticity by one: α, and mass properties by the density ρ. These four numbers may be stored in the Master Constitutive Property Table (MCPT) described below and identified by a constitutive property code. The processing of a routine simply uses that code, extracted from the Individual Element Definition List or `eDL`, to access the material properties, as schematized in Figure 5.1.

As a second scenario, consider a homogeneous element fabricated of a aelotropic (non-isotropic) linear elastic material. Now the constitutive properties depend upon *orientation*. For example, the properties of an orthotropic material (9 elastic constants, 3 dilatation coefficients, and one density) are normally stored with reference to three preferred directions. The orientation of those so-called *material directions* with respect to an element or global system is a geometric property and not a constitutive one, because it depends on the way the user selects reference axes. It follows that the element routines must use both the constitutive and the fabrication property codes to access constitutive data. The data dependencies for this case are illustrated in Figure 5.2.

As a more complex third scenario, consider now a heterogeneous element, the components of which display material behavior modeled by linear aelotropic constitutive relations. Now the element processing routine needs to know the extent and location of the different material regions. For example, the fabrication of laminated composite plate described in a layer-by-layer fashion is geometrically specified by the layer thicknesses. Those

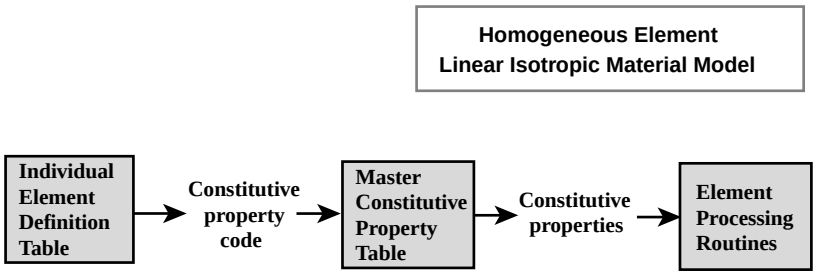


Figure 5.1. Access of constitutive element properties for a homogeneous element characterized by a linear elastic isotropic material model.

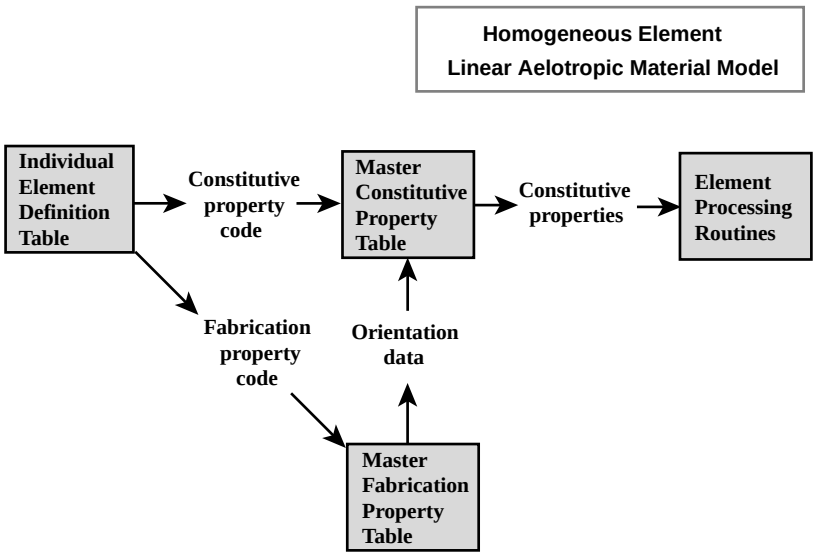


Figure 5.2. Access of constitutive element properties for a homogeneous element characterized by a linear elastic aelotropic (non-isotropic) material model.

extent properties are geometric in nature and must be procured from the fabrication property tables. The data dependencies for this case is illustrated in Figure 5.3. If all material regions exhibit isotropic behavior, the orientation data is omitted but the extent data is still be required.

Finally, if the material behavior becomes *nonlinear*, more difficulties arise. This is illustrated in Figures 5.4 through 5.6, which correspond to the three cases previously examined. [The qualifier "nonlinear isotropic" is somewhat of a misnomer because all nonlinear models introduce anisotropy; it means that such anisotropy is state induced.] The complexity is due to the effect of *state variables* such as temperatures, stresses, strains and strain rates, on the constitutive law. This brings up several undesirable complications:

1. Accessing and passing state variables into constitutive computations greatly raises the level of program complexity. Initialization and feedback effects appear, and detrimental effects on modularity increase the chances for errors.
2. Retrieval by simple table lookup is no longer feasible. Constitutive functions, which can become quite complex, are required.
3. For constitutive models that require access to prior history, as in nonlinear viscoelasticity, the state data can be become quite voluminous.

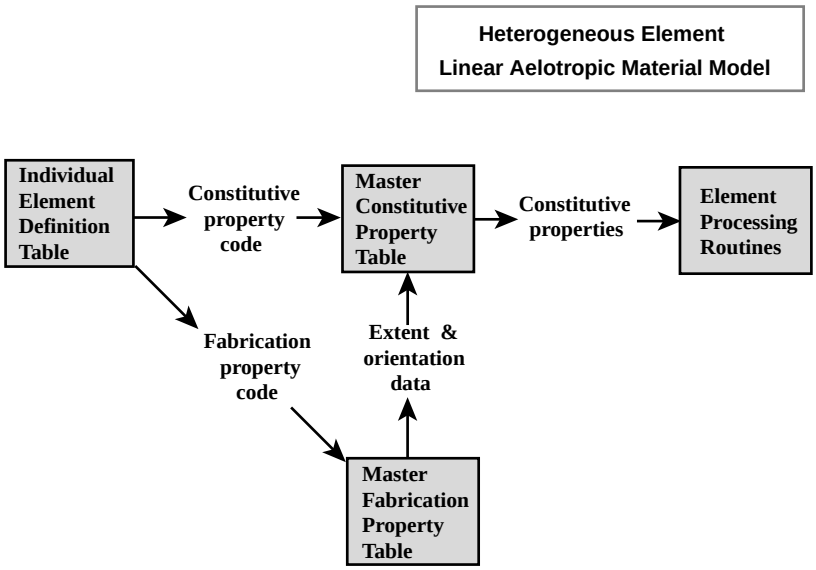


Figure 5.3. Access of constitutive element properties for a heterogeneous element characterized by linear elastic aelotropic material models.

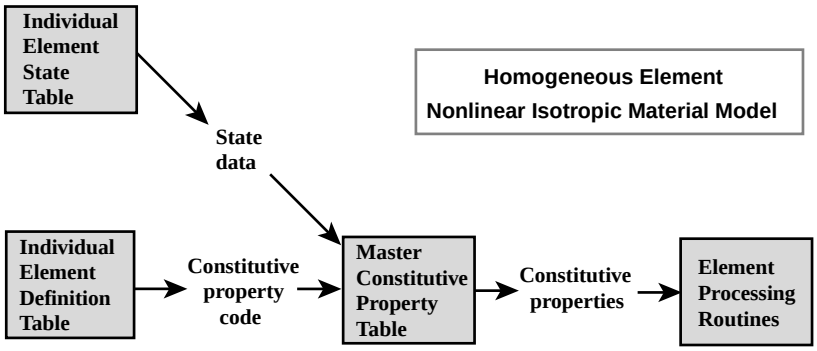


Figure 5.4. Access of constitutive element properties for a homogeneous element characterized by nonlinear isotropic material model.

There is a class of FEM programs, common in aerospace applications, that consider only linear material response but bring in the effect of temperature (and ocassionally other non-mechanical state variables, such as moisture in composites) on material properties. For these the linkage to state information takes on a different meaning, because the temperature distribution may come from the solution of an auxiliary problem, or simply from the users load input. The couplings displayed in Figures 5.3 and 5.4, however, are still typical for those programs, if "state data" is interpreted to cover thermomechanics.

The lesson that can be gathered from these case by case examination is that the implementation and processing of constitutive information has be carefully thought out even in linear programs. This is particularly true if there is a chance for future expansion to treat nonlinear or state-dependent material behavior.

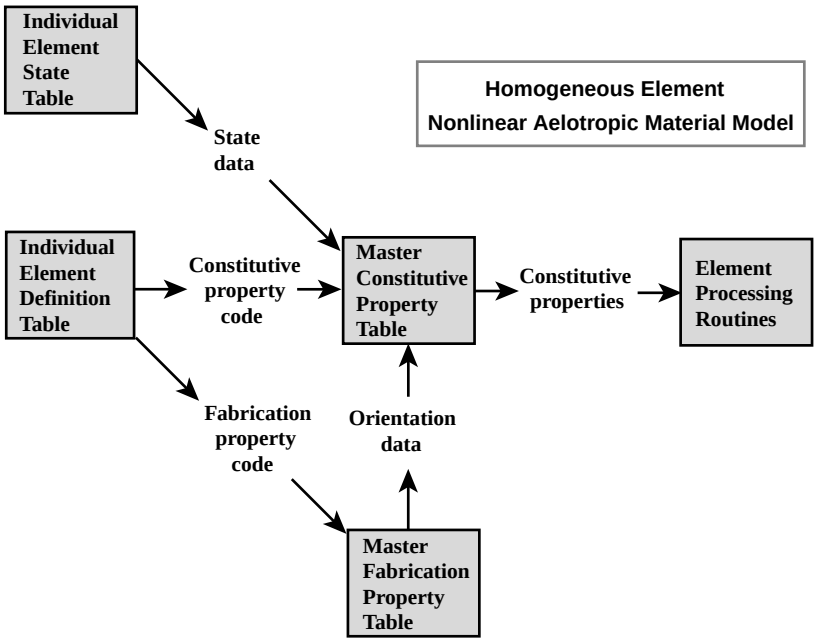


Figure 5.5. Access of constitutive element properties for a homogeneous element characterized by nonlinear aelotropic material model.

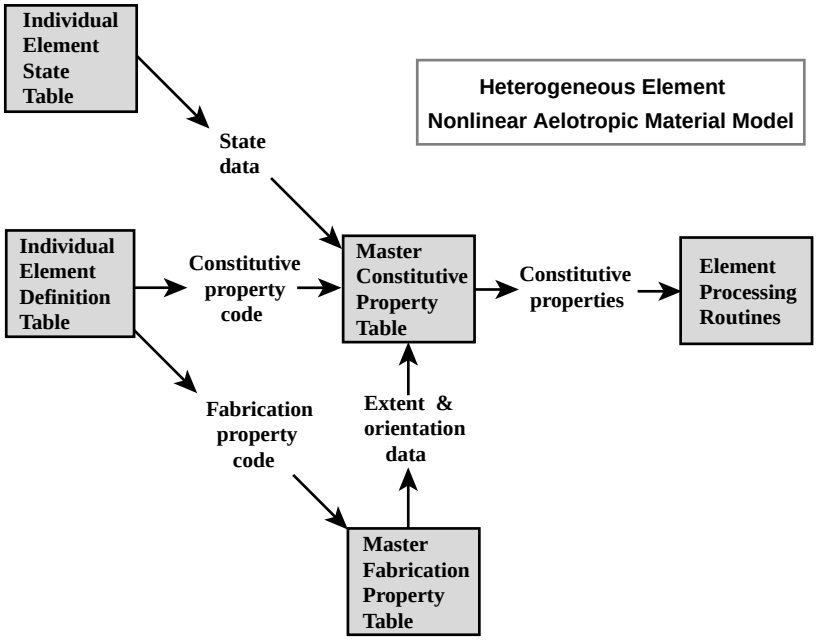


Figure 5.6. Access of constitutive element properties for a heterogeneous element characterized by nonlinear aelotropic material models.

§5.3 NAMES VERSUS CODES

A key implementation decision responds to the question: should materials be identify by name or numeric codes?

A *material name* is an identifier that refers to predefined data for specific and widely used materials. For example `"Al-7075"` for structural elements and `"Water"` for fluid elements. All necessary constitutive data and processing functions for the constitutive models of such materials are built in.

Material names are convenient for users. If used exclusively, however, the flexibility of the program is compromised. Another difficulty is that use of names may force the use of certain physical units that may not fit the problem or, worse, induce errors because of users misunderstanding. [This problem, however, may be circumvented by the "physical-unit keyword" device explained in the next section.] Furthermore, the possibility of doing what-if parameter studies in which material properties are systematically varied is precluded.

The use of numeric *constitutive codes* provides more flexibility but requires more input work from the user. For linear state-independent constitutive models the input work is light as data can be easily picked up from technical reference manuals. For nonlinear and state-dependent models the increased preparation burden may tilt the scales toward the use of material names.

It is recommended here that constitutive codes be always made available as the standard method, but the possibilty of use of built-in materials be kept in mind. The easiest way to prepare for that eventuality is to *reserve* a group of constitutive codes. The simplest choice is to reserve *negative code numbers* for built-ins, e.g. the name `"Al-7075"` may be associated with code -15, and to branch accordingly in the table lookup process discussed below.

A code of zero is reserved for elements that do not need constitutive properties. This is the case for Multi Freedom Constraint (MFC) elements.

The constitutive property code is usually designated by `cpc` or `cc` in programming.

§5.4 THE MASTER CONSTITUTIVE PROPERTY TABLE

The Master Constitutive Property Table, or `MCPT`, defines the constitutive properties of all materials that appear in the finite element model.

§5.4.1 Configuration of the MCPT

Assuming for simplicity that no built-in materials are defined (a case considered in the following subsection), the simplest configuration of the `MCPT` is a flat table:

$$\texttt{MCPT := \{ cPL(1), cPL(2), ..., cPL(lascpc)\}} \tag{5.1}$$

where `cPL` are lists that define individual materials. The properties for material with code $\texttt{cpc} \geq 1$ are stored in `cPL(cpc)`. Variable `lascpc` designates the largest constitutive code defined. If code $1 \leq \texttt{cpc} < \texttt{lascpc}$ has not been defined by the user, an empty list is stored as a gap placeholder.

As explained above, the configuration of a `cPL` may range from a simple list of few numbers for state-independent linear models, up to highly elaborated multilevel tables defining dependence of

nonlinear constitutive models on state variables such as temperature, moisture, stress level and strain rate. One advantage of use of a list structure is that such complexity is hidden behind the scenes and (5.1) still applies.

The cPL configuration assumed in the following example is

```
cPL := { Model-name, Physical-units, Property-values }
```
(5.2)

The first two items are keywords identifying the constitutive model and the physical units, respectively. This is followed by a property value list whose configuration depends on the constitutive model.

EXAMPLE 5.1

To give an example of an actual MCPT, suppose that two state-independent linear elastic materials appear in the finite element model. Code 1 is isotropic with $E = 21 \times 10^6$, $\nu = 0.30$, $\alpha = 1.2 \times 10^{-6}$ and $\rho = 453.6$ in the system of physical units (MKSC) chosen by the user. Code 4 is orthotropic, with $E_1 = E_2 = 12.5 \times 10^6$, $E_3 = 4.3 \times 10^6$, $\nu_{12} = 0.15$, $\nu_{23} = \nu_{31} = 0.28$, $G_{12} = 5.2 \times 10^6$, $G_{13} = G_{23} = 1.8 \times 10^6$, $\alpha_1 = \alpha_2 = 0.83 \times 10^{-6}$, $\alpha_3 = 1.35 \times 10^{-6}$ and $\rho = 230.7$ in the same unit system. The configuration of a MCPT that obeys the style (5.1)-(5.2) is

```
MCPT = {{"Isot.L.E.C","MKSC",{21.E6,0.30,1.2E-6,453.6}}, { }, { },
        {"Orth.L.E.C","MKSC",{12.5E6,12.5E6,4.3E6, 0.15,0.28,0.28,
         5.2E6,1.8E6,1.8E6, 0.83E-6,0.83E-6,1.35E-6, 230.7}}}
```
(5.3)

in which the two material modes have been specified by

```
cPL(1) = {"Isot.L.E.C","MKSC",{21.E6,0.30,1.2E-6,453.6}}
cPL(4) = {"Orth.L.E.C","MKSC",{12.5E6,12.5E6,4.3E6, 0.15,0.28,0.28,
          5.2E6,1.8E6,1.8E6, 0.83E-6,0.83E-6,1.35E-6, 230.7}}
```
(5.4)

Some explanatory comments for (5.3)-(5.4) are in order.

Constitutive models are identified by the first keyword found in the cPL. For example "Isot.L.E.C" identifies an *isotropic linear-elastic continuum* material model.

This is followed by a keyword expressing the system of physical units in which the property values are recorded. This is important for reuse of material properties when the system of units chosen by the user changes, because the program should be able to effect the necessary transformations (or at least issue an error message). The organization also helps in setting up invariant built-in property tables for named materials, as discussed in the next subsection.

A list of numerical properties follows. In the example a flat positional structure of the property list is assumed. For example, for the "Isot.L.E.C" model the values are expected in the order

$$E, \nu, \alpha, \rho \tag{5.5}$$

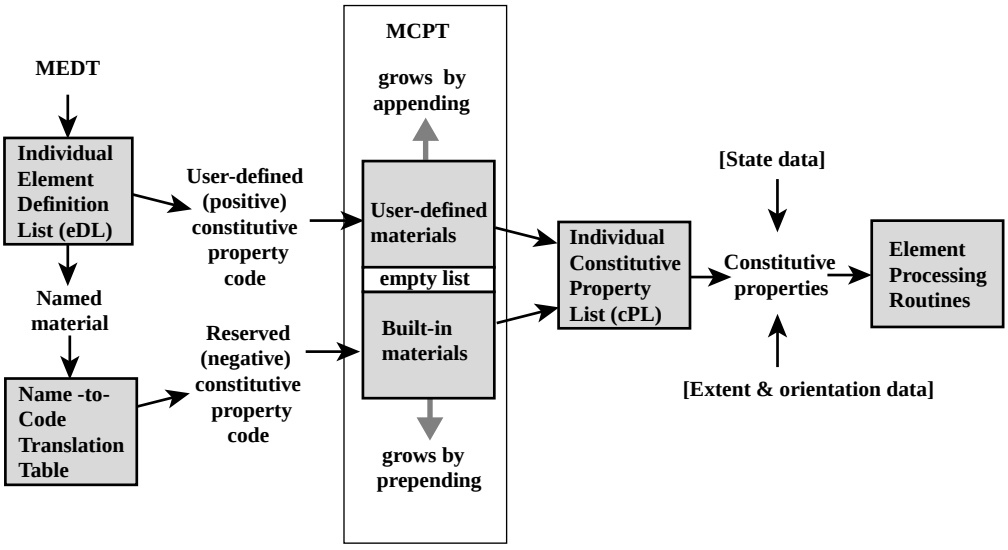


Figure 5.7. Accessing constitutive properties when named built-in materials are implemented. The source of extent, orientation and state data is omitted to avoid clutter.

whereas for the `"Orth.L.E.C"` model the values are expected in the order

$$E_1, E_2, E_3, \nu_{12}, \nu_{23}, \nu_{31}, G_{12}, G_{23}, G_{31}, \alpha_1, \alpha_2, \alpha_3, \rho. \tag{5.6}$$

More elaborated storage schemes using multilevel property sublists can of course be devised. Such schemes become essential when dependence on state variables like temperature is considered. Those more elaborate formats will not be studied here.

REMARK 5.1

The identification of property lists such as (5.5) and (5.6) as "continuum models" is useful to establish a distinction from section-integrated or thickness-integrated constitutive models, when such models are specified from the outset. For example, the constitutive behavior of beams, plates and shells is often described through moment/curvature and axial-force/stretch relations.

§5.4.2 Built-in Material Property Tables

As mentioned above, properties for commonly used structural materials may be predefined once and for all, permitting the use of mnemonic *material names* in lieu of numerical constitutive property codes. This reduces input preparation and the chance for errors.

It was noted that the implementation should always allow use of codes from the start. To permit the gradual implementation of predefined materials, it is appropriate to reserve a group of material codes for built-ins. Reserving negative integers is most convenient because it does not interfere with the natural use of positive integers for user-defined constitutive codes.

Cell 5.1 Defining an Individual Material

```
DefineIndividualMaterial[MCPT_,cPL_,ccod_]:= Module [
  {cc,CPT=MCPT,i,lenCPT,numres},
   If [Length[CPT]==0, CPT={{}}]; lenCPT=Length[CPT]; numres=0;
   Do [If [Length[CPT[[cc]]]==0, Break[]]; numres++, {cc,1,lenCPT}];
   If [ccod==0, CPT=Insert[CPT,cPL,numres+1]; Return[CPT]];
   Do [AppendTo[CPT,{}], {i,lenCPT+1,ccod+numres+1}];
       CPT[[ccod+numres+1]]=cPL; Return[CPT];
 ];

 cPL1={"General",{"Isot.L.E.C","symb"},{E,nu,alpha,rho}};
 cPL2={"Steel",  {"Isot.L.E.C","MKSF"},{21000,.3,.0014,6.65}};
 cPL3={"Water",  {"Isot.L.E.C","MKSF"},{0,.5,0.,1.45}};
 cPL4={"Phonium",{"Isot.L.E.C","MKSF"},{10000,.25,.000145,213.}};

 MCPT={};
 MCPT=DefineIndividualMaterial[MCPT,cPL1,8];
 MCPT=DefineIndividualMaterial[MCPT,cPL2,0];
 MCPT=DefineIndividualMaterial[MCPT,cPL3,0];
 MCPT=DefineIndividualMaterial[MCPT,cPL4,1];
 (* Print["MCPT=",MCPT]; *)
 PrintMasterConstitutivePropertyTable[MCPT];
```

Cell 5.2 Output from the Program of Cell 5.1

```
Ccod  Name     Model       Units  Properties
-2    Steel    Isot.L.E.C  MKSF   {21000, 0.3, 0.0014, 6.65}
-1    Water    Isot.L.E.C  MKSF   {0, 0.5, 0., 1.45}
1     Phonium  Isot.L.E.C  MKSF   {10000, 0.25, 0.000145, 213.}
8     General  Isot.L.E.C  symb   {E, nu, alpha, rho}
```

If negative codes are used for predefined material properties, it is recommended that these properties be stored in a reserved region of the MCDT rather than in a separate table. This permits the same modules to be used for both user-defined and built-in materials, simplifying program maintenance.

In the implementation described below, this is done by allowing the MCDT to *grow at both ends*, as illustrated in in Figure 5.7. That is, a new material is prepended if it is built-in, and appended if it is user-defined. The two regions are separated by an empty-list marker, which may be thought as belonging to code zero. This strategy makes the MCDT completely self-contained, which is a bonus when doing saves and restarts, or moving data between computers.

Cell 5.3 Retrieving the Properties of an Individual Material

```
GetIndividualMaterial[MCPT_,ccod_]:= Module [
  {cc,lenCPT=Length[MCPT],numres}, numres=0;
   Do [If [Length[MCPT[[cc]]]==0, Break[]]; numres++, {cc,1,lenCPT}];
   Return[MCPT[[ccod+numres+1]]];
 ];

MCPT={{"General",{"Isot.L.E.C","symb"},{E,nu,alpha,rho}},
      {"Steel",  {"Isot.L.E.C","MKSF"},{21000,.3,.0014,6.65}},
      {"Water",  {"Isot.L.E.C","MKSF"},{0,.5,0.,1.45}},
      {},
      {"Phonium",{"Isot.L.E.C","MKSF"},{10^6,.25,.000145,213.}}};
Print[GetIndividualMaterial[MCPT,-2]];
Print[GetIndividualMaterial[MCPT, 1]];
```

Cell 5.4 Output from the Program of Cell 5.3

```
{Steel, {Isot.L.E.C, MKSF}, {21000, 0.3, 0.0014, 6.65}}
{Phonium, {Isot.L.E.C, MKSF}, {1000000, 0.25, 0.000145, 213.}}
```

§5.5 IMPLEMENTATION OF CONSTITUTIVE OPERATIONS

This section lists modules pertaining to the creation, access and display of the Master Constitutive Property Table (MCFT). Modules are listed in alphabetic order.

§5.5.1 Defining an Individual Material

Module `DefineIndividualMaterial` is displayed in Cell 5.1. It defines a new material by inserting its material property list `cPL` in the `MCPT`.

The arguments are the incoming `MCPT`, the `cPL` and a constitutive code `ccod`. The latter may be positive or zero. If positive, e.g. 5, the material is to be accessed by that numeric code. If zero, the material is to be accessed by the name provided as first item in the `cPL`, for example `"Steel"`, and a negative constitutive code is internally assigned. This device assumes that negative code numbers are reserved for built-in materials, which defined by the FEM program and not the user.

If the constitutive code is already in use, the existing `cPL` is replaced by the new one. Else the incoming `cPL` is appended or prepended, depending on whether the material is user defined or built-in, respectively. The internal constitutive code of zero is conventionally represented by an empty `ePL`, which is used in the table access logic as "barrier" that separates negative from positive code numbers. See Figure 5.5. The module returns the updated `MCPT`.

The statements following `DefineIndividualMaterial` in Cell 5.1 test the module by

Cell 5.5 Printing Constitutive Properties of Individual Material

```
PrintIndividualConstitutivePropertyList[cPL_]:= Module[
  {t},
   t=Table[" ",{4}];
   t[[1]]=cPL[[1]];
   t[[2]]=cPL[[2,1]];
   t[[3]]=cPL[[2,2]];
   t[[4]]=ToString[cPL[[3]]];
   Print[TableForm[t,TableAlignments->{Left,Right},
           TableDirections->Row,TableSpacing->{2}]];
];

cPL={"name",{"Isot.L.E.C","symb"},{E,nu,alpha,rho}};
PrintIndividualConstitutivePropertyList[cPL];
```

Cell 5.6 Output from the Program of Cell 5.5

```
name  Isot.L.E.C  symb  {E, nu, alpha, rho}
```

building a MCPT with several materials. The table is then printed with the module PrintMasterConstitutivePropertyTable described below. The test output is shown in Cell 5.2.

§5.5.2 Getting Individual Material Properties

Module GetIndividualMaterial, listed in Cell 5.3, retrieves the cPL of an individual material given the code number. The arguments are the MCPT and ccod, and the function returns the appropriate cPL if the material exists.

Note that in the case of a built-in material, access is by its negative code number . This assumes that a mapping from the material name to its code has been effected.

The code is tested by the statements that follow the module, and the results of running the test program are shown in Cell 5.4.

In practice, GetIndividualMaterial is rarely used as a function, as the retrieval operation is usually done inline for efficiency. It is provided here to illustrate the table access logic, which is here complicated by the presence of positive and negative codes.

§5.5.3 Printing Individual Constitutive Properties

Module PrintIndividualConstitutivePropertyList is listed in Cell 5.5. This module print a Individual Constitutive Property List or CPL, which is supplied receives as argument.

Cell 5.7 Printing the Master Constitutive Property Table

```
PrintMasterConstitutivePropertyTable[MCPT_]:= Module[
  {cc,lenCPT,m,nummat,numres,reserve,t},
   lenCPT=Length[MCPT]; nummat=numres=0; reserve=True;
   Do [If [Length[MCPT[[cc]]]==0, reserve=False; Continue[]];
       nummat++; If [reserve,numres++],
   {cc,1,lenCPT}];
   t=Table["  ",{nummat+1},{5}]; m=0;
   Do [ If [Length[MCPT[[cc]]]==0, Continue[]]; m++;
      t[[m+1,1]]=ToString[cc-numres-1];
      t[[m+1,2]]=MCPT[[cc,1]];
      t[[m+1,3]]=MCPT[[cc,2,1]];
      t[[m+1,4]]=MCPT[[cc,2,2]];
      t[[m+1,5]]=ToString[MCPT[[cc,3]]],
   {cc,1,lenCPT}];
   t[[1]] = {"Ccod","Name","Model","Units","Properties"};
   Print[TableForm[t,TableAlignments->{Right,Left},
         TableDirections->{Column,Row},TableSpacing->{0,2}]];
];

MCPT={{"General",{"Isot.L.E.C","symb"},{E,nu,alpha,rho}},
      {"Steel",  {"Isot.L.E.C","MKSF"},{21000,.3,.0014,6.65}},
      {"Water",  {"Isot.L.E.C","MKSF"},{0,.5,0.,1.45}},
      {},
      {"Phonium",{"Isot.L.E.C","MKSF"},{10^6,.25,.000145,213.}}};
PrintMasterConstitutivePropertyTable[MCPT];
```

Cell 5.8 Output from the Program of Cell 5.7

```
Ccod  Name     Model       Units  Properties
-3    General  Isot.L.E.C  symb   {E, nu, alpha, rho}
-2    Steel    Isot.L.E.C  MKSF   {21000, 0.3, 0.0014, 6.65}
-1    Water    Isot.L.E.C  MKSF   {0, 0.5, 0., 1.45}
1     Phonium  Isot.L.E.C  MKSF   {1000000, 0.25, 0.000145, 213.}
```

The module is tested by the statements shown in Cell 5.5, which built a `MCPT`, store type indices, print the `MCPT` before and after execution. The outputs are shown in Cell 5.6.

§5.5.4 Printing the Complete MCPT

Module `PrintMasterConstitutivePropertyTable`, listed in Cell 5.7, prints the complete `MCPT` in a tabular format. Its only input argument is the `MCPT`.

The module is tested by the statements that follow it, which build a sample table directly and print it. The output from the test statements is shown in Cell 5.8.

-

6

Element Fabrication Properties

This Chapter concludes the discussion of element-level information by studying how fabrication properties are defined and assigned to individual elements.

§6.1 GENERAL DESCRIPTION

The term *fabrication properties* is used here with the following meaning: any property, usually of geometric nature or origin, that cannot be deduced, directly or indirectly, from the information in node definition tables and from other element tables. Because this definition operates by exclusion, it may confuse finite element users. Hence its meaning is best conveyed through some examples.

§6.1.1 Extent and Orientation Properties

Consider a solid element, such as a brick or tetrahedron, made up of a *homogeneous isotropic* linear material. All the so called *extent* properties of such element (shape and geometric dimensions) are defined by the node coordinates. And as illustrated in Figure 6.1, orientation data is not required for the specification of constitutive properties. We conclude that no fabrication properties are required for this element type. And indeed its fabrication property list is empty, or (alternatively) the element fabrication code is zero.

Now suppose the material of the solid element is orthotropic. The extent properties are still defined by the node coordinates. However, the direction of the preferred orthotropic material directions x_m, y_m, y_m with respect to the global system x, y, z must be known for the element processing routines to work, as sketched in Figure 6.2. This *orientation* data is not available from other sources. Hence it belongs in the fabrication property tables.

Next consider a 2-node linear bar (truss) element. The axial rigidity needed by the element stiffness computation routine is EA/L. The elastic modulus E can be retrieved from the constitutive tables, and the length L calculated from the node coordinates. But the cross sectional area A is not available elsewhere. Thus A clearly pertains to the fabrication property table, where it is stored as extent information.

§6.1.2 Fabrication of Heterogeneous Elements

Many elements require both extent and orientation properties. A classical example is a space beam. Cross section properties such as areas and moments of inertia clearly belong to the extent property list. But how about the orientation of the cross section axes? This information belongs to the orientation property list.

Plate and shell elements provide a great variety of possibilities. The simplest case would be a homogeneous plate or shell element with solid wall construction, and described by a linear isotropic material model. The only extent property is the thickness h, and no orientation properties are needed. If the plate or shell is still homogeneous but orthotropic, orientation data is required.

As a fairly complex example consider a laminated plate built up of 8 layers of orthotropic composite materials. A solid element model is illustrated in Figure 6.2. The 8 layer thicknesses must be stored as fabrication extent properties. And 8 sets of material directions data (which in most practical cases would consist of one rotation angle) must be stored as fabrication orientation data.

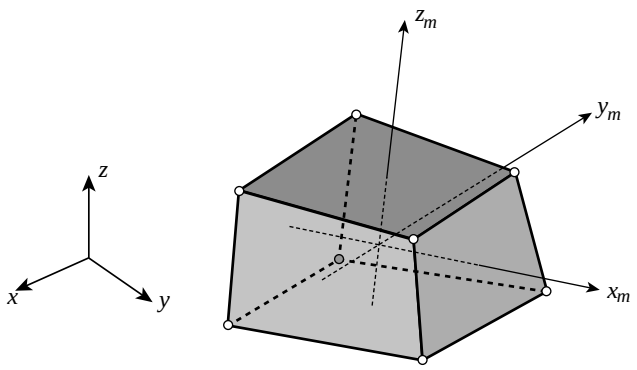


Figure 6.1. Principal material directions $\{x_m, y_m, z_m\}$ for a orthothropic solid element. The specification of the directions of $\{x_m, y_m, z_m\}$ with respect to $\{x, y, z\}$ must be part of the orientation fabrication properties of the element.

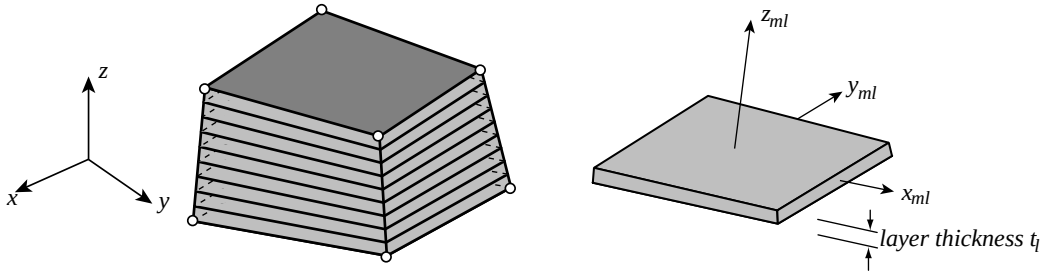


Figure 6.2. An example of heterogenous element used to model an 8-orthotropic-layer plate.

§6.2 THE INDIVIDUAL FABRICATION PROPERTY LIST

The foregoing discussion makes clear that element fabrication properties can be classified into two groups: *extent* and *orientation*. Extent properties define dimensional geometric entities such as lengths, areas and area-moments. The latter define element-related coordinate systems.

To cover all bases it is convenient to introduce a third category called *miscellaneous*, where one can deposit any piece of information that does not clearly fit into the preceding ones. The last class is in fact quite useful to take care of MultiFreedom Constraint (MFC) elements, which otherwise may appear as misfits.

We are now ready to introduce the Individual Fabrication Property List or `fPL`. There is one `fPL` for each defined geometric property code. The `fPL` configuration resembles that of its constitutive cousin `cPL` discussed in the previous Chapter:

```
fPL := { Fabrication-id, Physical-units, Property-values }
```
(6.1)

The first two items are keywords. Fabrication-id steers the processing and interpretation of the

property value list, and can provide valuable crosschecks against element definition data. For example, `"Isot.Hm.C1.Plate"` may be used to indicate that the property table is for an isotropic homogeneous C^1 plate element, whereas `"MFC"` identifies properties appropriate to a MultiFreedom Constraint element. If the user mistakingly has defined, say, a bar element with a property code that encounters one of these keywords, an error should be reported. The Physical-units keyword identifies the physical unit system in the same way explained for constitutive properties.

The Property-values list contains up to three sublists:

```
Property-values := { Extent, Orientation, Miscellanea }
```
(6.2)

where unused trailing sublists can be omitted. Empty lists may be required if leading information is missing. For example, properties for an MFC element are placed in the miscellaneous property list, in which case the extent and orientation lists must be present but empty.

EXAMPLE 6.1

An isotropic, homogeneous, C^1 plate has a thickness of 0.25 ft in a foot-pound-second-°F system of units:

```
fPL = {"Isot.Hm.C1.Plate","FPSF", {{0.25}}}
```
(6.3)

EXAMPLE 6.2

A MFC element links three degrees of freedom, u_{x4}, u_{x5} and θ_{z8} by the homogeneous constraint

$$0.2341\,u_{x4} - 1.608\,u_{x5} - \theta_{z8} = 0 \tag{6.4}$$

again in the customary English unit system. The coefficients of this relation are placed in the fPL of that element:

```
fPL = {"MFC.L.Hm","FPSF", {{},{},{0.2341,-1.608,-1.0}}}
```
(6.5)

Note that the extent and orientation lists must be explicitly specified as empty. The MFC keyword modifiers `"L.Hm"` say that the constraint is linear and homogeneous, and consequently the zero on the right-hand side of (6.4) need not (and must not) be stored. The identification of the freedoms that appear in (6.4) is obtained from other sources.

EXAMPLE 6.3

Consider the `fPL` for an *orthotropic* homogeneous solid element such as a brick or tetrahedron. Only the orientation of the 3 preferred material directions with respect to the global axes is needed. Suppose that such orientation is specified indirectly by three node numbers (see Chapter 1): 45, 68 and 21. Then

```
fPL = {"Solid.Hm.Orth"," ", {{},{45,68,21}}}
```
(6.6)

The physical-unit system keywork is blanked out here since the information is not unit dependent.

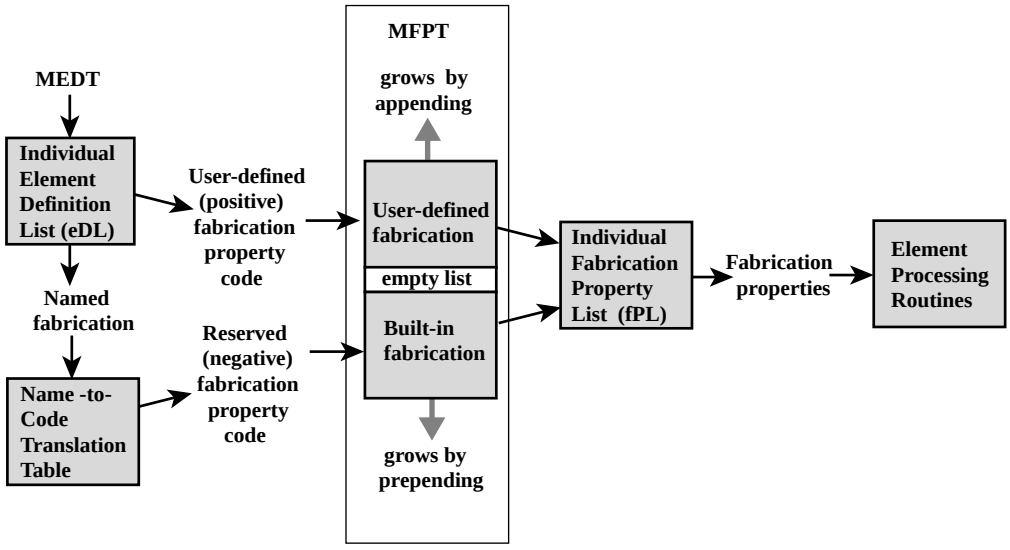


Figure 6.3. Accessing fabrication properties when named built-in fabrications are allowed. This implementation mimics that of the MCPT shown in Figure 5.7.

REMARK 6.1

The most general way to specify the fabrication of a one-dimensional structural member such as a beam or stiffener is by a *two-dimensional finite element mesh* that defines the cross section. This kind of "FE within FE" description is particularly useful for composite fabrication. Although it can be accomodated in the present framework, it introduces additional levels of indirection, which is triggered by appropriate keywords. Because this capability is of advanced nature, it will not be described here.

§6.3 THE FABRICATION PROPERTY CODE

Individual elements are linked to their fPLs through the *geometric property code* introduced in Chapter 3. This is a positive integer in the range 1 through lasfpc, where lasfpc denotes the largest defined fabrication code.

The alternative use of "geometric names" or "fabrication names" does not seem to be as attractive as in the case of constitutive properties. One reason is that property lists tend to be short and model-specific, whereas the properties of, for example, steel or water are universal and trascend specific applications.

Nevertheless, a case could be made for built-in tabulation of commercial cross sections of structural members such as beams, cables or panels, if the area of program application warrants such extra work. If this is done, negative geometric codes might be reserved to facilitate the implementation. An alternative but more elaborate procedure might be to branch on encountering a keyword within a fPL sublist.

The implementation reported below makes allowance for the case of built-in fabrications. These are assigned negative fabrication codes. The implementation to make up the MFCT is diagrammed

in Figure 6.4. This closely mimics that of the MCPT, as can be observed by comparing this with Figure 5.7.

§6.3.1 The Master Fabrication Property Table

The Master Fabrication Property Table or MFPT is constructed simply as a list of all fPLs, arranged by geometric property codes:

```
MFPT := {fPL(1),  fPL(2),  ....,  fPL(lasfpc)}          (6.7)
```

EXAMPLE 6.4

If the fPLs given in (6.1), (6.3) and (6.5) are defined with fabrication codes 1, 3 and 4, respectively, the master table becomes

```
MFPT = {{"Isot.Hm.C1.Plate","FPSF", {{0.25}}}, { },
        {"MFC.L.Hm","FPSF", {{ },{ },{0.2341,-1.608,-1.0}}}          (6.8)
        {"Solid.Hm.Orth"," ", {{ },{45,68,21}}}}
```

where it may be noticed that code 2 is empty.

§6.3.2 *Processing Modularity

Access of the MFPT by element processing routines is usually strightforward and modular, following the data flow schematized in Figure 6.1. The communication complexities that can affect constitutive processing, discussed in the previous Chapter, should not occur if proper design is followed. Two minor deviations from the ideal situation should be noted:

1. Constitutive processing may require orientation and/or extent data, as illustrated in Figures 5.2 through 5.6. For modularity reasons it is important to preclude constitutive processors from directly referencing fabrication property tables; such operation should be controlled by element processing routines. Thus the arrows in those figures should not be interpreted literally.
2. Computation of orientation properties may need to access the Master Node Definition Table if directions are specified indirectly through node locations.

§6.3.3 *A Flatter Implementation

An alternative implementation of the fabrication data set deserves study. It consist of separating the extent, orientation and miscellaneous properties into three separate tables, which are accessed through three property codes rather than one. The miscellaneous property table can then be merged with the more rarely used template table.

This organization “flattens” the configuration of the individual property tables at the cost of additional codes in the MEDT. The relative advantages of this implementation as opposed to the one shown above may depend on the complexity of the element library. If a program grows up to contain a very large number of elements and element versions, this kind of flattening may be advantageous as regards the simplification of the model processing logic.

Cell 6.1 Defining an Individual Fabrication

```
DefineIndividualFabrication[MFPT_,fPL_,fcod_]:= Module [
  {fc,ftp=MFPT,i,lenFPT,numres},
   If [Length[ftp]==0, ftp={{}}]; lenFPT=Length[ftp]; numres=0;
   Do [If [Length[ftp[[fc]]]==0, Break[]]; numres++, {fc,1,lenFPT}];
   If [fcod==0, ftp=Insert[ftp,fPL,numres+1]; Return[ftp]];
   Do [AppendTo[ftp,{}], {i,lenFPT+1,fcod+numres+1}];
       ftp[[fcod+numres+1]]=fPL; Return[ftp];
 ];

 fPL1={"GenThick",{"Thickness","MKFS"},{h}};
 fPL2={"GenArea",{"Area","MKSF"},{A}};
 fPL3={"GenLam",{"Thickness","MKSF"},{h1,h2,h3,h4}};
 fPL4={"GenOrient",{"Angle"," "},{},{phi}};
 fPL5={"GenMFC",{"MFC","MKSF"},{},{},{{n1,f1,c1}}};

 MFPT={};
 MFPT=DefineIndividualFabrication[MFPT,fPL1,3];
 MFPT=DefineIndividualFabrication[MFPT,fPL2,15];
 MFPT=DefineIndividualFabrication[MFPT,fPL3,0];
 MFPT=DefineIndividualFabrication[MFPT,fPL4,1];
 MFPT=DefineIndividualFabrication[MFPT,fPL5,6];
 PrintMasterFabricationPropertyTable[MFPT];
```

Cell 6.2 Output from the Program of Cell 6.1

```
fcod Name      Model     Units Extent           Orient MFC-coeff
-1   GenLam    Thickness MKSF  {h1, h2, h3, h4}
1    GenOrient Angle           {}               {phi}
3    GenThick  Thickness MKFS  {h}
6    GenMFC    MFC       MKSF  {}               {}     {{n1, f1, c1}}
15   GenArea   Area      MKSF  {A}
```

§6.4 IMPLEMENTATION OF FABRICATION OPERATIONS

This section lists modules pertaining to the creation, access and display of the Master Fabrication Property Table (MFPT). Modules are listed in alphabetic order.

§6.4.1 Defining an Individual Fabrication

Module `DefineIndividualFabrication` is displayed in Cell 6.1. It defines a new material by inserting its material property list `cPL` in the `MFPT`.

The arguments are the incoming `MFPT`, the `cPL` and a fabrication code `ccod`. The latter may be positive or zero. If positive, e.g. 5, the material is to be accessed by that numeric code. If zero, the

Cell 6.3 Retrieving the Properties of an Individual Fabrication

```
GetIndividualFabrication[MFPT_,fcod_]:= Module [
  {fc,lenFPT=Length[MFPT],numres}, numres=0;
   Do [If [Length[MFPT[[fc]]]==0, Break[]]; numres++, {fc,1,lenFPT}];
   Return[MFPT[[fcod+numres+1]]];
 ];

MFPT={{"GenLam", {"Thickness", "MKSF"}, {h1, h2, h3, h4}}, {},
   {"GenOrient", {"Angle", " "}, {}, {phi}}, {},
   {"GenThick", {"Thickness", "MKFS"}, {h}}, {}, {},
   {"GenMFC", {"MFC", "MKSF"}, {}, {}, {{n1, f1, c1}}}, {}, {}, {}, {},
   {}, {}, {}, {}, {"GenArea", {"Area", "MKSF"}, {A}}};
Print[GetIndividualFabrication[MFPT,6]];
Print[GetIndividualFabrication[MFPT,-1]];
```

Cell 6.4 Output from the Program of Cell 6.3

```
{GenMFC, {MFC, MKSF}, {}, {}, {{n1, f1, c1}}}
{GenLam, {Thickness, MKSF}, {h1, h2, h3, h4}}
```

material is to be accessed by the name provided as first item in the cPL, for example "Steel", and a negative fabrication code is internally assigned. This device assumes that negative code numbers are reserved for built-in materials, which defined by the FEM program and not the user.

If the fabrication code is already in use, the existing cPL is replaced by the new one. Else the incoming cPL is appended or prepended, depending on whether the material is user defined or built-in, respectively. The internal fabrication code of zero is conventionally represented by an empty ePL, which is used in the table access logic as "barrier" that separates negative from positive code numbers. The module returns the updated MFPT.

The statements following DefineIndividualFabrication in Cell 6.1 test the module by building a MFPT with several materials. The table is then printed with the module PrintMasterFabricationPropertyTable described below. The test output is shown in Cell 6.2.

§6.4.2 Getting Individual Fabrication Properties

Module GetIndividualFabrication, listed in Cell 6.3, retrieves the cPL of an individual material given the code number. The arguments are the MFPT and ccod, and the function returns the appropriate cPL if the material exists.

Note that in the case of a built-in material, access is by its negative code number . This assumes

Cell 6.5 Printing Properties of Individual Fabrication

```
PrintFabricationPropertyList[fPL_]:= Module[
  {t,p,lp,lfPL},
   t=Table[" ",{6}]; lfPL=Length[fPL];
   t[[1]]=fPL[[1]];
   t[[2]]=fPL[[2,1]];
   t[[3]]=fPL[[2,2]];
   If [lfPL>2, t[[4]]=ToString[fPL[[3]]]];
   If [lfPL>3, t[[5]]=ToString[fPL[[4]]]];
   If [lfPL>4, t[[6]]=ToString[fPL[[5]]]];
   Print[TableForm[t,TableAlignments->{Left,Right},
         TableDirections->Row,TableSpacing->{2}]];
];

fPL={"GenLam",{"Thickness","MKSF"},{h1,h2,h3,h4}};
PrintFabricationPropertyList[fPL];
```

Cell 6.6 Output from the Program of Cell 6.5

```
GenLam  Thickness  MKSF  {h1, h2, h3, h4}
```

that a mapping from the material name to its code has been effected.

The code is tested by the statements that follow the module, and the results of running the test program are shown in Cell 6.4.

In practice, `GetIndividualFabrication` is rarely used as a function, as the retrieval operation is usually done inline for efficiency. It is provided here to illustrate the table access logic, which is here complicated by the presence of positive and negative codes.

§6.4.3 Printing Individual Fabrication Properties

Module `PrintIndividualFabricationPropertyList` is listed in Cell 6.5. This module print a Individual Fabrication Property List or `fPL`, which is supplied receives as argument.

The module is tested by the statements shown in Cell 6.5, which built a `MFPT`, store type indices, print the `MFPT` before and after execution. The outputs are shown in Cell 6.6.

§6.4.4 Printing the Complete MFPT

Module `PrintMasterFabricationPropertyTable`, listed in Cell 6.7, prints the complete `MFPT` in a tabular format. Its only input argument is the `MFPT`.

Cell 6.7 Printing the Master Fabrication Property Table

```
PrintMasterFabricationPropertyTable[MFPT_]:= Module[
  {fc,lenFPT,m,numfab,numres,reserve,lfPL,t},
   lenFPT=Length[MFPT]; numfab=numres=0; reserve=True;
   Do [If [Length[MFPT[[fc]]]==0, reserve=False; Continue[]];
       numfab++; If [reserve,numres++],
   {fc,1,lenFPT}];
   t=Table["  ",{numfab+1},{7}]; m=0;
   Do [lfPL=Length[MFPT[[fc]]]; If [lfPL==0, Continue[]]; m++;
      t[[m+1,1]]=ToString[fc-numres-1];
      t[[m+1,2]]=MFPT[[fc,1]];
      t[[m+1,3]]=MFPT[[fc,2,1]];
      t[[m+1,4]]=MFPT[[fc,2,2]];
      If [lfPL>2, t[[m+1,5]]=ToString[MFPT[[fc,3]]]];
      If [lfPL>3, t[[m+1,6]]=ToString[MFPT[[fc,4]]]];
      If [lfPL>4, t[[m+1,7]]=ToString[MFPT[[fc,5]]]],
   {fc,1,lenFPT}];
   t[[1]] = {"fcod","Name","Model","Units","Extent","Orient","MFC-coeff"};
   Print[TableForm[t,TableAlignments->{Right,Left},
         TableDirections->{Column,Row},TableSpacing->{0,1}]];
];

fPL1={"GenThick",{"Thickness","MKFS"},{h}};
fPL2={"GenArea",{"Area","MKSF"},{A}};
fPL3={"GenLam",{"Thickness","MKSF"},{h1,h2,h3,h4}};
fPL4={"GenOrient",{"Angle"," "},{},{phi}};
fPL5={"GenMFC",{"MFC","MKSF"},{},{},{{18,4,-2.0}}};
MFPT={{},fPL1,fPL2,fPL3,fPL4,fPL5};
PrintMasterFabricationPropertyTable[MFPT];
```

Cell 6.8 Output from the Program of Cell 6.8

```
fcod Name      Model     Units Extent           Orient MFC-coeff
1    GenThick  Thickness MKFS  {h}
2    GenArea   Area      MKSF  {A}
3    GenLam    Thickness MKSF  {h1, h2, h3, h4}
4    GenOrient Angle           {}               {phi}
5    GenMFC    MFC       MKSF  {}               {}     {{18, 4, -2.}}
```

The module is tested by the statements that follow it, which build a sample table directly and print it. The output from the test statements is shown in Cell 6.8.

-

7

Freedom Assignment

This section describes how degrees of freedom are assigned to nodes, and their activity described.

§7.1 GENERAL DESCRIPTION

Degrees of freedom or simply *freedoms* are the primary variables that collectively define the state of a finite element model. They are sometimes abbreviated to DOF in the sequel. Associated with each freedom is a conjugate quantity generally known as *force*. A freedom and its associated force form a duality pair.

In finite element models freedoms are specific in nature. They are physical quantities assigned (or assignable) at *nodes*. For example the displacement of a corner point or the pressure at the center of an element. Consequently freedoms and nodes are interwined in the data structures described here.

From Chapter 2 it should be recalled that not all nodes possess freedoms. Those that do are known as *state nodes*. Nodes used for geometric purposes only are simply called *geometric nodes*. In the finite element models commonly used in practice, nodes do a double duty. In this Chapter the term *node* is used in the sense of state node

§7.2 FREEDOM ATTRIBUTES

Three attributes of freedoms must be considered when designing data structures: identification, configuration and activity. These are defined and discussed in the following subsections.

Master data structures for freedoms, and in particular the Master Node Freedom Table or `MNFT` defined later, are *node based*. This means that freedoms are defined by *node number* and *freedom index* rather than by a freedom number. This is in accordance with the way a FEM user is trained to think about freedoms. It is more natural to say "the y-displacement component at node 56 is zero" than "freedom 332 is zero."

Most data structures used in the assembly and solution of the FEM equations are freedom based. These are not *source* data structures, however; they are derived by post-processing the master freedom and element definition tables.

A consequence of the node-by-node organization is that configuration and activity attributes are actually intermixed and cannot be divorced from the standpoint of storage representation. For pedagogic purposes it is convenient to assume, however, that they are logically separated. This track is followed in the following sections.

§7.2.1 Freedom Identifiers

Freedoms in the data structures studied here are externally known by short mnemonic names. For example `tx` always denotes the translation about the nodal freedom axis $\bar{x}_n$ whereas `ry` always denotes the rotation about the nodal freedom axis $\bar{y}_n$. The term "nodal freedom axis" is explained later.

When it is useful to associate a freedom identifier with a specific or generic node whose internal number is `n`, two notations are available. The node number is enclosed in parenthesis, as in `tx(n)` or `ry(n)`. Or is separated by a dot, as in `tx.n` or `ry.n`. Each notation has its uses.

Table 7.1 Linkage Between Freedom Identifiers and Freedom Indices

Intended Application	Freedom Identifiers	Conjugate Identifiers	Freedom indices	Physical DOF Description
Structures	`tx`, `ty`, `tz`	`qx`, `qy`, `qz`	1,2,3	Translations about $\bar{x}_n$, $\bar{y}_n$, $\bar{z}_n$
Structures	`rx`, `ry`, `rz`	`mx`, `my`, `mz`	4,5,6	Rotations about $\bar{x}_n$, $\bar{y}_n$, $\bar{z}_n$
TBD	TBD	TBD	7–24	Reserved for future use in structural or multiphysics models

Note: $\{\bar{x}_n, \bar{y}_n, \bar{z}_n\}$ is the local Freedom Coordinate System (FCS) for node n; see §7.2.5. In the present implementation the FCS is assumed to coincide with $\{x, y, z\}$ at each node.

Two additional notational devices should be noted. Associated with, say, freedom `tx`, there is an integer called a *tag*, which is denoted by `txt`. There is also a *value*, which is denoted by `txv`. These may be associated with specific nodes by writing `txt(n)` or `txt.n` and `txv(n)` or `txv.n`.

§7.2.2 Freedom Configuration Indices

Freedoms assigned to nodes fill *freedom slots* marked by positive integers called the *freedom indices*. The *configuration* attribute specifies which freedom identifier corresponds to each index, and whether the freedom is assigned or not.

To make matters specific, the present study assumes that each state node may have up to a maximum of 24 freedom slots. Of these the first six are the standard displacement freedoms of structural mechanics: three translations and three rotations. The other 18 are reserved for future applications in either structural mechanics (for example, pressure freedoms in incompressible media) or coupled system analysis. See Table 7.1.

In the sequel only standard structural freedoms will be considered. Paired with each freedom are conjugate nodal load quantities identified in Table 7.2: `qx`, `qy` and `qz` are nodal forces conjugate to `tx`, `ty` and `tz`, while `mx`, `my` and `mz` are nodal moments conjugate to `rx`, `ry` and `rz`, respectively.

REMARK 7.1

The foregoing implementation names vary from standard matrix notation for displacement and force vectors, which are usually denoted by **u** and **f**, respectively. Use of `tx` and `rx` has the advantage of being more mnemonic in distinguishing between translational and rotational freedoms. Use of `fx`, `fy` and `fz` for forces is precluded by the letter "f" being reserved for freedom. The notational scheme, however, is also tentative and is subject to change.

§7.2.3 Freedom Configuration Tags

Although the connection between freedom index and identifier cannot be broken, it is not necessary to carry along all possible freedoms at each node. For example, in a structural problem involving only solid elements, only the three translational freedoms `tx`, `ty` and `tz` participate. This situation can be handled by saying that freedom indices 1, 2 and 3 are *assigned* or *present*. The three others:

Table 7.2 Freedom Configuration Tag Values

Tag	Signature	Assignment	Activity	Meaning
-1	0	Unassigned		Freedom is absent from FEM equations
0	1	Assigned	Active	Freedom value is unknown
1	2	Assigned	Passive	Freedom value is known
2–8	3–9	Assigned	Coactive	Freedom value appears in $n_c \geq 1$ MultiFreedom Constraints that involve other freedoms. Tag is set to $\min(1 + n_c, 8)$

4, 5 and 6, are *unassigned* or *absent*. Unsassigned freedoms are ignored in the formation of the FEM equations discussed in future Chapters.

The presence or absence of a freedom is indicated by an Freedom Configuration Tag or `fCT`. This is an integer which may take the values shown in Table 7.2. If the tag is nonnegative the freedom is present and absent otherwise. The Freedom Signature or `fS` is obtained by adding one to the `fCT`. The distinction between assigned tag values is explained in the following subsection.

Configuration tags for freedoms of a specific node are collected in a data structure called the *node Freedom Configuration Tag*, which is arranged as

```
nFCT = { txt, tyt, tzt, rxt, ryt, rzt }
```
(7.1)

The *node Freedom Signature Tags* are obtained by adding one to each of the above:

```
nFST = { txt+1, tyt+1, tzt+1, rxt+1, ryt+1, rzt+1 }
```
(7.2)

To conserve memory this is stored as a decimally packed integer called the Node Freedom Signature or `nFS`:

```
nFS = 100000*(txt+1) + 10000*(tyt+1) + 1000*(tzt+1)
    + 100*(rxt+1) + 10*(ryt+1) + (rzt+1)
```
(7.3)

This integer displays as a 6-digit number such as 110420, which is the packing of FCTs `{0,0,-1,3,1,-1}`. This is the reason for restricting tag values to the range -1 through 8.

§7.2.4 Freedom Activity

We have seen that the configuration tag — or alternatively the signature — indicates whether a freedom is assigned (present) or unassigned (absent). If a freedom is present, it has an *activity* attribute. This classifies each freedom into one of three possibilities: *active*, *passive* or *coactive*. The meaning of these attributes is given in Table 7.2.

REMARK 7.2

The first two possibilities are standard part of any finite element code. An active freedom is retained in the list of unknowns submitted to the equation solver.

Table 7.3 Specifications for Freedom Coordinate System

How	FCS-list	FCS construction
blank	ignored	Same as global directions
`"N"`	`n1,n2,n3`	Orient by 3 nodes as follows: x_n defined by `n1`→`n2`; y_n normal to x_n and in the plane defined by `n1`, `n2` and `n3` with positive projection on `n3`; and $z_n = x_n \times y_n$ forming a RH system. Here `n1,n2,n3` are *external* node numbers.
`"E"`	`e1,e2,e3,e4`	Orient by the 4 Euler parameters of $\{x_n, y_n, z_n\}$ wrt $\{x, y, z\}$
`"S"`	pST>0	pST points to an Side Table; used for Side Nodes in some bending elements
`"F"`	pfL>0	pFL points to a Face Table; used for Face Nodes in some solid and shell elements

The third possibility occurs when the freedom is part of a multifreedom constraint (MFC). For example, suppose that the z-translations of nodes 21 and 33 are linked by the algebraic constraint $2u_{z21} - 3u_{z33} = 1.75$, which translated into external freedom notation is

```
2 tz(21) - 3 tz(33) = 1.75
```
(7.4)

As discussed in Chapters 3 and 4, this information is represented as the definition of a *constraint element*. As far as freedom tables is concerned, the activity of `tz(21)` and `tz(33)` becomes an issue. They cannot be classified as passive because their value is not known *a priori*. They cannot be called active because the value of the conjugate forces is not known. They are closer, however, to the latter because the freedoms are retained in the discrete equations when a MFC is treated by penalty function of Lagrange multiplier methods. Hence the term *co-active*.

REMARK 7.3

Note that nothing prohibits a freedom from appearing in several Multifreedom Constraints, as long as these are linearly independent. If that number is 1 through 7, it can be directly obtained from the `fCT` by subtracting one; for example a tag of 4 means that the freedom appears in three MFCs. If the freedom appears in 8 or more MFCs, which is a highly unlikely occurrence, the tag value is 8.

§7.2.5 *Freedom Directions

Attached to each node `n` there is a Freedom Coordinate System (FCS), denoted by $\{\bar{x}_n, \bar{y}_n, \bar{z}_n\}$. This is a local Rectangular Cartesian system that specifies the directions along which the freedoms are defined. For many applications this is exactly the same, except for origin, as the global Cartesian reference system $\{x, y, z\}$ at each node. And indeed that is the default.

For some applications, however, it is convenient to have the ability to orient FCS at some or all nodes. Since those applications are the exception rather than the rule, this subsection has been marked as advanced and may be ignored on first reading.

The following situations favor the use of nodal FCS different from the global system:

1. Skew or "oblique" boundary conditions. These are common in systems with curved members and in shell structures. Only the FCS at boundary nodes need to be adjusted.

2. Problems that are more naturally described in cylindrical or spherical coordinates. Because the global system is always of Cartesian type, one simple way to accomodate those geometries is the ability to attach a local system at each node.
3. Some refined elements have freedom sets that are naturally expressed in local directions related to the element geometry. If those freedom sets are incomplete (for example, the rotation-about-side-direction in some plate bending and shell elements) it is necessary to assemble such freedoms in a local system defined by the geometry of node-attached elements.

There are several ways in which the FCS orientation can be defined. The most natural or convenient way is problem dependent and also may vary from node to node. To accomodate the variety a data structure called a Node Freedom Direction Selector or `nFDS` is available:

$$\texttt{nFDS = \{ How, FCS-list \}} \tag{7.5}$$

Here `How` is a letter that specifies how to construct the FCS, and `FCS-list` is a list of coefficients or a pointer to a Geometric Object Table. Some combinations are noted in Table 7.3. This information is placed in the Master Node Freedom Table described below.

§7.3 FREEDOM DATA STRUCTURES FOR NODES AND MFCS

§7.3.1 The Individual Node Freedom List

We now examine freedom data structures for individual nodes. There are two: the Individual Node Freedom List or `nFL`, and the Individual Node State List or `nSL`. The former is described below. The latter is discussed in the following Chapter, which deals with state data.

The Individual Node Freedom List, or `nFL`, records the freedom assignment, activity and direction at specific nodes of a FEM model. There is one `nFL` per node. This data structure has the following configuration:

$$\boxed{\texttt{nFL(n) = \{ xn, nFS(n), ndof(n), nFDS(n) \}}} \tag{7.6}$$

Here `xn` is the external node number, `n` the internal node number, `nFS` is the decimally packed Node Freedom Signature (7.3), `ndof` is the count of assigned freedoms at the node, and `nFDS` the Node Freedom Direction Selector (7.5). The `nFDS` may be altogether omitted if the freedom directions are aligned with the global directions.

EXAMPLE 7.1

All elements of a FEM model are solid elements (bricks or tetrahedra) with corner nodes only. All freedom directions are aligned with the global system, and consequently the `nFDS` may be omitted. A typical node has three assigned freedoms, which are the three displacement components labeled `tx`, `ty` and `tz`. The `nFL` of all unconstrained nodes (nodes not subjected to displacement BCs or MFCs) is

$$\texttt{\{ xn, 111000, 3 \}} \tag{7.7}$$

where `xn` is the external node number.

EXAMPLE 7.2

All elements of a FEM model are plate bending elements with corner nodes only. All freedom directions are aligned with the global system, and consequently the `nFDS` may be again omitted. A typical node has three assigned freedoms, which are the three displacement components labeled `tz`, `rx` and `ry`. The `nFL` of all unconstrained nodes is

```
{ xn, 001110, 3 }
```
(7.8)

where `xn` is the external node number.

EXAMPLE 7.3

Node 67 of a shell-plate-beam FEM model with six freedoms per node is subject to skew simply-support boundary conditions (BC). To simplify the BC specification without using MFCs, a freedom coordinate system $\{\bar{x}_{67}, \bar{y}_{67}, \bar{z}_{67}\}$ is specified with Euler parameters $0.7, 0.1, -0.1, 0.7$ with respect to (x, y, z). In this FCS all translations are zero and so is the rotation about $\bar{y}_{67}$ but the other two rotations are free. The `nFL` for this node is

```
{67, 222121, 6, { "E",0.7,0.1,-0.1,0.7 }}
```
(7.9)

§7.3.2 The Individual Multiplier Freedom List

If MultiFreedom Constraints (MFCs) are specified and these are treated by Lagrange multipliers, a data structure called the Individual Multiplier Freedom List or `mFL` appears. There is one `mFL` for each MFC.

This configuration of this data structure for a MFC defined by element `e` and involving `nummfc` freedoms is

```
mFL(n) = { e, { nfc(1), ...  nfc(mfcdof) }}
```
(7.10)

Here `e` is the internal element number, and the `nfc(i)` are three-item node-freedom-coefficient sublists:

```
nfc(i) = { xn(i),f(i),c(i) }
```
(7.11)

where `xn(i)`, `f(i)` and `c(i)` are the $\mathtt{i}^{th}$ external node number, freedom index, and coefficient, respectively, incolved in the constraint. Note that the list following `e` in (7.10) is exactly the same as that stored in the `MFPT`, cf. Chapter 6.

EXAMPLE 7.4

Suppose that the following MFC, where parenthesized numbers are external node numbers,

```
4.5 tx(41) - 3.2 tz(11) + 5.9 rz(72) = 0
```
(7.12)

is assigned to element with internal number 48. The constraints involves three nodes: 41, 11 and 72. The freedom indices are 1, 3 and 6, and the coefficients are 4.5, −3.2 and 5.9, respectively. The `mFL` is

```
mFL(48) = { 48, {{41,1,4.5 }, {11,3,-3.2 }, {72,6,5.9 }}
```
(7.13)

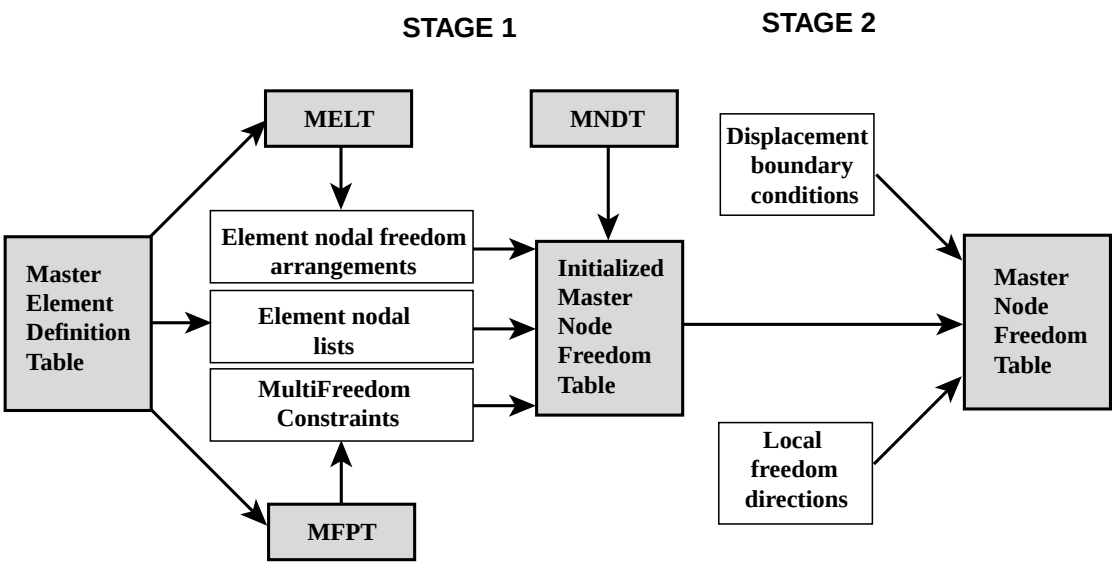


Figure 7.1. Construction of the Master Node Freedom Table as a two stage process.

REMARK 7.4

Note that this information cannot be assigned to individual nodes because MFCs are defined as elements. Thus it is necessary to keep `nFL` and `mFL` separate.

Also note that internal element numbers are carried in the `mFL`s whereas external node numbers are carried in the `nFL`s. This is dictated by the natural structure of the master tables defined below. In those tables there is a `nFL` for each defined node but there is no `mFL` for each defined element.

§7.4 MASTER FREEDOM TABLES

§7.4.1 The Master Node Freedom Table

The Master Node Freedom Table, or `MNFT`, is a flat list of the Individual Node Freedom Lists of all nodes in the FEM model, organized by ascending internal node numbers:

```
MNFT = { nFL(1), nFL(2),  ... nFL(numnod) }
```
(7.14)

The number in parenthesis is the internal node number `n` and not the external node number. Thus the `MNFT` has exactly the same arrangement as the `MNDT`. This allows the `nFL` for internal node `n` to be accessed simply as `nFL(n)`.

The `MNFT` is formed in two stages diagrammed in Figure 7.1:

Initialization. An initialized `MNFT` is constructed from information extracted from four master tables: `MNDT`, `MEDT`, `MELT` and `MFPT`. This operation records freedom assignments as well as Multifreedom Constraint information, because the MFCs are defined as elements. The freedom directions at each node are assumed to be the same as global directions.

Update. User-specified Displacement Boundary Conditions (DBC) are incorporated by tagging appropriate freedoms as passive. Freedom direction information that deviates from the default is inserted.

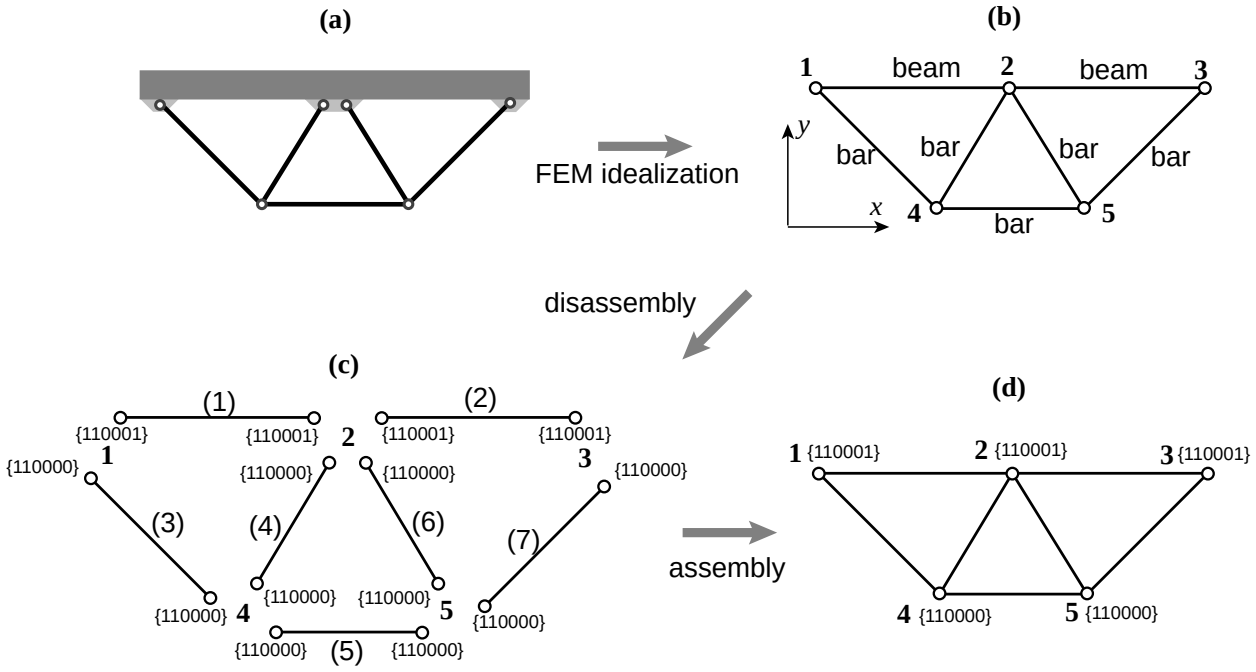


Figure 7.2. Example illustrating initialization of the `MNFT` through the rule (7.15). Numbers in parenthesis in (c) are internal element numbers.

EXAMPLE 7.5

The initialization stage is best explained graphically through the simple two-dimensional example depicted in Figure 7.2. A plane beam member is reinforced by five plane bar (truss) members, as shown in (a). The structure can only displaced in the $\{x, y\}$ plane but is otherwise free.

The finite element model idealization shown in (b) has five nodes and six elements: two beams and four bars. The model is disassembled into individual elements as shown in (c). The standard nodal freedom assignments for the disconnected nodes are shown as node freedom signatures enclosed in curly braces; a one means assigned or present whereas a zero means unassigned or absent. For example, a plane beam corner node has assigned freedoms `tx`, `ty` and `rz`, which encodes to the signature `110001`.

The structure is then reassembled as illustrated in (d). Nodal freedoms are constructed by the local-to-global freedom conservation rule:

If a freedom is assigned at the element level, it appears as such at the assembly level	(7.15)

Application of this rule shows that the signature of nodes 1, 2 and 3 is `110001` whereas that of nodes 4 and 5 is `110000`. The initialized `MNFT` is

```
{{1,110001,3}, {2,110001,3}, {3,110001,3}, {4,110000,2}, {5,110000,2}}
```  (7.16)

Consequently the model has $13 = 3 + 3 + 3 + 2 + 2$ assigned freedoms, which in the usual finite element notation form the node freedom vector

$$\mathbf{u}^T = [\, u_{x1} \quad u_{y1} \quad \theta_{z1} \quad u_{x2} \quad u_{y2} \quad \theta_{z2} \quad u_{x3} \quad u_{y3} \quad \theta_{z3} \quad u_{x4} \quad u_{y4} \quad u_{x5} \quad u_{y5} \,] \tag{7.17}$$

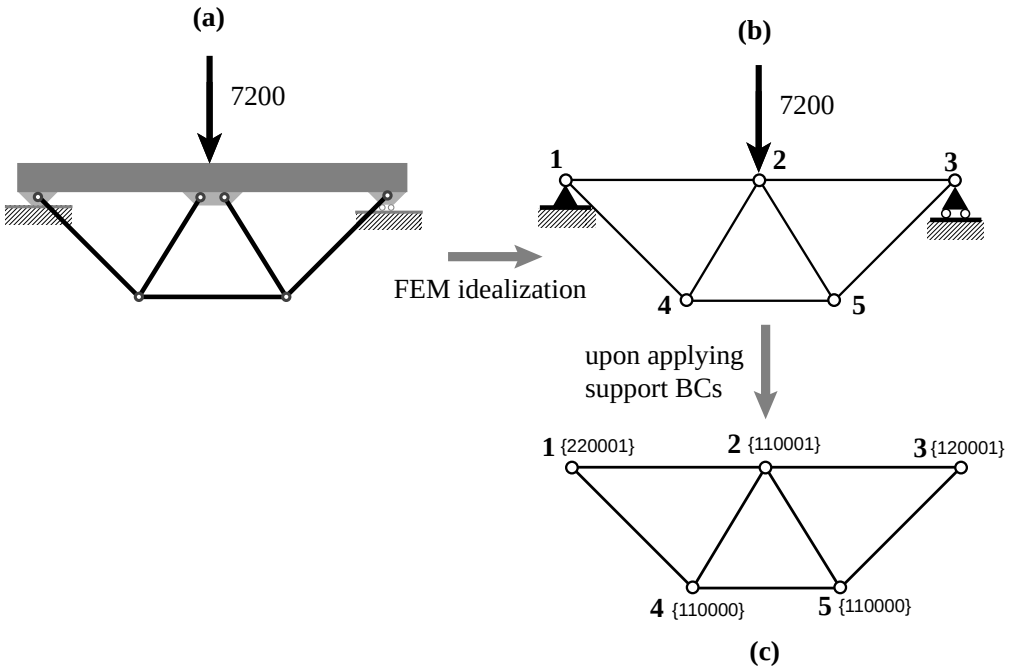


Figure 7.3. Support conditions and applied forces on the example structure of Figure 7.2.

EXAMPLE 7.6

So far the structure is unsupported. The displacement boundary conditions indicated in Figure 7.3 are $u_{x1} = u_{y1} = u_{y3} = 0$, which renders freedoms `tx(1)`, `ty(1)` and `ty(3)` passive. Upon marking these support conditions, the `MNFT` becomes

```
{{1,220001,3}, {2,110001,3}, {3,120001,3}, {4,110000,2}, {5,110000,2}}
```
(7.18)

EXAMPLE 7.7

To illustrate the effect of applying MultiFreedom Constraints (MFCs) on the `MNFT` of the structure of Figure 7.2, suppose that the user specifies that that both the x and y displacements of nodes 4 and 5 must be the same. Mathematically, $u_{x4} = u_{x5}$ and $u_{y4} = u_{y5}$. In freedom identifier notation this is

```
tx(4) − tx(5) = 0,   ty(4) − ty(5) = 0.
```
(7.19)

We assume that the two MFCs (7.19) are specified in the Master Element Definition Table as "MFC elements" with internal numbers 7 and 8, respectively. See Figure 7.4(a).

The results of applying the rule (7.15) and the MFC-marking convention of Table 7.2 is shown on the right of Figure 7.4. The initialized `MNFT` becomes

```
MNFT = {{110001}, {110001}, {110001}, {330000}, {330000}}
```
(7.20)

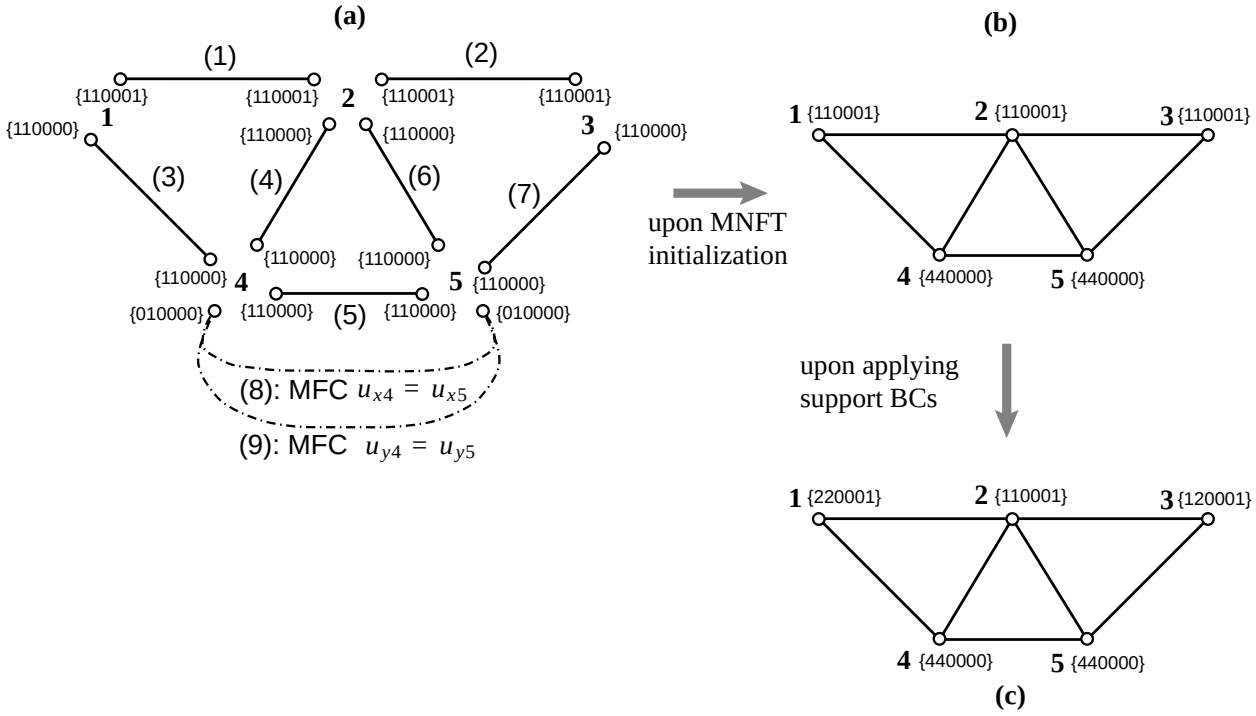


Figure 7.4. Example illustrating initialization of the MNFT when the MultiFreedom Constraints (7.19) are present. These are specified as MFC elements with internal numbers 8 and 9 that link nodes 4 and 5.

Note that a signature digit of 3 means that the freedom participates in *one* MFCs. This is the case here because each freedom assigned to nodes 4 and 5 appears exactly once in (7.16).

Assuming that the same support conditions of Figure 7.3 apply, we get finally

$$\texttt{MNFT = \{\{220001\}, \{110001\}, \{120001\}, \{330000\}, \{330000\}\}} \tag{7.21}$$

In addition to these changes in the MNFT, the presence of MFCs trated by Lagrange multipliers prompt the emergence of another data structure, called the Master Multiplier Freedom Table, which collects the mMLs described in §7.3.2. If MFCs are treated by penalty function methods, no additional tables appear.

§7.4.2 The Master Multiplier Freedom Table

If nummfc>0 MFCs are present and they are treated by Lagrange multipliers, a data structure called the Master Multiplier Freedom Table or MMFT is formed by stacking the Individual Multiplier Freedom List of all MFCs:

$$\boxed{\texttt{MMFT = \{mFL(1), ... mFL(nummfc) \}}} \tag{7.22}$$

where the numbers in parenthesis in (7.22) are constraint indices.

EXAMPLE 7.8

For the structure of Figure 7.4 under the two MFCs (7.19):

```
MMFT = {{8, {4,1,1},{5,1,-1}}, {9, {4,2,1},{5,2,-1}}}
```
(7.23)

REMARK 7.5

The `MMFT` contains simply a subset of information already present in the `MEDT` and `MFPT`. Thus its use is not strictly necessary should those tables are available. It is defined as a matter of convenience to head situations in which access to the complete element tables is precluded because it would waste storage. This is the case in parallel solution procedures, in which the `MEDT` is broken up into pieces for individual processors, or altogether unavailable. On the other hand, the `MMFT` is typically compact and can be replicated in each processor without serious storage penalties.

§7.5 IMPLEMENTATION OF FREEDOM ASSIGNMENT OPERATIONS

This section lists modules pertaining to the creation, access and display of the Master Freedom Table for nodes (the `MNFT`) and for multipliers (the `MMFT`). Modules are roughly listed in their order of appearance in the `Mathematica` file `FreedomTables.ma`.

§7.5.1 Initializing the MNFT

Module `InitializeMasterNodeFreedomTable` is displayed in Cell 7.1. It initializes the `MNFT` for the assembled FEM model by assigning appropriate freedoms to nodes. The initialized table does not mark for passive freedoms due to displacement boundary conditions, but it does account for MultiFreedom Constraints (MFCs).

The module receives four master tables as arguments: `MNDT`, `MEDT`, `MELT` and `MFPT`. The latter is used to extract MFC information. The output is the initialized `MNFT`.

As is evident from a glance at Cell 7.1, the code is more complex that anything presented so far, because of two reasons. First, the present implementation strives for minimal freedom assignement, that is, only the minimal number required to get the solution and nothing more. This goal requires more elaborate logic than the usual Fortran implementations that assume a fixed number of DOFs per node, because dynamic data structures must be used. Secons, the presence of MFC requires a second pass over the element tables.

The module calls the utility functions `UnpackNodeFreedomTags`, `PackNodeFreedomTags` and `CountAssignedNodeFreedoms`. These are described in §7.5.4.

Test statements for `InitializeMasterNodeFreedomTable` are shown separately in Cell 7.2 because of their length. These statements prepare the necessary data for the bar-beam finite element model of Figures 7.2–7.4, under the MFCs (7.19). The result of executing the program in Cells 7.1–7.2 (with the support of the utility functions noted above) is listed in Cell 7.3.

§7.5.2 Fixing Freedoms in the MNFT

The initialized `MNFT` does not account for displacement boundary conditions on individual freedoms. Those conditions can can be applied through the module `FixFreedomInMasterNodeFreedomTable`, which is listed in Cell 7.4.

Cell 7.1 Initializing the Master Node Freedom Table

```
InitializeMasterNodeFreedomTable[MNDT_,MEDT_,MELT_,MFPT_]:=
  Module[{e,eCL,ef,efp,efx,emfcl,eNL,i,itype,j,k,kNL,leneCL,leneNL,
   lenfpt,lenkNL,lenmfc,m,n,nf,nfp,NFT,nj,numele=Length[MEDT],
   numnod=Length[MNDT],numres,Nx2i,xn},
   NFT=Table[{0,0,0,{}},{numnod}]; emfcl={};
   k=0; Do [xn=MNDT[[n,1]]; k=Max[k,xn]; NFT[[n,1]]=xn, {n,1,numnod}];
   Nx2i=Table[0,{k}]; Do [xn=MNDT[[n,1]]; Nx2i[[xn]]=n, {n,1,numnod}];
   Do [If [MEDT[[e,2,1]]=="MFC", AppendTo[emfcl,e]; Continue[]];
      itype=MEDT[[e,2,2]]; eNL=MEDT[[e,3]];
      leneNL=Length[eNL]; efx=MELT[[itype,8]];
      Do [kNL=eNL[[k]]; lenkNL=Length[kNL]; If [lenkNL==0,Continue[]];
         efp=efx[[k]]; ef=UnpackNodeFreedomTags[efp];
         Do [xn=kNL[[i]]; If [xn<=0, Continue[]];
            n=Nx2i[[xn]]; If [n<=0, Continue[]]; nfp=NFT[[n,2]];
            If [nfp==efp, Continue[]]; nf=UnpackNodeFreedomTags[nfp];
            Do [nf[[j]]=Max[nf[[j]],ef[[j]]],{j,1,6}];
            nfp=PackNodeFreedomTags[nf]; NFT[[n,2]]=nfp;
            NFT[[n,3]]=CountAssignedNodeFreedoms[nfp],
         {i,1,lenkNL}],
      {k,1,leneNL}],
   {e,1,numele}];   (*Print["emfcl=",emfcl];*)
   lenmfc=Length[emfcl]; If [lenmfc==0, Return[NFT]];
   lenfpt=Length[MFPT]; numres=0;
   Do [If [Length[MFPT[[fc]]]==0, Break[]]; numres++, {fc,1,lenfpt}];
   Do [e=emfcl[[m]]; fc=MEDT[[e,4,2]]; eCL=MFPT[[fc+numres+1,5]];
      leneCL=Length[eCL];
      Do [xn=eCL[[k,1]]; If [xn<=0, Continue[]];
         n=Nx2i[[xn]]; If [n<=0, Continue[]];
         nf=UnpackNodeFreedomTags[NFT[[n,2]]];
         j=eCL[[k,2]]; nj=nf[[j]]; nf[[j]]=Min[Max[2,nj+1],8];
         nfp=PackNodeFreedomTags[nf]; NFT[[n,2]]=nfp;
         NFT[[n,3]]=CountAssignedNodeFreedoms[nfp],
      {k,1,leneCL}],
   {m,1,lenmfc}];
   Return[NFT];
 ];
```

The module receives as arguments the initialized `MNFT` and a list called `fFL`. This describes `m` specified displacements through `m` sublists `xn,i`, where `xn` is the external node number and `i` the freedom index. For example the `fFL`

$$\{\{1,1\},\ \{1,2\},\ \{3,2\}\} \tag{7.24}$$

declares that freedoms `tx(1)`, `ty(2)` and `ty(3)` are fixed. This implements the support conditions $u_{x1} = u_{y1} = u_{y3} = 0$ of Example 7.6.

Cell 7.2 Test Statements for Module of Cell 7.2

```
MNDT= {{1,{0,0,0}},{2,{5,0,0}},{3,{10,0,0}},{4,{-3,3,0}},{5,{-3,7,0}}};
MEDT= {{"Beam.1",{"Beam2D.2",2},{{1,2},{},{},{}}, {1,1}},
       {"Beam.2",{"Beam2D.2",2},{{2,3},{},{},{}}, {1,1}},
       {"Bar.1", {"Bar.2D",1},  {{1,4},{},{},{}}, {2,2}},
       {"Bar.2", {"Bar.2D",1},  {{2,4},{},{},{}}, {2,2}},
       {"Bar.3", {"Bar.2D",1},  {{4,5},{},{},{}}, {2,2}},
       {"Bar.4", {"Bar.2D",1},  {{2,5},{},{},{}}, {2,2}},
       {"Bar.5", {"Bar.2D",1},  {{3,5},{},{},{}}, {2,2}},
       {"MFC.1", {"MFC",3},     {{4,5},{},{},{}}, {0,3}},
       {"MFC.2", {"MFC",3},     {{4,5},{},{},{}}, {0,4}}};
MELT={{"Bar2D.2", "STM",2,"BAR",  "MOM","SEG",1000,{110000}},
      {"Beam2D.2","STM",3,"BEAM", "MOM","SEG",1000,{110001}},
      {"MFC",     "STM",2,"CON",  "EXT","ARB",1000,{0,0,0,0}}};
fPL1={"2DBeam",{"Xsec","MKFS"},{A},{Iz},{}};
fPL2={"2DBar", {"Xsec","MKSF"},{A},{},{}};
fPL3={"GenMFC",{"MFC","MKSF"},{},{},{{4,1,1},{5,1,-1}}};
fPL4={"GenMFC",{"MFC","MKSF"},{},{},{{4,2,1},{5,2,-1}}};
MFPT={{},fPL1,fPL2,fPL3,fPL4};
MNFT=InitializeMasterNodeFreedomTable[MNDT,MEDT,MELT,MFPT];
Print["MNFT=",MNFT];
```

Cell 7.3 Output from the Program of Cells 7.1 and 7.2

```
MNFT={{1, 110001, 3, {}}, {2, 110001, 3, {}}, {3, 110001, 3, {}},

   {4, 330000, 2, {}}, {5, 330000, 2, {}}}
```

The module returns the MNFT with the passive freedoms marked. It should be noticed that FixFreedomInMasterNodeFreedomTable will refuse marking unassigned freedoms.

The code is tested by the statements that follow the module. The initialized MNFT of Cell 7.3 is used as input. The displacement BCs $u_{x1} = u_{y1} = u_{y3} = 0$ described by (7.24) are applied. The results of running the test program are shown in Cell 7.5.

§7.5.3 Printing the Complete Master Node Freedom Table

Module PrintMasterNodeFreedomtable, listed in Cell 7.6, prints the complete Master Node Freedom Table supplied as argument. This module is tested using the MNFT of Cell 7.5. The output from the print routine is shown in Cell 7.7.

§7.5.4 Constructing the Master Multiplier Freedom Table

Module MakeMasterMultiplierFreedomTable, listed in Cell 7.8, builds the Master Multiplier

Cell 7.4 Fixing Freedoms in the MNFT

```
FixFreedomInMasterNodeFreedomTable[MNFT_,fFL_]:= Module [
  {NFT=MNFT,i,j,k,n,nf,Nx2i,numnod=Length[MNFT],xn},
   k=0; Do [k=Max[k,NFT[[n,1]]],{n,1,numnod}]; Nx2i=Table[0,{k}];
   Do [xn=NFT[[n,1]]; Nx2i[[xn]]=n, {n,1,numnod}];
   Do [{xn,j}=fFL[[i]]; n=Nx2i[[xn]]; If [n<=0,Continue[]];
     nf=UnpackNodeFreedomTags[NFT[[n,2]]]; Print["j=",j," nf=",nf];
     If[nf[[j]]>=0, nf[[j]]=1];
     NFT[[n,2]]=PackNodeFreedomTags[nf],
   {i,1,Length[fFL]}];
   Return[NFT];
 ];

MNFT={{1, 110001, 3, {}}, {2, 110001, 3, {}}, {3, 110001, 3, {}},
      {4, 330000, 2, {}}, {5, 330000, 2, {}}};
Print[FixFreedomInMasterNodeFreedomTable[MNFT,{{1,1},{1,2},{3,2}}]];
```

Cell 7.5 Output from the Program of Cell 7.4

```
{{1, 220001, 3, {}}, {2, 110001, 3, {}}, {3, 120001, 3, {}},

  {4, 330000, 2, {}}, {5, 330000, 2, {}}}
```

Freedom Table from information provided in its two arguments: `MEDT` and `MFPT`. The `MMFT` is returned as function value. If there are no MFCs, an empty list is returned.

The module is tested by the statements shown after the module. The test output is listed in Cell 7.9.

§7.5.5 Printing the Master Multiplier Freedom Table

Module `PrintMasterMultiplierFreedomTable`, listed in Cell 7.10, prints the complete Master Multiplier Freedom Table provided as argument.

The module is tested by the statements shown after the module, which input the `MMFT` output in Cell 7.9. The test output is listed in Cell 7.11.

§7.5.6 FCT Manipulation Utilities

Cell 7.12 show four utility functions that implement frequent operations on packed and unpacked freedom configurations. These functions are also used by modules described in other Chapters.

`PackNodeFreedomTags` receives as input six Freedom Configuration Tags and returns the packed Node Freedom Signature.

Cell 7.6 Printing the Master Node Freedom Table

```
PrintMasterNodeFreedomTable[MNFT_]:= Module[
  {numnod=Length[MNFT],t,n},
   t=Table["    ",{numnod+1},{4}];
   Do [t[[n+1,1]]=ToString[MNFT[[n,1]]];
       t[[n+1,2]]=ToString[MNFT[[n,2]]];
       t[[n+1,3]]=ToString[MNFT[[n,3]]];
       t[[n+1,4]]=ToString[MNFT[[n,4]]],
   {n,1,numnod}];
   t[[1]] = {"Xnode","Signature","DOF#","DOF-directions"};
   Print[TableForm[t,TableAlignments->{Bottom,Right},
           TableDirections->{Column,Row},TableSpacing->{0,2}]];
];

MNFT={{1, 220001, 3, {}}, {2, 110001, 3, {}}, {3, 120001, 3, {}},
      {4, 330000, 2, {}}, {5, 330000, 2, {}}};
PrintMasterNodeFreedomTable[MNFT];
```

Cell 7.7 Output from the Program of Cell 7.6

```
Xnode  Signature  DOF#  DOF-directions
    1     220001     3              {}
    2     110001     3              {}
    3     120001     3              {}
    4     330000     2              {}
    5     330000     2              {}
```

`UnpackNodeFreedomTags` does the reverse operation: receives the packed Node Freedom Signature and returns the six Freedom Configuration tags.

`UnpackAssignedNodeFreedomTags` receives the packed Node Freedom Signature and returns two items: the six Freedom Configuration tags, and a list of six integers { `fx1,...fx6` }. If the i-th freedom is assigned, `fxi` returns its ordinal in the cumulative count of assigned node freedoms, else zero.

`CountAssignedNodeFreedoms` receives the packed Node Freedom Signature as input and returns the count of freedoms assigned to the node.

The test statements shown at the bottom of Cell 7.12 exercise the four utility functions. The results from executing these statements is shown in Cell 7.13.

Cell 7.8 Constructing the Master Multiplier Freedom Table

```
MakeMasterMultiplierFreedomTable[MEDT_,MFPT_]:= Module[
  {e,fc,lenFPT=Length[MFPT],MFT,numele=Length[MEDT],numrfc},
   MFT={}; numrfc=0;
   Do [If [Length[MFPT[[fc]]]==0, Break[]]; numrfc++, {fc,1,lenFPT}];
   Do [If [MEDT[[e,2,1]]=="MFC", fc=MEDT[[e,4,2]];
       AppendTo[MFT,{e,MFPT[[fc+numrfc+1,5]]}]],
  {e,1,numele}];
  Return[MFT]
];

MNDT= {{1,{0,0,0}},{2,{5,0,0}},{3,{10,0,0}},{4,{-3,3,0}},{5,{-3,7,0}}};
MEDT= {{"Beam.1",{"Beam2D.2",2},{{1,2},{},{},{}}, {1,1}},
       {"Beam.2",{"Beam2D.2",2},{{2,3},{},{},{}}, {1,1}},
       {"Bar.1", {"Bar.2D",1},  {{1,4},{},{},{}}, {2,2}},
       {"Bar.2", {"Bar.2D",1},  {{2,4},{},{},{}}, {2,2}},
       {"Bar.3", {"Bar.2D",1},  {{4,5},{},{},{}}, {2,2}},
       {"Bar.4", {"Bar.2D",1},  {{2,5},{},{},{}}, {2,2}},
       {"Bar.5", {"Bar.2D",1},  {{3,5},{},{},{}}, {2,2}},
       {"MFC.1", {"MFC",3},     {{4,5},{},{},{}}, {0,3}},
       {"MFC.2", {"MFC",3},     {{4,5},{},{},{}}, {0,4}}};
MELT={{"Bar2D.2", "STM",2,"BAR",  "MOM","SEG",1000,{110000}},
      {"Beam2D.2","STM",3,"BEAM", "MOM","SEG",1000,{110001}},
      {"MFC",     "STM",2,"CON",  "EXT","ARB",1000,{0,0,0,0}}};
fPL1={"2DBeam",{"Xsec","MKFS"},{A},{Iz},{}};
fPL2={"2DBar", {"Xsec","MKSF"},{A},{},{}};
fPL3={"GenMFC",{"MFC","MKSF"},{},{},{{4,1,1},{5,1,-1}}};
fPL4={"GenMFC",{"MFC","MKSF"},{},{},{{4,2,1},{5,2,-1}}};
MFPT={{},fPL1,fPL2,fPL3,fPL4};
MMFT=MakeMasterMultiplierFreedomTable[MEDT,MFPT];
Print["MMFT=",MMFT];
```

Cell 7.9 Output from the Program of Cell 7.8

```
MMFT={{8, {{4, 1, 1}, {5, 1, -1}}}, {9, {{4, 2, 1}, {5, 2, -1}}}}
```

Cell 7.10 Printing the Master Multiplier Freedom Table

```
PrintMasterMultiplierFreedomTable[MMFT_]:= Module[
  {nummul=Length[MMFT],t,n},
   t=Table["    ",{nummul+1},{2}];
   Do [t[[n+1,1]]=PaddedForm[MMFT[[n,1]],4];
       t[[n+1,2]]=ToString[MMFT[[n,2]]],
   {n,1,nummul}];
   t[[1]] = {"Ielem","Node-freedom-coefficient list"};
   Print[TableForm[t,TableAlignments->{Right,Right},
         TableDirections->{Column,Row},TableSpacing->{0,2}]];
];

MMFT={{8, {{4, 1, 1}, {5, 1, -1}}}, {9, {{4, 2, 1}, {5, 2, -1}}}};
PrintMasterMultiplierFreedomTable[MMFT];
```

Cell 7.11 Output from the Program of Cell 7.10

```
Ielem  Node-freedom-coefficient list
    8        {{4, 1, 1}, {5, 1, -1}}
    9        {{4, 2, 1}, {5, 2, -1}}
```

Cell 7.12 Freedom Configuration Tag (FCT) Utility Functions

```
PackNodeFreedomTags[f_]:= Module[{},
   Return[ 100000*(f[[1]]+1)+10000*(f[[2]]+1)+1000*(f[[3]]+1)+
              100*(f[[4]]+1)   +10*(f[[5]]+1)     +(f[[6]]+1) ];
 ];

UnpackNodeFreedomTags[p_]:= Module[{f1,g2,f2,g3,f3,g4,f4,g5,f5,f6},
   If [p==0, Return[{-1,-1,-1,-1,-1,-1}]]; f1=Floor[p/100000];
   g2=p -100000*f1;   f2=Floor[g2/10000]; g3=g2-10000*f2;
   f3=Floor[g3/1000]; g4=g3-1000*f3;      f4=Floor[g4/100];
   g5=g4-100*f4;      f5=Floor[g5/10];    f6=g5-10*f5;
   Return[{f1,f2,f3,f4,f5,f6}-1];
 ];

UnpackAssignedNodeFreedomTags[p_]:= Module[
  {f1,g2,f2,g3,f3,g4,f4,g5,f5,f6,f,fx,i,j},
   fx={0,0,0,0,0,0}; If [p==0, Return[{fx-1,fx}]]; f1=Floor[p/100000];
   g2=p -100000*f1;   f2=Floor[g2/10000]; g3=g2-10000*f2;
   f3=Floor[g3/1000]; g4=g3-1000*f3;      f4=Floor[g4/100];
   g5=g4-100*f4;      f5=Floor[g5/10];    f6=g5-10*f5;
   f={f1,f2,f3,f4,f5,f6}; j=1; Do [If [f[[i]]>0,fx[[i]]=j++],{i,1,6}];
   Return[{f-1,fx}];
 ];

CountAssignedNodeFreedoms[p_]:= Module[{f1,g2,f2,g3,f3,g4,f4,g5,f5,f6},
   If [p==0, Return[0]]; f1=Floor[p/100000];
   g2=p -100000*f1;   f2=Floor[g2/10000]; g3=g2-10000*f2;
   f3=Floor[g3/1000]; g4=g3-1000*f3;      f4=Floor[g4/100];
   g5=g4-100*f4;      f5=Floor[g5/10];    f6=g5-10*f5;
   Return[Min[1,f1]+Min[1,f2]+Min[1,f3]+Min[1,f4]+Min[1,f5]+Min[1,f6]];
 ];

p=PackNodeFreedomTags[{1,-1,0,-1,0,3}]; Print["p=",p];
f=UnpackNodeFreedomTags[p]; Print["f=",f];
Print["p again=",PackNodeFreedomTags[f]];
Print["c=",CountAssignedNodeFreedoms[p]];
Print["a=",UnpackAssignedNodeFreedomTags[p]];
```

Cell 7.13 Output from the Program of Cell 7.12

```
p=201014
f={1, -1, 0, -1, 0, 3}
p again=201014
c=4
a={{1, -1, 0, -1, 0, 3}, {1, 0, 2, 0, 3, 4}}
```

-

8

Nodal State

This Chapter discusses the configuration and initialization of the state tables. Familiarity with the contents of Chapter 7 is essential.

§8.1 GENERAL DESCRIPTION

Chapter 7 describes how nodal displacement freedoms are assigned and their activity specified. But it does not explain how the *value* of those displacements and their conjugate forces is set. These (force,displacement) pairs collectively form the `nodal state`, which is stored in a Master Node State Table or `MNST`. One half of this information is known from force and displacement boundary conditions. The other half is initialized with zero-entry placeholders, and is completed by the equation solver.

If MultiFreedom Constraints (MFCs) are specified, the associated Lagrange multipliers become part of the state. Because multipliers are associated with elements and not nodes, they are held in their own separate Master Multiplier State Tables.

§8.2 INDIVIDUAL STATE LISTS

§8.2.1 The Individual Node State List

The Individual Node State Table, or `nSL`, records the numerical values of *assigned* freedoms and their conjugate forces at a specific node, as two sublists, with unassigned freedoms skipped. There is one `nSL` per node. The number of items in each `nSL` is twice the number of assigned freedoms.

To clarify the configuration of a `nSL`, suppose that only the three translation freedoms, as in a solid model, are assigned. The `nSL` configuration for a typical node `n` with external node number `xn` is

```
nSL(n) = { xn, { qxv(xn),qyv(xn),qzv(xn) }, { txv(xn),tyv(xn),tzv(xn) }}
```
(8.1)

Forces appear first, then displacements. Observe that values pertaining to nodal rotations and moments are left out of (8.1) because they are unassigned.

If a value is unknown, a zero is entered as a placeholder. This of course happens before the FEM equations are solved. Upon solution the placeholders are replaced by numeric (or symbolic) values delivered by the solver. These values are said to define the *state* of the node. The state of all nodes defines the state of the complete FEM model.

EXAMPLE 8.1

Consider external node 46 of a plate-bending FEM model, at which only the three freedoms `tz`, `rx` and `ry`, aligned with the global axes, are assigned. The corresponding conjugate forces are `qz`, `mx` and `my`. Assume that `qzv(46) = 42.5` and `my(46) = 0.808` are known forces and moments, while `rx(46) = -0.0293` is a known rotation. Their three conjugate values are unknown. Then the freedom signature and state list are

```
nFS(46) = 001210,   nST(46) = {46, { 42.5,0,0.808 }, { 0,-0.0293,0 }}
```
(8.2)

REMARK 8.1

In time-domain dynamic analysis, the configuration of the `nSL` has to be extended to include velocities and momenta, because these are part of the initial conditions. Alternatively, these may be placed in a separate table.

§8.2.2 The Individual Multiplier State List

If MultiFreedom Constraints (MFCs) are specified, and *they are treated by Lagrange multiplier adjunction*, for each constraint we define a data structure called the Individual Multiplier State List, or `mSL`. There is one `mSL` for each MFC. It consists of two items:

```
mSL = { crhs, lambda }
```
(8.3)

Here `crhs` is the given right hand side of the MFC, and `lambda` is a slot reserved for the Lagrange multiplier value. Upon solving the FEM equations the computed multiplier value is stored in the last item of (8.3).

This data structure is introduced here because is is closely related to the state tables defined below.

REMARK 8.2

If MFCs are treated by penalty element techniques, this data structure and those constructed from it are unnecessary. In the present implementation of `MathFET`, a treatment by Lagrange multipliers is used.

§8.3 THE MASTER NODE STATE TABLE(S)

The Master Node State Table, or `MNST`, is a flat list of the Individual Node State Lists of all nodes in the FEM model, ordered by internal node number:

```
MNST = {nSL(1), nSL(2),  ... nSL(numnod) }
```
(8.4)

The `MNST` is constructed in several stages as described below, and is not complete until the FEM equations are solved.

The preceding definition applies to a linear static analysis with a single load case. Much of the FEM analysis done in practice, however, is concerned with computing *a sequence of states*. Examples include static linear analyses with multiple load cases, eigenvalue analyses for vibration frequencies and modes, nonlinear and dynamic analyses.

These situations are handled by using *multiple* state tables, in which the results of each analysis are recorded. For example, if a static analysis involves 420 load cases, 420 `MNST`s are created. In a transient dynamic analysis using 2500 steps of direct time integration, 2501 `MNST`s are created, with the extra one accounting for the initial conditions.

REMARK 8.3

For large-scale models, the efficient handling of thousands of state tables can become a significant data management problem. This topic, howver, is beyond the scope of the present implementation.

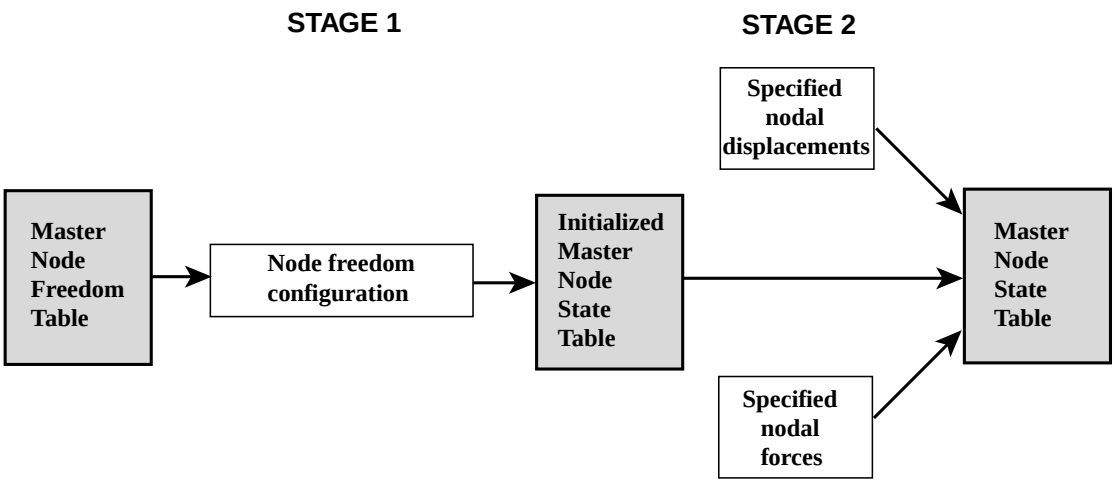


Figure 8.1. Construction of the Master Node State Table as a two stage process.

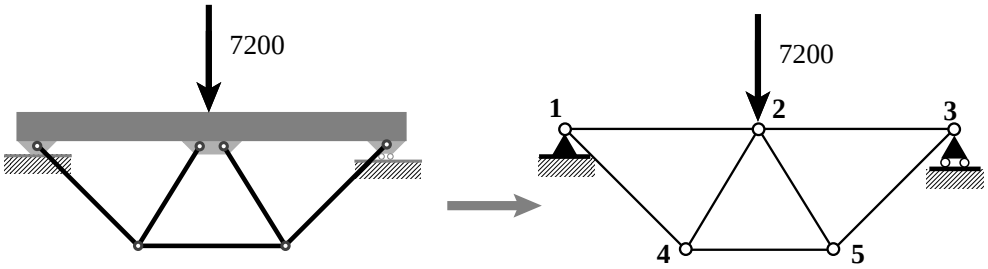


Figure 8.2. The example structure under loads and supports.

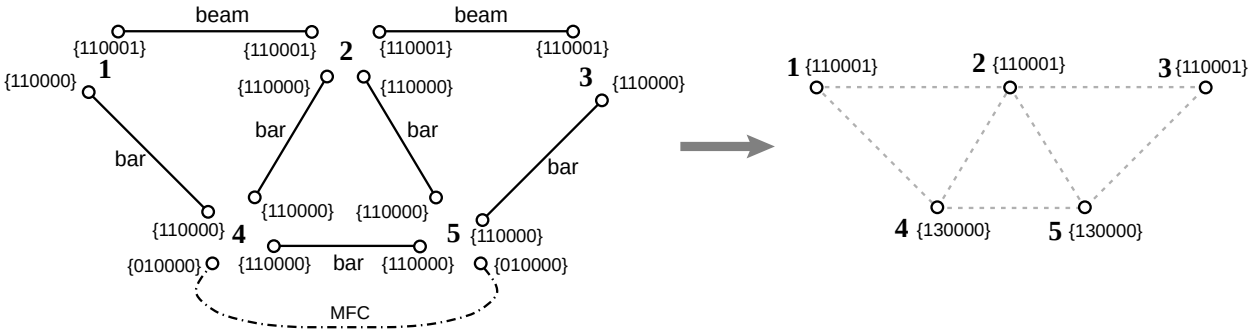


Figure 8.3. The example structure with additional MFCs.

§8.3.1 Construction of Master Node State Table

This is done in two stages. See Figure 8.1. An initialized `MNST` is constructed in node-by node fashion from the `MNFT` by reserving `nST` space for the assigned freedoms and associated forces. These reserved entries are set to zero. Then information on spacified node displacements and node forces is obtained from user input and placed in the appropriate slots.

EXAMPLE 8.2

For the example structure of Figure 8.2 the first stage yields the initialized table

```
MNST = {{{0,0,0},{0,0,0}}, {{0,0,0},{0,0,0}},
        {{0,0,0},{0,0,0}}, {{0,0},{0,0}}, {{0,0},{0,0}}}
```
(8.5)

Incorporation of the support and applied force conditions of Figure 8.3 changes the above to

```
MNST = {{{0.,0.,0},{0,0,0.}}, {{0,0,0},{0.,-7200.,0.}},
        {{0,0.,0},{0.,0.,0.}}, {{0,0},{0.,0.}}, {{0,0},{0.,0.}}}
```
(8.6)

in which the 13 `MNFT` entries that are known are shown as floating-point numbers. The value of the remaining 13 unknowns is to be provided eventually by the FEM equation solver.

REMARK 8.4

It would be elegant to be able to set the unknown entries to an "indefinite" value, but that kind of value is not supported uniformly by programming languages.

§8.3.2 The Master Multiplier Tables

If there are `Nc > 0` MultiFreedom Constraints, the `mFL`s of the MFCs are collected in a data structure called the Master Multiplier State Table, or `MMST`, ordered by increasing MFC number:

```
MMST = { mST(1), mST(2), ...   mST(Nc) }
```
(8.7)

where the `mSL`s are configured according to (8.3). This data structure is similar in function to the `MNST`.

§8.3.3 Accounting for MFC Lagrange Multipliers

If a MFC such as (8.6) is implemented through Lagrange multiplier adjunction, the vector of FEM unknowns gains one extra component:

$$\mathbf{u}^T = [\, u_{x1} \quad u_{y1} \quad \theta_{z1} \quad u_{x2} \quad u_{y2} \quad \theta_{z2} \quad u_{x3} \quad u_{y3} \quad \theta_{z3} \quad u_{x4} \quad u_{y4} \quad u_{x5} \quad u_{y5} \quad \lambda^7 \,] \tag{8.8}$$

Here λ^7 is the multiplier for the MFC, because the MFC is assumed to have been specified through an element with global number 7.

To facilitate the treatment of the multiplier, the two master data structures described in §8.4.3 are constructed. For this case they are

```
MMFT = { 7 }
```
(8.9)

```
MMST = {{ 0., 0 }}
```
(8.10)

If the constraint (8.6) has a nonzero right hand side, say

$$\texttt{ty(4)} - \texttt{ty(5)} = 0.24 \tag{8.11}$$

the last table would become

```
MMST = {{ 0.24, 0 }}
```
(8.12)

Cell 8.1 Initializing the Master Node State Table

```
InitializeMasterNodeStateTable[MNFT_]:= Module[
  {c,NST,numnod=Length[MNFT],xn},
   If [numnod==0, Return[{}]];
   NST=Table[{MNFT[[n,1]],{},{}},{n,1,numnod}];
   Do [c=MNFT[[n,3]];
      If [c>0, NST[[n,2]]=NST[[n,3]]=Table[0.,{c}]],
   {n,1,numnod}];
   Return[NST]
 ];

MNFT={{1, 110001, 3, {}}, {2, 110001, 3, {}}, {3, 110001, 3, {}},
      {4, 330000, 2, {}}, {5, 330000, 2, {}}};
MNST=InitializeMasterNodeStateTable[MNFT];
Print["Initialized MNST=",MNST//InputForm];
```

Cell 8.3 Output from the Program of Cell 8.1

```
Initialized MNST={{1, {0., 0., 0.}, {0., 0., 0.}},

   {2, {0., 0., 0.}, {0., 0., 0.}}, {3, {0., 0., 0.}, {0., 0., 0.}},

   {4, {0., 0.}, {0., 0.}}, {5, {0., 0.}, {0., 0.}}}
```

§8.4 IMPLEMENTATION OF STATE OPERATIONS

This section lists modules pertaining to the creation, access and display of the Master State Table for nodes (the `MNST`) and for multipliers (the `MMST`). Modules are roughly listed in their order of appearance in the `Mathematica` file `StateTables.ma`.

§8.4.1 Initializing the MNFT

Module `InitializeMasterNodeStateTable` is displayed in Cell 8.1. It initializes the `MNST` for the assembled FEM model.

The module receives the `MNFT` as argument, and the output is the initialized `MNST`.

Test statements for `InitializeMasterNodeStateTable` follow the module. These set up the data structures for the same test structure used in the last Chapter. The result of executing the program in Cell 8.1 is listed in Cell 8.2.

§8.4.2 Setting Up Force and Displacement Values in the MNFT

Nonzero forces are specified through module `SetForcesInMasterNodeStateTable`, which is listed in Cell 8.3. The output of the test statement is shown in Cell 8.4.

Cell 8.3 Setting Force Values in the MNST

```
SetForcesInMasterNodeStateTable[MNST_,forces_,MNFT_]:= Module[
  {ft,fx,i,j,k,lenfor=Length[forces],n,NST=MNST,
   numnod=Length[MNFT],Nx2i,val,xn},
   k=0; Do [k=Max[k,MNFT[[n,1]]],{n,1,numnod}]; Nx2i=Table[0,{k}];
   Do [xn=MNFT[[n,1]]; Nx2i[[xn]]=n, {n,1,numnod}];
   Do [{xn,j,val}=forces[[i]];
     n=Nx2i[[xn]]; If [n<=0,Continue[]];
     {ft,fx}=UnpackAssignedNodeFreedomTags[MNFT[[n,2]]];
     k=fx[[j]]; If [ft[[j]]!=1, NST[[n,2,k]]=val],
   {i,1,lenfor}];
   Return[NST]
 ];

MNFT={{1, 110001, 3, {}}, {2, 110001, 3, {}}, {3, 110001, 3, {}},
      {4, 330000, 2, {}}, {5, 330000, 2, {}}};
MNST={{1, {0., 0., 0.}, {0., 0., 0.}},
      {2, {0., 0., 0.}, {0., 0., 0.}}, {3, {0., 0., 0.}, {0., 0., 0.}},
      {4, {0., 0.}, {0., 0.}}, {5, {0., 0.}, {0., 0.}}};
forces= {{2,2,-7200},{3,1,4500},{5,2,-6000}};
MNST=SetForcesInMasterNodeStateTable[MNST,forces,MNFT];
Print["MNST=",MNST]; PrintMasterNodeStateTable[MNST];
```

Cell 8.4 Output from the Program of Cell 8.3

```
MNST={{1, {0., 0., 0.}, {0., 0., 0.}},

   {2, {0., -7200, 0.}, {0., 0., 0.}}, {3, {4500, 0., 0.}, {0., 0., 0.}},

   {4, {0., 0.}, {0., 0.}}, {5, {0., -6000}, {0., 0.}}}
Xnode  Node-forces      Node-displacements
    1  {0., 0., 0.}     {0., 0., 0.}
    2  {0., -7200, 0.}  {0., 0., 0.}
    3  {4500, 0., 0.}   {0., 0., 0.}
    4  {0., 0.}         {0., 0.}
    5  {0., -6000}      {0., 0.}
```

Nonzero prescribed displacements are specified through module `SetForcesInMasterNodeStateTable`, which is listed in Cell 8.5. The output of the test statement is shown in Cell 8.6.

§8.4.3 Printing the Complete Master Node State Table

Cell 8.5 Setting Displacement Values in the MNST

```
SetDisplacementsInMasterNodeStateTable[MNST_,disp_,MNFT_]:= Module[
  {ft,fx,i,j,k,lendis=Length[disp],n,NST=MNST,
   numnod=Length[MNFT],Nx2i,val,xn},
   k=0; Do [k=Max[k,MNFT[[n,1]]],{n,1,numnod}]; Nx2i=Table[0,{k}];
   Do [xn=MNFT[[n,1]]; Nx2i[[xn]]=n, {n,1,numnod}];
   Do [{xn,j,val}=disp[[i]];
     n=Nx2i[[xn]]; If [n<=0,Continue[]];
     {ft,fx}=UnpackAssignedNodeFreedomTags[MNFT[[n,2]]];
     k=fx[[j]]; If [ft[[j]]==1, NST[[n,3,k]]=val],
   {i,1,lendis}];
   Return[NST]
 ];

MNFT={{1, 220001, 3, {}}, {2, 110001, 3, {}}, {3, 120001, 3, {}},
      {4, 330000, 2, {}}, {5, 330000, 2, {}}};
MNST={{1, {0., 0., 0.}, {0., 0., 0.}}, {2, {0., -7200, 0.},
      {0., 0., 0.}}, {3, {4500, 0., 0.}, {0., 0., 0.}},
      {4, {0., 0.}, {0., 0.}}, {5, {0., -6000}, {0., 0.}}};
disp= {{3,2,0.00274},{1,1,-.00168}};
MNST=SetDisplacementsInMasterNodeStateTable[MNST,disp,MNFT];
Print["MNST=",MNST//InputForm]; PrintMasterNodeStateTable[MNST];
```

Cell 8.6 Output from the Program of Cell 8.5

```
MNST={{1, {0., 0., 0.}, {-0.00168, 0., 0.}},

   {2, {0., -7200, 0.}, {0., 0., 0.}},

   {3, {4500, 0., 0.}, {0., 0.00274, 0.}}, {4, {0., 0.}, {0., 0.}},

   {5, {0., -6000}, {0., 0.}}}
Xnode  Node-forces      Node-displacements
    1  {0., 0., 0.}     {-0.00168, 0., 0.}
    2  {0., -7200, 0.}  {0., 0., 0.}
    3  {4500, 0., 0.}   {0., 0.00274, 0.}
    4  {0., 0.}         {0., 0.}
    5  {0., -6000}      {0., 0.}
```

Module `PrintMasterNodeStateTable`, listed in Cell 8.7, prints the complete Master Node State Table supplied as argument.. The output from the test statements is shown in Cell 8.8.

§8.4.4 Initializing the Master Multiplier State Table

Cell 8.7 Printing the Master Node State Table

```
PrintMasterNodeStateTable[MNST_]:= Module[
  {numnod=Length[MNST],t,n},
   t=Table["    ",{numnod+1},{3}];
   Do [t[[n+1,1]]=PaddedForm[MNST[[n,1]],4];
       t[[n+1,2]]=ToString[MNST[[n,2]]];
       t[[n+1,3]]=ToString[MNST[[n,3]]],
   {n,1,numnod}];
   t[[1]] = {"Xnode","Node-forces","Node-displacements"};
   Print[TableForm[t,TableAlignments->{Left,Left},
           TableDirections->{Column,Row},TableSpacing->{0,2}]];
];

MNST={{1, {0., 0., 0.}, {-0.00168, 0., 0.}},
      {2, {0., -7200, 0.}, {0., 0., 0.}},
      {3, {4500, 0., 0.}, {0., 0.00274, 0.}}, {4, {0., 0.}, {0., 0.}},
      {5, {0., -6000}, {0., 0.}}};
PrintMasterNodeStateTable[MNST];
```

Cell 8.8 Output from the Program of Cell 8.7

```
Xnode  Node-forces      Node-displacements
    1  {0., 0., 0.}     {-0.00168, 0., 0.}
    2  {0., -7200, 0.}  {0., 0., 0.}
    3  {4500, 0., 0.}   {0., 0.00274, 0.}
    4  {0., 0.}         {0., 0.}
    5  {0., -6000}      {0., 0.}
```

Module `InitMasterMultiplierStateTable`, listed in Cell 8.9, initializes the Master Multiplier State Table from information provided in its arguments. The module is tested by the statements shown after the module. The test output is listed in Cell 8.10.

§8.4.5 Seeting Gaps in the Master Multiplier State Table

Module `SetGapsInMasterMultiplierStateTable`, listed in Cell 8.11, stores the constraint RHS values (“gaps”) in the Master Multiplier State Table. The module is tested by the statements shown after the module. The test output is listed in Cell 8.12.

§8.4.6 Printing the Master Multiplier State Table

Module `PrintMasterMultiplierStateTable`, listed in Cell 8.13, prints the complete Master Multiplier State Table provided as argument. The module is tested by the statements shown after

Cell 8.9 Initializing the Master Multiplier State Table

```
InitializeMasterMultiplierStateTable[MEDT_]:= Module[
  {e,MST,numele=Length[MEDT]},
   MST={};
   Do [If [MEDT[[e,2,1]]=="MFC", AppendTo[MST,{e,{0,0}}]],
   {e,1,numele}];
   Return[MST]
 ];

MEDT= {{"Beam.1",{"Beam2D.2",2},{{1,2},{},{},{}}, {1,1}},
       {"Beam.2",{"Beam2D.2",2},{{2,3},{},{},{}}, {1,1}},
       {"Bar.1", {"Bar.2D",1},  {{1,4},{},{},{}}, {2,2}},
       {"Bar.2", {"Bar.2D",1},  {{2,4},{},{},{}}, {2,2}},
       {"Bar.3", {"Bar.2D",1},  {{4,5},{},{},{}}, {2,2}},
       {"Bar.4", {"Bar.2D",1},  {{2,5},{},{},{}}, {2,2}},
       {"Bar.5", {"Bar.2D",1},  {{3,5},{},{},{}}, {2,2}},
       {"MFC.1", {"MFC",3},     {{4,5},{},{},{}}, {0,3}},
       {"MFC.2", {"MFC",3},     {{4,5},{},{},{}}, {0,4}}};
MMST=InitializeMasterMultiplierStateTable[MEDT];
Print["MMST=",MMST];
```

Cell 8.10 Output from the Program of Cell 8.9

```
MMST={{8, {0, 0}}, {9, {0, 0}}}
```

the module. The test output is listed in Cell 8.14.

Cell 8.11 Setting Gaps in the Master Multiplier State Table

```
SetGapsInMasterMultiplierStateTable[MMST_,gaps_]:= Module[
  {e,i,k,lengap=Length[gaps],MST=MMST,m,nummul=Length[MMST],Ei2m,val},
   k=0; Do [k=Max[k,MST[[m,1]]],{m,1,nummul}]; Ei2m=Table[0,{k}];
   Do [e=MST[[m,1]]; Ei2m[[e]]=m, {m,1,nummul}];
   Do [{e,val}=gaps[[i]]; m=Ei2m[[e]]; If [m>0, MST[[m,2,1]]=val],
   {i,1,lengap}];
   Return[MST]
 ];

MMST={{8, {0, 0}}, {9, {0, 0}}};
MMST=SetGapsInMasterMultiplierStateTable[MMST,{{9,3.5},{8,-0.671}}];
PrintMasterMultiplierStateTable[MMST];
```

Cell 8.12 Output from the Program of Cell 8.11

```
Ielem  MFC-gap  MFC-multiplier
    8   -0.671              0
    9      3.5              0
```

Cell 8.13 Printing the Complete Master Multiplier State Table

```
PrintMasterMultiplierStateTable[MMST_]:= Module[
  {nummul=Length[MMST],t,n},
   t=Table["    ",{nummul+1},{3}];
   Do [t[[n+1,1]]=PaddedForm[MMST[[n,1]],4];
       t[[n+1,2]]=ToString[MMST[[n,2,1]]];
       t[[n+1,3]]=ToString[MMST[[n,2,2]]],
   {n,1,nummul}];
   t[[1]] = {"Ielem","MFC-gap","MFC-multiplier"};
   Print[TableForm[t,TableAlignments->{Right,Right},
           TableDirections->{Column,Row},TableSpacing->{0,2}]];
];

MMST={{8, {0,0}}, {9, {3.5,0}}, {12,{0.671,0}}};
PrintMasterMultiplierStateTable[MMST];
```

Cell 8.14 Output from the Program of Cell 8.13

```
Ielem  MFC-gap  MFC-multiplier
    8        0               0
    9      3.5               0
   12    0.671               0
```

-

10

Global Connectivity

§10.1 GENERAL DESCRIPTION

Global connectivity data structures specify information about the linkage between nodes, elements and freedoms of the global Finite Element model. This information is needed to set up efficient computational data structures for both sequential and parallel processing. The information is purely topological in nature because it does not depend on model geometry, fabrication or constitutive properties.

As can be expected, the principal attribute reflected by these data structures is *connection* in the sense of "attached to" or "belongs to." This property, which is defined more precisely in the next subsection, characterizes the sparseness of matrices and vectors that appear in direct and iterative solution methods.

This Chapter introduces data structures that precisely define connection attributes for the global (source) model. These are sufficient for sequential processing. Partitioned versions, which are suitable for parallel processing, are treated in Chapter 15.

§10.1.1 Connections

The *connection attribute* identifies attachment relationships between two finite element objects. In the following definitions, *internal node numbers* and *internal element numbers* are used.

The structural models of Figures 10.1 and 10.2 will be often used to illustrate connectivity concepts and table configuration. In this models external and internal node numbers are taken to be the same for simplicity, and the internal element numbers are shown enclosed in parentheses within the elements. Element names are not shown as they play no role in the following operations.

A node `n` is connected to element `e` if the node appears in the nodelist of the element. In the example of Figure 10.1, nodes 3 and 5 are connected to element 8.

Two nodes, `ni` and `nj`, are connected if they are connected to the same element. A node is connected to itself. In the example of Figure 10.1, nodes 2 and 5 are connected.

Two elements, `ei` and `ej` are connected if they have at least one common node. An element is connected to itself. In the example of Figure 10.2(a), elements 8 and 10 are connected because they share node 6.

Two degrees of freedom `fi` and `fj` are said to be connected if they belong to two connected nodes, or to the same node.

The opposite attribute is *disconnection*. Two objects that are not connected in the stated sense stated above are said to be *disconnected*. For example, in the model of Figure 10.2(a), nodes 6 and 10, and elements 2 and 4, are disconnected.

The presence of MultiFreedom Constraints does not change those definitions if each MFC is regarded as an element (and indeed, that is how they are defined). In the model of Figure10.2(b), where the added MFC is introduced as element (11), nodes 6 and 10 are now connected, and so are elements 1 and 3.

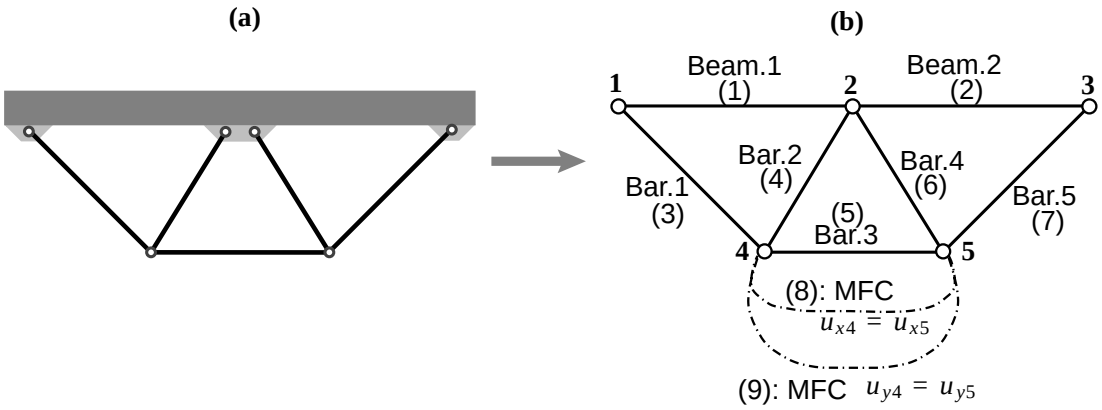


Figure 10.1. Plane skeletal structure with two MFCs for connectivity examples. External and internal node number coincide. Internal element numbers are shown in parenthesis alongside element names.

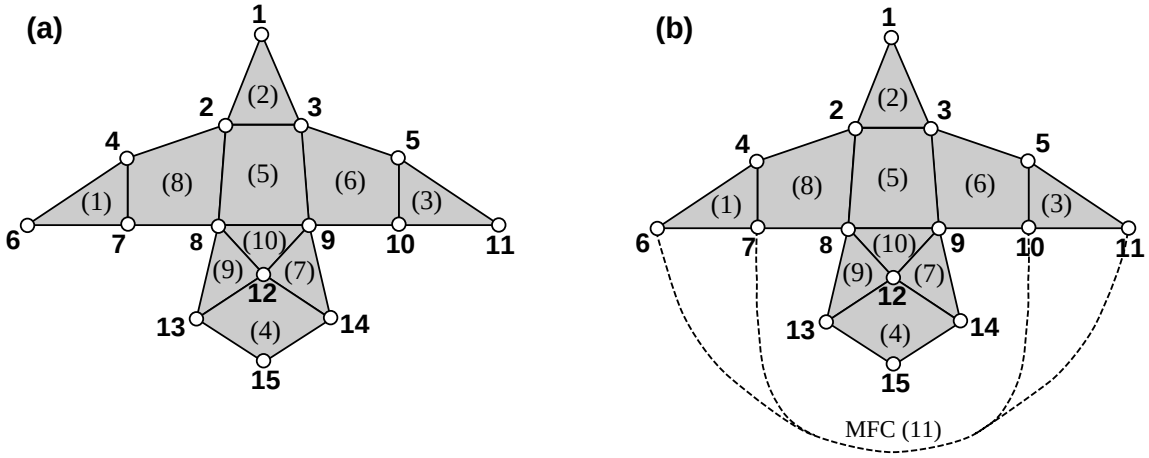


Figure 10.2. Continuum structure model for connectivity examples: (a) contains only standard elements, whereas (b) has a MFC, labeled as element 11, that links freedoms of nodes 6, 7, 8 and 9.

REMARK 10.1

For some table forms studied below, MFC elements are supposed to be ignored as regards connectivity. Such forms are called *eXcluded MFC versions* or *X versions* for short. They are identified by appending an `X` to the table name. X versions are useful for certain storage preallocation tasks when MFCs are treated by Lagrange multipliers.

§10.2 NODE TO ELEMENT CONNECTIONS

§10.2.1 The Individual Element-Node List

The Individual Element-Node List, or `eNL`, of an individual element is a list of all nodes connected to that element, specified by *internal* node number. For a specific element `e` we write `eNL(e)`.

The list is readily constructed from the information on element nodes stored in the Individual Element Definition List or `eDL` discussed in Chapter 3.3. Recall that the information was stored as a two-level list called `eXNL`. The construction of the `eNL` involves the following operations:

1. The `eXNL` is flattened because no distinction needs to be made between corner, side, face and internal nodes.
2. External node numbers are mapped to internal ones.
3. Optionally, nodes are sorted by increasing sequence.

The example below, which includes several node types, illustrates the procedure.

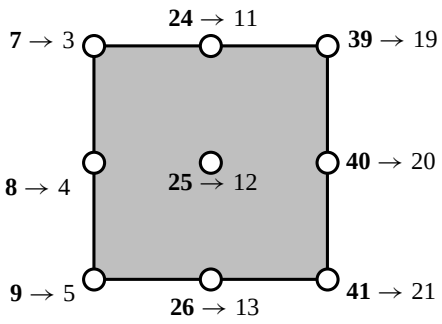


Figure 10.3. Example element to illustrate the construction of the Individual Element Node List. External node numbers appear in **boldface** on the left, and internal node numbers on the right.

EXAMPLE 10.1

The `element-nodes` list in the `eDL` of the 9-node quadrilateral element shown in Figure 10.3 is

$$\texttt{eXNL = \{\{}\mathbf{7,9,41,39}\texttt{\}, \{}\mathbf{8,26,40,24}\texttt{ \}, \{}\mathbf{25}\texttt{\}\}} \tag{10.1}$$

The list is flattened:

$$\texttt{eXNL = \{ }\mathbf{7,9,41,39,8,26,40,24,25}\texttt{ \}} \tag{10.2}$$

and the external node numbers replaced by internal ones:

$$\texttt{eNL = \{3,5,21,19,4,13,20,11,12 \}} \tag{10.3}$$

The replacement (10.2)→(10.3) can be efficiently done through the mapping array `Nx2i` described in §2.3. Finally, the node numbers in (10.3) may optionally be sorted:

$$\texttt{eNL = \{3,4,5,11,12,13,19,20,21 \}} \tag{10.4}$$

which speeds up some downstream operations.

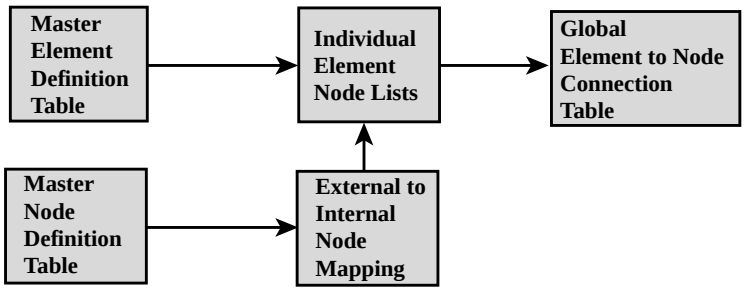


Figure 10.4. Construction of the GENCT.

§10.2.2 The Global Element To Node Connection Table

The Global Element-to-Node Connection Table, or GENCT, is the sequential list of all eNL(e):

$$\texttt{GENCT = \{ eNL(1), eNL(2), ... eNL(numele) \}} \tag{10.5}$$

This data structure is obtained by constructing the eNLs as described in §10.2.1, in a loop over elements. The schematics is illustrated in Figure 10.4.

For subsequent use of this data structure in the allocation of storage space for certain solution procedures, it is convenient to define an alternative version of GENCT, called the "eXcluded-MFC version" or "X version" and abbreviated to GENCTX. The X version assumes that MFC elements are *nodeless* and their eNLs are empty. In other words, MFC elements are viewed as disconnected from the FEM model. The main use of the X version is in the preparation for skyline solver processing, as described in Chapter 14.

EXAMPLE 10.2

For the structure of Figure 10.1:

$$\texttt{GENCT = \{\{1,2\},\{2,3\},\{1,4\},\{2,4\},\{4,5\},\{2,5\},\{3,5\},\{4,5\},\{4,5\}\}} \tag{10.6}$$

whereas

$$\texttt{GENCTX = \{\{1,2\},\{2,3\},\{1,4\},\{2,4\},\{4,5\},\{2,5\},\{3,5\},\{\},\{\}\}} \tag{10.7}$$

because the last two elements (8 and 9) are MFCs.

§10.2.3 The Individual Node-Element List

The inverse of the eNL specifies all elements connected to an individual node. Both the node and the elements are identified by *internal* numbers. This data structure is called the Individual Node Element List, or nEL. For a specific node n we write nEL(n).

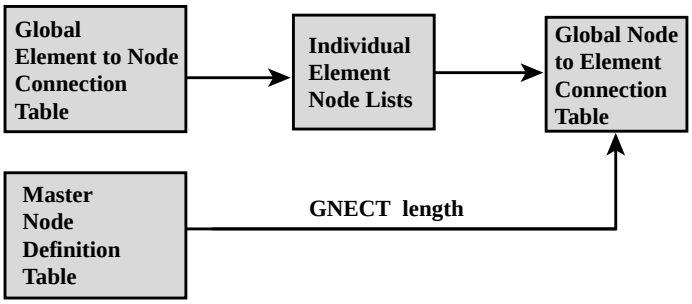


Figure 10.5. Schematics of construction of the GNECT.

EXAMPLE 10.3

For the model of Figure 10.2(b):

$$\texttt{nEL(1) = \{2\},} \qquad \texttt{nEL(4) = \{1,8\},} \qquad \texttt{nEL(11) = \{1,3,6,8 \}} \tag{10.8}$$

In this example, internal element numbers are sorted by ascending order in each eNL, but this arrangement, while convenient, is an implementation option.

§10.2.4 The Global Node to Element Connection Table

The Global Node-to-Element Connection Table, or GNECT, is the sequential list of all nEL(n):

$$\texttt{GNECT = \{ nEL(1), nEL(2), ... nEL(numnod) \}} \tag{10.9}$$

This data structure is derived from the MEDT and Nx2i. The eNLs are constructed element by element, and the GNECT built by aggregation node by node. Figure 10.5 schematizes the table construction process.

The X version of GNECT, called GNECTL, is obtained by starting from the GENCTX instead of the GENCT. This version ignores all MFC elements.

EXAMPLE 10.4

For the example model of Figure 10.1

$$\texttt{GNECT = \{\{1,3\}, \{1,2,4,6\}, \{2,7\}, \{3,4,5,8,9\}, \{5,6,7,8,9\}\}} \tag{10.10}$$

$$\texttt{GNECTX = \{\{1,3\}, \{1,2,4,6\}, \{2,7\}, \{3,4,5\}, \{5,6,7\}\}} \tag{10.11}$$

EXAMPLE 10.5

For the example model of Figure 10.2(a)

$$\texttt{GNECT = \{\{2\}, \{2,5,8\}, \{2,5,6\}, \{1,8\}, \{3,6\}, ... \{4,7\}, \{4\}\}} \tag{10.12}$$

This is also the GNECTX for Figure 10.2(b).

§10.3 NODE TO NODE CONNECTIONS

§10.3.1 The Individual Node-Node List

The list of all nodes connected to a given node is called the Individual Node-Node List, or nNL. All nodes are identified by internal numbers. For a given node n we write nNL(n). Although a node is connected to itself, n is usually excluded from the list.

EXAMPLE 10.6

In the model of Figure 10.1, node 2 is connected to 4 other nodes:

$$\texttt{nNL(2) = \{ 1,3,4,5 \}} \tag{10.13}$$

EXAMPLE 10.7

In the model of Figure 10.2(a), node 7 is connected to four other nodes:

$$\texttt{nNL(8) = \{ 2, 4, 6, 8 \}} \tag{10.14}$$

whereas in the model of Figure 10.2(b), node 7 is connected to six other nodes:

$$\texttt{nNL(8) = \{ 2, 4, 6, 8, 10, 11 \}} \tag{10.15}$$

because of the presence of the MFC element 11.

§10.3.2 The Global Node to Node Connection Table

The Global Node to Node Connection Table, or GNNCT, is simply the sequential list of all nNLs:

$$\texttt{GNNCT = \{ nNL(1), nNL(2) , ... nNL(numnod) \}} \tag{10.16}$$

This data structure can be constructed directly from the GNECT. See Figure 10.6.

The X version, called GNNCTX, is obtained by starting from the GNECTX; this version ignores connections due to MFCs.

EXAMPLE 10.8

For the example model of Figure 10.1

$$\texttt{GNNCT = \{\{2,4\}, \{1,3,4,5\}, \{2,5\},\{1,2,5\},\{2,3,4\}\}} \tag{10.17}$$

The GNNCTX is identical.

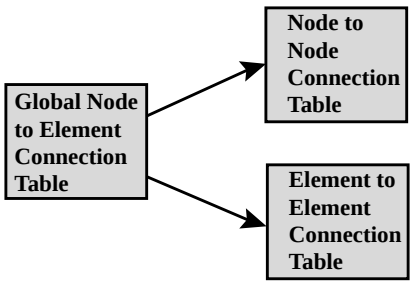


Figure 10.6. Schematics of the construction of the GNNCT and GEECT connection data structures.

EXAMPLE 10.9

For the example model of Figure 10.1(b)

```
GNNCT = {{2,3}, {1,3,7,8,9}, {1,2,5,8,9,10}, ...  {12,13,14}}
```
(10.18)

The GNNCTX is different in the eNLs of nodes 7 through 10.

REMARK 10.2

The GNNCT configuration illustrated in the above example records each connection relation between two nodes ni and nj twice, once in nNL(ni) and again in nNL(nj). It is possible to eliminate this redundancy by recording in nNL(n) only the internal nodes smaller (or greater) than n. Although this cuts down on the GNNCT storage by one half, it may lead to processing inefficiencies in some operations.

§10.4 ELEMENT TO ELEMENT CONNECTIONS

§10.4.1 The Individual Element-Element List

For some sparse matrix algorithms it is useful to have the list of all elements connected to a given element. This is called the Individual Element-Element List or eEL. Internal numbers are used for all elements. For element e we write eEL(e). Although an element is by definition connected to itself, e is usually excluded from the list.

EXAMPLE 10.10

In the model of Figure 10.1, element 1 is connected to two other elements:

```
eEL(1) = { 2,3 }
```
(10.19)

EXAMPLE 10.11

In the model of Figure 10.2(a), element 6 is connected to five other elements:

```
eEL(6) = { 2, 3, 5, 7, 10}
```
(10.20)

In the model of Figure 10.2(b), the Multifreedom Constraint element 11 is connected to four other elements:

```
eEL(11) = {1, 3, 6, 8}
```
(10.21)

§10.4.2 The Global Element to Element Connection Table

The Global Element to Element Connection Table, or GEECT, is simply the sequential list of all Individual Element-Element Lists::

$$\texttt{GEECT = \{ eEL(1), eEL(2) , ... eEL(numele) \}} \tag{10.22}$$

This data structure is easily constructed from the GNECT. See Figure 10.5. The X version, called GEECTX, is obtained by starting from the GNECTX.

EXAMPLE 10.12

In the model of Figure 10.1,

$$\texttt{GEECT} = \{\{2,3,4,6\}, \{1,4,6,7\}, \{1,4,5,8,9\}, \{1,2,3,5,6,8,9\}, \{3,4,6,7,8,9\}, \{1,2,4,5,7,8,9\}, \{2,5,6,8,9\}, \{3,4,5,6,7,9\}, \{3,4,5,6,7,8\}\} \tag{10.23}$$

$$\texttt{GEECTX} = \{\{2,3,4,6\}, \{1,4,6,7\}, \{1,4,5\}, \{1,2,3,5,6\}, \{3,4,6,7\}, \{1,2,4,5,7\}, \{2,5,6\}, \{\}, \{\}\} \tag{10.24}$$

EXAMPLE 10.13

In the example model of Figure 10.1(a) there are ten elements, and

$$\texttt{GEECT = \{\{8\}, \{5,6,8\}, ... \{4,5,6,7,8,9\}\}} \tag{10.25}$$

§10.5 IMPLEMENTATION OF GLOBAL CONNECTION TABLE CONSTRUCTION

This section lists several modules that built Global Connection Tables. Modules are roughly listed in their order of appearance in the Mathematica file GlobalConnectionTables.ma.

§10.5.1 Building the Global Element To Node Connection Table

Module MakeGlobalNodeToElementConnectionTable, listed in Cell 10.1, builds the Global Element To Node Connection Table or GENCT.

Module MakeGlobalElementToNodeConnectionTableX, also listed in that Cell, builds the X version or GENCTX, which eXcludes nodes of MFC elements. More precisely, if e is an MFC element, eNL(e) is considered empty. The X version is useful for certain pre-skyline operations when MFCs are treated via Lagrange multipliers. Both modules receive as arguments MEDT and MNDT and return the connection table as function value.

This module is exercised by the test statements shown in Cell 10.1, which pertain to the beam-bar structure of §7.6, augmented with the two constraints (7.19) between freedoms at nodes 4 and 5. This structure is reproduced in Figure 10.6 for convenience.

The test output of both versions is listed in Cell 10.2. Note that in the second form of GENCT element 8, which is the internal number of the MFC, has an empty eNL.

Cell 10.1 Making the Global Element To Node Connection Table

```
MakeGlobalElementToNodeConnectionTable[MEDT_,MNDT_]:= Module[
  {e,eNLf,GENCT,k,n,numele=Length[MEDT],numnod=Length[MNDT],Nx2i,xn},
   GENCT=Table[{},{numele}]; k=0;
   Do [k=Max[k,MNDT[[n,1]]],{n,1,numnod}]; Nx2i=Table[0,{k}];
   Do [xn=MNDT[[n,1]]; Nx2i[[xn]]=n, {n,1,numnod}];
   Do [eNLf=Flatten[MEDT[[e,3]]];
      Do [xn=eNLf[[i]]; If[xn>0,eNLf[[i]]=Nx2i[[xn]]],
      {i,1,Length[eNLf]}]; GENCT[[e]]=InsertionSort[eNLf],
  {e,1,numele}];
  Return[GENCT]
];
MakeGlobalElementToNodeConnectionTableX[MEDT_,MNDT_]:= Module[
  {e,eNLf,GENCT,k,n,numele=Length[MEDT],numnod=Length[MNDT],Nx2i,xn},
   GENCT=Table[{},{numele}]; k=0;
   Do [k=Max[k,MNDT[[n,1]]],{n,1,numnod}]; Nx2i=Table[0,{k}];
   Do [xn=MNDT[[n,1]]; Nx2i[[xn]]=n, {n,1,numnod}];
   Do [If [MEDT[[e,2,1]]=="MFC", Continue[]];
      eNLf=Flatten[MEDT[[e,3]]];
      Do [xn=eNLf[[i]]; If[xn>0,eNLf[[i]]=Nx2i[[xn]]],
      {i,1,Length[eNLf]}]; GENCT[[e]]=InsertionSort[eNLf],
  {e,1,numele}];
  Return[GENCT]
];

MNDT= {{1,{0,0,0}},{2,{5,0,0}},{3,{10,0,0}},{4,{-3,3,0}},{5,{-3,7,0}}};
MEDT= {{"Beam.1",{"Beam2D.2",2},{{1,2},{},{},{}}, {1,1}},
       {"Beam.2",{"Beam2D.2",2},{{2,3},{},{},{}}, {1,1}},
       {"Bar.1", {"Bar.2D",1},  {{1,4},{},{},{}}, {2,2}},
       {"Bar.2", {"Bar.2D",1},  {{2,4},{},{},{}}, {2,2}},
       {"Bar.3", {"Bar.2D",1},  {{4,5},{},{},{}}, {2,2}},
       {"Bar.4", {"Bar.2D",1},  {{2,5},{},{},{}}, {2,2}},
       {"Bar.5", {"Bar.2D",1},  {{3,5},{},{},{}}, {2,2}},
       {"MFC.1", {"MFC",3},     {{4,5},{},{},{}}, {0,3}},
       {"MFC.2", {"MFC",3},     {{4,5},{},{},{}}, {0,4}}};
GENCT=MakeGlobalElementToNodeConnectionTable[MEDT,MNDT];
Print["GENCT= ",GENCT];
GENCTX=MakeGlobalElementToNodeConnectionTableX[MEDT,MNDT];
Print["GENCTX=",GENCTX];
```

Cell 10.2 Output from the Program of Cell 10.1

```
GENCT= {{1, 2}, {2, 3}, {1, 4}, {2, 4}, {4, 5}, {2, 5}, {3, 5},

   {4, 5}, {4, 5}}
GENCTX={{1, 2}, {2, 3}, {1, 4}, {2, 4}, {4, 5}, {2, 5}, {3, 5}, {}, {}}
```

Cell 10.3 Making the Global Node To Element Connection Table

```
MakeGlobalNodeToElementConnectionTable[GENCT_,MNDT_]:= Module[
  {e,eNL,found,GNECT,lenenl,lennel,
   i,j,n,nEL,numele=Length[GENCT],numnod=Length[MNDT]},
   GNECT=Table[{},{numnod}];
   Do [eNL=GENCT[[e]]; lenenl=Length[eNL];
      Do [n=eNL[[i]]; If [n<=0,  Continue[]];
         nEL=GNECT[[n]];  lennel=Length[nEL]; found=False;
         Do [If [e==nEL[[j]], found=True; Break[]],{j,1,lennel}];
         If [found, Continue[]];
         If [lennel==0, GNECT[[n]]={e}, AppendTo[GNECT[[n]],e] ],
      {i,1,lenenl}],
   {e,1,numele}];
   (*Do [GNECT[[n]]=InsertionSort[GNECT[[n]]],
   {n,1,numnod}];  *) (* not needed I think *)
  Return[GNECT]
];

MNDT= {{1,{0,0,0}},{2,{5,0,0}},{3,{10,0,0}},{4,{-3,3,0}},{5,{-3,7,0}}};
GENCT= {{1,2}, {2,3}, {1,4}, {2,4}, {4,5}, {2,5}, {3,5}, {4,5}, {4,5}};
GNECT=MakeGlobalNodeToElementConnectionTable[GENCT,MNDT];
Print["GNECT= ",GNECT//InputForm];
GENCTX={{1,2}, {2,3}, {1,4}, {2,4}, {4,5}, {2,5}, {3,5}, {}, {}};
GNECTX=MakeGlobalNodeToElementConnectionTable[GENCTX,MNDT];
Print["GNECTX=",GNECTX//InputForm];
```

Cell 10.4 Output from the Program of Cell 10.3

```
GNECT= {{1, 3}, {1, 2, 4, 6}, {2, 7}, {3, 4, 5, 8, 9}, {5, 6, 7, 8, 9}}
GNECTX={{1, 3}, {1, 2, 4, 6}, {2, 7}, {3, 4, 5}, {5, 6, 7}}
```

§10.5.2 Building the Global Node To Element Connection Table

Module `MakeGlobalNodeToElementConnectionTable`, listed in Cell 10.3, receives as arguments `GENCT` and `MNDT` and returns the Global Node To Element Connection Table or `GNECT`.

To produce the X version of this table, simply supply `GENCTX`, produced by `MakeGlobalElementToNodeConnect` as first argument.

This module is exercised by the test statements shown in Cell 10.3. The test output is listed in Cell 10.4.

§10.5.3 Construction of the Global Node To Node Connection Table

Cell 10.5 Making the Global Node To Node Connection Table

```
MakeGlobalNodeToNodeConnectionTable[GNECT_,GENCT_]:= Module[
  {e,ee,eNL,found,i,j,GNNCT,lenenl,lennel,lennnl,n,nn,
   nEL,nNL,numele=Length[GENCT],numnod=Length[GNECT]},
   GNNCT=Table[{},{numnod}];
   Do [nEL=GNECT[[nn]]; lennel=Length[nEL];
       Do [e=nEL[[ee]]; eNL=GENCT[[e]]; lenenl=Length[eNL];
          Do [n=eNL[[i]]; nNL=GNNCT[[nn]]; lennnl=Length[nNL];
             If [n==nn, Continue[]]; found=False;
             Do [If [n==nNL[[j]], found=True; Break[]], {j,1,lennnl}];
             If [found, Continue[]];
             If [lennnl==0, GNNCT[[nn]]={n}, AppendTo[GNNCT[[nn]],n]],
          {i,1,lenenl}],
       {ee,1,lennel}];
       GNNCT[[nn]]=InsertionSort[GNNCT[[nn]]],
   {nn,1,numnod}];
   Return[GNNCT]
];

GENCT= {{1,2}, {2,3}, {1,4}, {2,4}, {4,5}, {2,5}, {3,5}, {4,5}, {4,5}};
GNECT= {{1,3}, {1,2,4,6}, {2,7}, {3,4,5,8,9}, {5,6,7,8,9}};
GNNCT=MakeGlobalNodeToNodeConnectionTable[GNECT,GENCT];
Print["GNNCT= ",GNNCT];
GENCTX={{1,2}, {2,3}, {1,4}, {2,4}, {4,5}, {2,5}, {3,5}, {}, {}};
GNECTX={{1,3}, {1,2,4,6}, {2,7}, {3,4,5}, {5,6,7}};
GNNCTX=MakeGlobalNodeToNodeConnectionTable[GNECTX,GENCTX];
Print["GNNCTX=",GNNCTX];
```

Cell 10.6 Output from the Program of Cell 10.5

```
GNNCT= {{2, 4}, {1, 3, 4, 5}, {2, 5}, {1, 2, 5}, {2, 3, 4}}
GNNCTX={{2, 4}, {1, 3, 4, 5}, {2, 5}, {1, 2, 5}, {2, 3, 4}}
```

Module `MakeGlobalNodeToNodeToConnectionTable`, listed in Cell 10.3, receives as arguments `GENCT` and `GNECT` and returns the Global Node To Node Connection Table or `GNNCT`.

To produce the X version, supply the X versions of `GENCT` and `GNECT` as arguments to this module.

This module is exercised by the test statements shown in Cell 10.5. The test output is listed in Cell 10.6.

Cell 10.7 Making the Global Element To Element Connection Table

```
MakeGlobalElementToElementConnectionTable[GNECT_,GENCT_]:= Module[
  {e,ee,eEL,found,GEECT,i,j,leneEL,lenenl,lennel,n,nn,nEL,
   numnod=Length[GNECT],numele=Length[GENCT]},
   GEECT=Table[{},{numele}];
    Do [eNL=GENCT[[ee]]; lenenl=Length[eNL];
       Do [n=eNL[[nn]]; nEL=GNECT[[n]]; lennel=Length[nEL];
          Do [e=nEL[[i]]; eEL=GEECT[[ee]]; leneEL=Length[eEL];
             If [e==ee, Continue[]]; found=False;
             Do [If [e==eEL[[j]], found=True; Break[]], {j,1,leneEL}];
             If [found, Continue[]];
             If [leneEL==0, GEECT[[ee]]={e}, AppendTo[GEECT[[ee]],e]],
          {i,1,lennel}],
       {nn,1,lenenl}];
       GEECT[[ee]]=InsertionSort[GEECT[[ee]]],
   {ee,1,numele}];
  Return[GEECT]
];

GENCT= {{1,2}, {2,3}, {1,4}, {2,4}, {4,5}, {2,5}, {3,5}, {4,5}, {4,5}};
GNECT= {{1,3}, {1,2,4,6}, {2,7}, {3,4,5,8,9}, {5,6,7,8,9}};
GEECT=MakeGlobalElementToElementConnectionTable[GNECT,GENCT];
Print["GEECT= ",GEECT];
GENCTX={{1,2}, {2,3}, {1,4}, {2,4}, {4,5}, {2,5}, {3,5}, {}, {}};
GNECTX={{1,3}, {1,2,4,6}, {2,7}, {3,4,5}, {5,6,7}};
GEECTX=MakeGlobalElementToElementConnectionTable[GNECTX,GENCTX];
Print["GEECTX=",GEECTX];
```

Cell 10.8 Output from the Program of Cell 10.7

```
GEECT= {{2, 3, 4, 6}, {1, 4, 6, 7}, {1, 4, 5, 8, 9},

   {1, 2, 3, 5, 6, 8, 9}, {3, 4, 6, 7, 8, 9}, {1, 2, 4, 5, 7, 8, 9},

   {2, 5, 6, 8, 9}, {3, 4, 5, 6, 7, 9}, {3, 4, 5, 6, 7, 8}}
GEECTX={{2, 3, 4, 6}, {1, 4, 6, 7}, {1, 4, 5}, {1, 2, 3, 5, 6},

   {3, 4, 6, 7}, {1, 2, 4, 5, 7}, {2, 5, 6}, {}, {}}
```

§10.5.4 Construction of the Global Element To Element Connection Table

Module `MakeGlobalElementToElementToConnectionTable`, listed in Cell 10.3, receives as arguments `GENCT` and `GNECT` and returns the `GEECT`. To produce the X version, supply X versions of `GENCT` and `GNECT` as arguments.

Cell 10.9 Insertion Sort Utility

```
InsertionSort[b_]:= Module[ {i,ii,j,jump,n,aj,a},
   n=Length[b]; a=b;
   Do [aj=a[[j]]; jump=False;
      Do [i=ii; If [a[[i]]<aj, jump=True; Break[],
         a[[i+1]]=a[[i]]], {ii,j-1,1,-1}];
      If [Not[jump],i=0]; a[[i+1]]=aj,
   {j,2,n}];
   Return[a];
];

Print[InsertionSort[{9,3,2,1,0,5,7,8,4,6}]];
```

Cell 10.10 Output from the Program of Cell 10.9

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

This module is exercised by the test statements shown in Cell 10.7. The test output is listed in Cell 10.8.

§10.5.5 Sort Utility

Uitility module `InsertionSort`, listed in Cell 10.9, sorts its integer argument list in ascending sequence using the insertion sort algorithm.

This function is used by the modules `MakeGlobalElementToNodeConnectionTable` and `MakeGlobalElementToNodeConnectionTableX` to sort the Individual Element Node Lists. It is an efficient sort method for short lists, as is the case with the `eNL`s.

This utility module is exercised by the test statement shown in Cell 10.9. The test output is listed in Cell 10.10.

11

Pre-Skyline Processing

This Chapter describes the construction of data structures that support a skyline sparse-storage arrangement to solve the finite element equations. Since these data structures also influence the assembly of the master stiffness equation, presentation at this stage is appropriate. The formation of the data structures receives the name *pre-skyline processing* or PSP.

§11.1 GENERAL DESCRIPTION

The Global Connection Tables described in Chapter 10are useful as starting points for any solution process, in sequential or parallel machines, using either direct and iterative solvers.

The present implementation of *MathFET* assumes that a *skyline* (also called profile) solver is used for sequential processing. The algorithmic details of that solution procedure are presented in the following Chapter. Here we are concerned with a preparatory stage, called pre-skyline processing or PSP, that defines the storage arrangement for the master stiffness equations.

PSP starts from the Global Node To Node Connection Table or `GNNCT` described in 10.4, and builds several data structures along the way. Three of them are node based, one (formed if there are MFCs treated by Lagrange multipliers) is multiplier based, and the last one, called the Global Skyline Diagonal Location Table or `GSDLT` is freedom based. The `GSDLT` represents the final result of the PSP in that it completely defines the configuration of a symmetric skyline matrix. A couple of other freedom address tables, with acronyms `GNFAT` and `GMFAT`, are useful in the assembly process.

The four data structures are presented as one-level (flat) lists, which reflects their implementation in *Mathematica*. In a lower level language such as C or Fortran they should be implemented as integer arrays for computational efficiency.

The example structures introduced in previous Chapters, one continuous and the other skeletal, are again used for illustration. They are reproduced in Figures 11.1 and 11.2 for convenience of the reader.

§11.2 THE GLOBAL NODE BANDWIDTH TABLE

The Global Node Bandwidth Table, or `GNBT` is an integer list with `numnod` entries. Given a node with internal number `n`, let `nlow` be the lowest internal node number connected to `n`, including `n` itself. Then set `GNBT(n)=nlow`.

This data structure obviously is a subset of the information in `GNNCT`. It may be easily formed from the `GNNCT` if available, or indirectly starting from the master element and node definition tables. The implementation displayed in §11.4 assumes the former case.

There is a second version of the `GNBT`, known as the "eXcluded MFC version" or "X version", which is abbreviated to `GNBTX`. This version ignores node connectivities due to MFCs, and must be used if the MFCs are treated by Lagrange multipliers, as in the present implementation of `MathFET`. It is generated by starting from the `GNNCTX` as source table.

EXAMPLE 11.1

For the skeletal structure of Figure 11.1,

$$\text{GNBT} = \{\, 1,1,2,1,2 \,\} \tag{11.1}$$

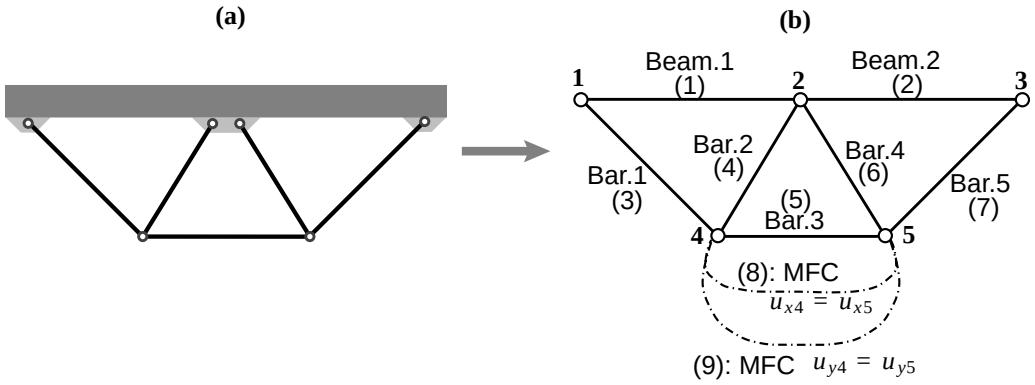


Figure 11.1. Example skeletal structure with two MFCs connecting nodes 4 and 5. External and internal node number coincide. Internal element numbers are shown in parenthesis alongside element names.

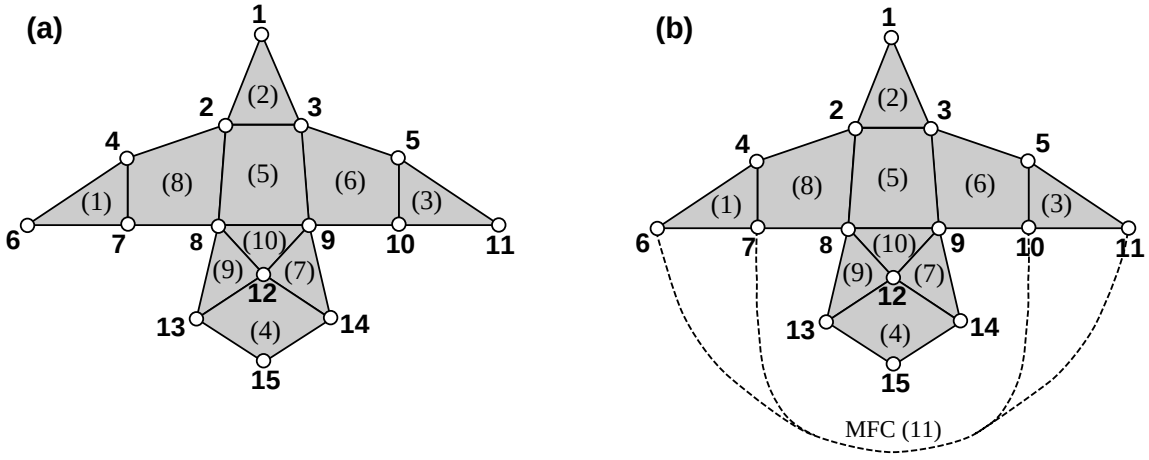


Figure 11.2. Example continuum structural model: (a) contains only standard plane stress elements, whereas (b) has a MFC, labeled as element 11, that links freedoms of nodes 6, 7, 8 and 9.

In this case the `GNBTX` is identical to the `GNCT` because the MFCs connect nodes 4 and 5, which are also linked by a bar member anyway.

EXAMPLE 11.2

For the continuum structure of Figure 11.2(a),

$$\text{GNBT} = \{\, 1,1,1,2,3,4,5,2,3,2,3,10,12,10,10,12 \,\} \tag{11.2}$$

REMARK 11.1

If MFCs are treated by penalty functions, the MFCs should be considered as direct node connectors, and the standard `GNBT` used.

§11.3 THE GLOBAL NODE FREEDOM COUNT TABLE

The Global Node Freedom Count Table, or GNFCT, holds the number of degrees of freedom assigned at each node. The count ignores activity.

EXAMPLE 11.3

For the skeletal structure of Figure 11.1,

$$\texttt{GNFCT} = \{3,3,3,2,2\} \tag{11.3}$$

EXAMPLE 11.4

If the model of Figure 11.2 represents a plane stress idealization, all 15 entries of GNFCT are 2. If it is a plate-bending idealization all entries are 3. If it is a shell structure all entries are either 5 or 6, depending on the degrees of freedom configuration of the membrane component.

REMARK 11.2

The freedom-count information is carried in the MNFT, as third item of each nFL. Consequently this data structure is redundant if MNFT is available, and in fact is not used in the present implementation of *MathFET* as a separate entity. It is presented here for instructional convenience.

§11.4 THE GLOBAL FREEDOM ADDRESS TABLES

Global Freedom Address Tables supply base addresses for accessing assembled equations from node and multiplier indices. There are two tables of this nature. One is associated with nodes, and always exists. The other is associated with multipliers, and exists only if there are MFCs and these are treated with Lagrange multipliers.

§11.4.1 The Global Node Freedom Address Table

The Global Node Freedom Address Table, or GNFAT, supplies base addresses for calculation of global freedom addresses from nodal information. The global address of the $\texttt{j}^{th}$ assigned freedom at node n is GNFAT(n)+j for j = 1,2,... GNFCT(n). The table actually contains numnod+1 entries. The last entry is set as: GNFAT(numnod+1)=numdof, where numdof is the total number of assigned fredoms.

The table is constructed by simply accumulating the entries of GNFCT, as the example below illustrates.

This table, in conjunction with GNFCT, is of fundamental importance in the assembly of the master stifness equations for the Direct Stiffness Method.

EXAMPLE 11.5

The skeletal example structure of Figure 11.1 has five nodes. Consequently the GNFAT contains six entries:

$$\texttt{GNFAT} = \{0,3,6,9,11,13\ \} \tag{11.4}$$

This is easily built from (11.3).

EXAMPLE 11.6

For the continuum structure of Figure 11.2 and assuming two freedoms per node:

$$\texttt{GNFAT = \{0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30\}} \tag{11.5}$$

§11.4.2 The Global Multiplier Freedom Address Table

This data structure is only generated when there are MultiFreedom Constraints treated with Lagrange multipliers, as in the present *MathFET* implementation. If there are no multipliers this table is empty.

The Global Multiplier Freedom Address Table, or GMFAT, supplies information for calculation of global freedom addresses given two pieces of information: the multiplier index and the freedom index. If there are nummul>0 Lagrange multipliers, the configuration of the GMFAT is

$$\texttt{\{mFA(1), ... mFA(nummul) \}} \tag{11.6}$$

where mFA are Individual Multiplier Freedom Address lists. If the $\texttt{m}^{th}$ multiplier involves k freedoms,

$$\texttt{mFA(m) = \{ j1,j2, ... jk \}} \tag{11.7}$$

where j1,j2,... are the global freedom addresses calculated through the GNFAT, *sorted in ascending order*.

EXAMPLE 11.7

For the skeletal example structure of Figure 11.1, which has two constraints:

$$\texttt{GMFAT = \{\{10,12\}, \{11,13\}\}} \tag{11.8}$$

Explanation for the first constraint, which is tx(4)=tx(5). This involves two freedoms tx(4) and tx(5), whose global addresses are GNFAT(4)+1=10 and GNFAT(5)+1=12, respectively; see (11.4). Likewise for the second constraint ty(4)=ty(5).

§11.5 THE GLOBAL SKYLINE DIAGONAL LOCATION TABLE

The Global Skyline Diagonal Location Table or GSDLT contains the diagonal addresses of a symmetric sparse matrix stored in a one-dimensional skyline array. Here only the configuration of GMDLT will be explained in enough detail to permit its construction. Further details as regards the assembly and solution of the master stiffness equations are provided in subsequent Chapters.

§11.5.1 The Global Equations

The present implementation of MathFET treats MultiFreedom Constraints (MFCs) by Lagrange multipliers. These multipliers are placed at the bottom of the solution vector. This leads to master stiffness equations of the form

$$\begin{bmatrix} \mathbf{K} & \mathbf{C}^T \\ \mathbf{C} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} \mathbf{f} \\ \mathbf{g} \end{bmatrix} \tag{11.9}$$

where the second matrix equation: $\mathbf{C}\boldsymbol{\lambda} = \mathbf{g}$ represents the MFCs. This overall equation fits the form

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{11.10}$$

where **A** is a symmetric sparse matrix. This matrix is stored in a skyline sparse format described in detail in the following Chapter.

§11.5.2 Configuration of the GSDLT

The `GSDLT` receives the addresses of the diagonals of **A**. It contains `numdof+nummul+1` entries, where `numdof` is the total number of assigned degrees of freedom and `nummul` the number of Lagrange multipliers. The first entry, `GSDLT(1)`, is conventionally zero.

Let `A(i,j)` be the (i,j)th entry of **A** for the global model. If the entry is inside the skyline, it is located at address `S(k)`, `k = GSDLT(i+1)+(j-i)`; else the entry is zero. This table (actually an array) is built from the three previous data structures.

If there are MFCs, a look at (11.9) shows that `GSDLT` splits into two parts. The first part has `numdof+1` entries and tracks the diagonal locations of **K**. This is called the *nodal* part of `GSLDT` because it is associated directly with nodal freedoms. The second part contains `nummul` entries and keeps track of the remaining diagonal locations. This is called the *multiplier* part of `GSLDT` because it is associated with the λ portion of the solution vector.

It follows that in general the `GSDLT` is constructed in two phases:

1. The nodal part of `GSDLT` is constructed from the information in `GNBT` and `GNFAT`.
2. If there are MFCs, the multiplier portion is constructed from information in `GNFAT` and `GMFAT`.

Algorithmic procedures are described in the implementation section below.

EXAMPLE 11.8

For the skeletal structure of Figure 11.1

```
GSDLT = { 0,1,3,6,10,15,21,25,30,36,46,57,66,76,81,86 }
```
(11.11)

The first 13+1=14 entries are the nodal part of the `GSDLT`. Entries 15 and 16 pertain to the multiplier part.

§11.6 IMPLEMENTATION OF GLOBAL CONNECTION TABLE CONSTRUCTION

This section lists several modules that built the pre-skyline data structures. Modules are roughly listed in their order of appearance in the `Mathematica` file `PreSkyTables.ma`.

Cell 11.1 Making the Global Node Bandwidth Table

```
MakeGlobalNodeBandwidthTable[GNNCT_]:= Module[
  {GNBT,n,nlow,numnod=Length[GNNCT]},
   GNBT=Table[0,{numnod}];
   Do [If [Length[GNNCT[[n]]]==0, nlow=n, nlow=Min[n,GNNCT[[n]]]];
       GNBT[[n]]=nlow,
   {n,1,numnod}];
  Return[GNBT]
];

GNNCTX= {{2, 4}, {1, 3, 4, 5}, {2, 5}, {1, 2, 5}, {2, 3, 4}};
GNBTX=MakeGlobalNodeBandwidthTable[GNNCTX]; Print["GNBTX= ",GNBTX];
```

Cell 11.2 Output from the Program of Cell 11.1

```
GNBTX= {1, 1, 2, 1, 2}
```

§11.6.1 Building the Node Bandwidth Table

Module `MakeGlobalNodeBandwidthTable`, listed in Cell 11.1, builds the Global Node Bandwidth Table or `GNBT` from the `GNNCT`.

To produce the X version of this table or `GNBTX`, which ignores MFC connections, use `GNNCTX` as input argument. The X version is the one needed for `MathFET`.

This module is exercised by the test statements shown in Cell 11.1, which pertain to the example structure of Figure 11.1.

§11.6.2 Building the Global Freedom Address Table

Module `MakeGlobalNodeFreedomAddressTable`, listed in Cell 11.3, receives as arguments `MNFT` and returns the Global Node Freedom Address Table or `GNFAT`.

This module is exercised by the test statements shown in Cell 11.3. The test output is listed in Cell 11.4.

§11.6.3 Building the Global Multiplier Freedom Address Table

Module `MakeGlobalMultiplierFreedomAddressTable`, listed in Cell 11.3, receives as arguments `MMFT`, `MNDT` and `GNFAT`, and returns the Global Multiplier Freedom Address Table or `GMFAT`.

This module is exercised by the test statements shown in Cell 11.5. The test output is listed in Cell 11.6.

Cell 11.3 Making the Global Node Freedom Address Table

```
MakeGlobalNodeFreedomAddressTable[MNFT_]:= Module[
  {c,GNFAT,n,numnod=Length[MNFT]},
   GNFAT=Table[0,{numnod+1}];
   Do [If [Length[MNFT[[n]]]==0, Continue[]]; c=MNFT[[n,3]];
       GNFAT[[n+1]]=GNFAT[[n]]+c,
   {n,1,numnod}];
  Return[GNFAT]
];

MNFT={{1,110001,3,{}},{1,110001,3,{}},{1,110001,3,{}},{1,110000,2,{}},
      {1,110000,2,{}}};
GNFAT=MakeGlobalNodeFreedomAddressTable[MNFT];
Print["GNFAT=",GNFAT//InputForm];
```

Cell 11.4 Output from the Program of Cell 11.3

```
GNFAT={0, 3, 6, 9, 11, 13}
```

Cell 11.5 Making the Global Multiplier Freedom Address Table

```
MakeGlobalMultiplierFreedomAddressTable[MMFT_,MNDT_,GNFAT_]:= Module[
  {coef,e,GMFAT,i,j,k,lenmCL,m,mCL,mf,n,
   nummul=Length[MMFT],numnod=Length[MNDT],Nx2i,xn},
   GMFAT=Table[{},{nummul}]; k=0;
   Do [k=Max[k,MNDT[[n,1]]],{n,1,numnod}]; Nx2i=Table[0,{k}];
   Do [xn=MNDT[[n,1]]; Nx2i[[xn]]=n, {n,1,numnod}];
   Do [mCL=MMFT[[m,2]];  lenmCL=Length[mCL];
      mf=Table[0,{lenmCL}];
      Do [{xn,j,coef}=mCL[[i]]; n=Nx2i[[xn]]; mf[[i]]=GNFAT[[n]]+j,
      {i,1,lenmCL}];
      GMFAT[[m]]=InsertionSort[mf],
  {m,1,nummul}];
  Return[GMFAT]
];

MMFT={{8, {{4, 1, 1}, {5, 1, -1}}}, {9, {{4, 2, 1}, {5, 2, -1}}}};
MNDT= {{1,{0,0,0}},{2,{5,0,0}},{3,{10,0,0}},{4,{-3,3,0}},{5,{-3,7,0}}};
GNFAT={0, 3, 6, 9, 11, 13};
GMFAT=MakeGlobalMultiplierFreedomAddressTable[MMFT,MNDT,GNFAT];
Print["GMFAT=",GMFAT//InputForm];
```

Cell 11.6 Output from the Program of Cell 11.5

```
GMFAT={{10, 12}, {11, 13}}
```

Cell 11.7 Making the Nodal Part of the Global Skyline Diagonal Location Table

```
MakeNPartOfGlobalSkymatrixDiagonalLocationTable[GNBT_,GNFAT_]:= Module[
  {c,GSDLT,i,j,k,n,nn,numdof,numnod=Length[GNBT]},
   numdof=GNFAT[[numnod+1]]; GSDLT=Table[0,{numdof+1}];
   Do [c=GNFAT[[n+1]]-GNFAT[[n]]; nn=GNBT[[n]];
       k=GNFAT[[nn]]; m=GNFAT[[n]];
       Do [GSDLT[[m+i+1]]=GSDLT[[m+i]]+m-k+i,{i,1,c}],
   {n,1,numnod}];
   Return[GSDLT]
];

GNBTX= {1, 1, 2, 1, 2};
GNFAT= {0, 3, 6, 9, 11, 13};
GSDLTX=MakeNPartOfGlobalSkymatrixDiagonalLocationTable[GNBTX,GNFAT];
Print["GSDLTX=",GSDLTX//InputForm];
```

Cell 11.8 Output from the Program of Cell 11.7

```
GSDLTX={0, 1, 3, 6, 10, 15, 21, 25, 30, 36, 46, 57, 66, 76}
```

§11.6.4 Building the Nodal Part of the Global Skyline Diagonal Location Table

Module `MakeNPartOfGlobalSkylineDiagonalLocationTable`, listed in Cell 11.7, receives as arguments `GNBTX` and `GNFAT` and returns the nodal part of the Global Skyline Diagonal Location Table. This is called `GSDLTX` to emphasize that it excludes MFCs. In this regard note that the first argument must be `GNBTX` and not `GNBT`.

This module is exercised by the test statements shown in Cell 11.7. The test output is listed in Cell 11.8.

§11.6.5 Building the Multiplier Part of the Global Skyline Diagonal Location Table

Module `MakeMPartOfGlobalSkylineDiagonalLocationTable`, listed in Cell 11.9, receives as arguments `GSDLTX` and `GMFAT`, where `GSDLTX` is produced by the module described in the previous subsection. It proceeds to append the multiplier-related portion of the diagonal locations of the

Cell 11.9 Making the Multiplier Part of the Global Skyline Diagonal Location Table

```
MakeMPartOfGlobalSkymatrixDiagonalLocationTable[GSDLTX_,GMFAT_]:=
   Module[{k,m,numdof,nummul=Length[GMFAT],p},
   If [nummul==0, Return[GSDLTX]];
   numdof=Length[GSDLTX]-1; p=Join[GSDLTX,Table[0,{nummul}]];
   Do [k=numdof+m; p[[k+1]]=p[[k]]+k-GMFAT[[m,1]]+1,
   {m,1,nummul}];
   Return[p]
];

GMFAT=  {{10, 12}, {11, 13}};
GSDLTX= {0, 1, 3, 6, 10, 15, 21, 25, 30, 36, 46, 57, 66, 76};
GSDLT=MakeMPartOfGlobalSkymatrixDiagonalLocationTable[GSDLTX,GMFAT];
Print["GSDLT=",GSDLT//InputForm];
numdof=Length[GSDLT]-1; numsky=GSDLT[[numdof+1]];
SymmSkyMatrixUpperTriangleMap[{GSDLT,Table[1.,{numsky}]}];
```

Cell 11.10 Output from the Program of Cell 11.9

```
GSDLT={0, 1, 3, 6, 10, 15, 21, 25, 30, 36, 46, 57, 66, 76, 81, 86}

                      1 1 1 1 1 1
    1 2 3 4 5 6 7 8 9 0 1 2 3 4 5

 1  + + + + + +       + +
 2    + + + + +       + +
 3      + + + +       + +
 4        + + + + + + + + +
 5          + + + + + + + +
 6            + + + + + + +
 7              + + + + + +
 8                + + + + +
 9                  + + + +
10                    + + + +
11                      + + + +
12                        + + +
13                          + +
14                            +
15
```

master equations. It returns the complete Global Skyline Diagonal Location Table or GSDLT as argument.

This module is exercised by the test statements shown in Cell 11.9, which produce the complete GSDLT for the structure of Figure 11.1. The test output is listed in Cell 11.10. The expression of

`GSDLT` agrees with (11.11).

Also shown in Cell 11.10 is the "skyline map" of the assembled 15×15 global coefficient matrix (11.9) for the example structure. This is done by building a fictitious skymatrix array of all ones, and calling `SymmSkyMatrixUpperTriangleMap` which is one of the skymatrix utility routines described in the following Chapter.

-

12

Symmetric Skyline Solver

§12.1 MOTIVATION FOR SPARSE SOLVERS

In the Direct Stiffness Method (DSM) of finite element analysis, the element stiffness matrices and consistent nodal force vectors are immediately assembled to form the *master stiffness matrix* and *master force vector*, respectively, by the process called *merge*. The basic rules that govern the merge process are described in Chapters 2 through 6 of the IFEM course. For simplicity the description that follows assumes that no MultiFreedom Constraints (MFCs) are present; their effect is studied in §12.6.

The end result of the assembly process are the master stiffness equations

$$\mathbf{K}\mathbf{u} = \mathbf{f} \tag{12.1}$$

where **K** is the master stiffness matrix, **f** the vector of node forces and **u** the vector or node displacements. Upon imposing the displacement boundary conditions, the system (12.1) is solved for the unknown node displacements. The solution concludes the main phase of DSM computations.

In practical applications the order of the stiffness system (12.1) can be quite large. Systems of order 1000 to 10000 are routinely solved in commercial software. Larger ones (say up to 100000 equations) are not uncommon and even millions of equations are being solved on suoercomputers. In *linear* FEM analysis the cost of solving this system of equations rapidly overwhelms other computational phases. Much attention has therefore given to matrix processing techniques that economize storage and solution time by taking advantage of the special structure of the stiffness matrix.

The master force vector is stored as a conventional one-dimensional array of length equal to the number N of degrees of freedom. This storage arrangement presents no particular difficulties even for very large problem sizes. Handling the master stiffness matrix, however, presents computational difficulties.

§12.1.1 The Curse of Fullness

If **K** is stored and processed as if it were a *full* matrix, the storage and processing time resources rapidly becomes prohibitive as N increases. This is illustrated in Table 12.1, which summarizes the storage and factor-time requirements for orders $N = 10^4$, 10^5 and 10^6.

As regards memory needs, a full square matrix stored without taking advantage of symmetry, requires storage for N^2 entries. If each entry is an 8-byte, double precision floating-point number, the required storage is $8N^2$ bytes. Thus, a matrix of order $N = 10^4$ would require 8×10^8 bytes or 800 MegaBytes (MB) for storage.

For large N the solution of (12.1) is dominated by the factorization of **K**, an operation discussed in §12.2. This operation requires approximately $N^3/6$ floating point operation units. [A floating-point operation unit is conventionally defined as a (multiply,add) pair plus associated indexing and data movement operations.] Now a fast workstation can typically do 10^7 of these operations per second, whereas a supercomputer may be able to sustain 10^9 or more. These times assume that the entire matrix is kept in high-speed memory; for otherwise the elapsed time may increase by factors of 10 or more due to I/O transfer operations. The elaspsed timed estimated given in Table 12.1 illustrate that for present computer resources, orders above 10^4 would pose significant computational difficulties.

Table 12.1 Storage & Solution Time for a Fully-Stored Stiffness Matrix

Matrix order N	Storage (double prec)	Factor op.units	Factor time workstation	Factor time supercomputer
10^4	800 MB	$10^{12}/6$	3 hrs	2 min
10^5	80 GB	$10^{15}/6$	4 mos	30 hrs
10^6	8 TB	$10^{18}/6$	300 yrs	3 yrs

Table 12.2 Storage & Solution Time for a Skyline Stored Stiffness Matrix Assuming $B = \sqrt{N}$

Matrix order N	Storage (double prec)	Factor op.units	Factor time workstation	Factor time supercomputer
10^4	8 MB	$10^8/2$	5 sec	0.05 sec
10^5	240 MB	$10^{10}/2$	8 min	5 sec
10^6	8000 MB	$10^{12}/2$	15 hrs	8 min

§12.1.2 The Advantages of Sparsity

Fortunately a very high percentage of the entries of the master stiffness matrix **K** are zero. Such matrices are call *sparse*. There are clever programming techniques that take advantage of sparsity that fit certain patterns. Although a comprehensive coverage of such techniques is beyond the scope of this course, we shall concentrate on a particular form of sparse scheme that is widely use in FEM codes: *skyline* storage. This scheme is simple to understand, manage and implement, while cutting storage and processing times by orders of magnitude as the problems get larger.

The skyline storage format is a generalization of its widely used predecessor called the *band storage* scheme. A matrix stored in accordance with the skyline format will be called a *skymatrix* for short. Only symmetric skymatrices will bve considered here, since the stiffness matrices in linear FEM are symmetric.

If a skymatrix of order N can be stored in S memory locations, the ratio $B = S/N$ is called the *mean bandwidth*. If the entries are, as usual, 8-byte double-precision floating-point numbers, the storage requirement is $8NB$ bytes. The factorization of a skymatrix requires approximately $\frac{1}{2}NB^2$ floating-point operation units. In two-dimensional problems B is of the order of $\sqrt{N}$. Under this assumption, storage requirements and estimated factorization times for $N = 10^4$, $N = 10^5$ and $N = 10^6$ are reworked in Table 12.2. It is seen that by going from full to skyline storage significant reductions in computer resources have been achieved. For example, now $N = 10^4$ is easy on a workstation and trivial on a supercomputer. Even a million equations do not look far-fetched on a supercomputer as long as enough memory is available.

In preparation for assembling **K** as a skymatrix one has to set up several auxiliary arrays related to

nodes and elements. These auxiliary arrays are described in the previous Chapter. Knowledge of that material is useful for understanding the following description.

§12.2 SPARSE SOLUTION OF STIFFNESS EQUATIONS

§12.2.1 Skyline Storage Format

The skyline storage arrangement for **K** is best illustrated through a simple example. Consider the 6×6 stiffness matrix

$$\mathbf{K} = \begin{bmatrix} K_{11} & 0 & K_{13} & 0 & 0 & K_{16} \\ & K_{22} & 0 & K_{24} & 0 & 0 \\ & & K_{33} & K_{34} & 0 & 0 \\ & & & K_{44} & 0 & K_{46} \\ & & & & K_{55} & K_{56} \\ symm & & & & & K_{66} \end{bmatrix} \tag{12.2}$$

Since the matrix is symmetric only one half, the upper triangle in the above display, need to be shown.

Next we define the *envelope* of **K** as follows. From each diagonal entry move *up* the corresponding column until the last nonzero entry is found. The envelope separates that entry from the rest of the upper triangle. The remaining zero entries are conventionally removed:

$$\mathbf{K} = \begin{bmatrix} K_{11} & & K_{13} & & & K_{16} \\ & K_{22} & 0 & K_{24} & & 0 \\ & & K_{33} & K_{34} & & 0 \\ & & & K_{44} & & K_{46} \\ & & & & K_{55} & K_{56} \\ symm & & & & & K_{66} \end{bmatrix} \tag{12.3}$$

What is left constitute the *skyline profile* of *skyline template* of the matrix. A sparse matrix that can be profitably stored in this form is called a *skymatrix* for brevity. Notice that the skyline profile may include zero entries. During the factorization step discussed below these zero entries will in general become nonzero, a phenomenon that receives the name *fill-in*.

The key observation is that only *the entries in the skyline template need to be stored,* because *fill-in in the factorization process will not occur outside the envelope*. To store these entries it is convenient to use a one-dimensional *skyline array*:

$$\mathtt{s}: \quad [\, K_{11},\, K_{22},\, K_{13},\, 0,\, K_{33},\, K_{24},\, K_{34},\, K_{44},\, K_{55},\, K_{16},\, 0,\, 0,\, K_{46},\, K_{56},\, K_{66}\,] \tag{12.4}$$

This array is complemented by a $(N+1)$ integer array `p` that contains addresses of *diagonal locations*. The array has $N+1$ entries. The $(i+1)^{th}$ entry of `p` has the location of the i^{th} diagonal entry of **K** in `s`. For the example matrix:

$$\mathtt{p}: \quad [\, 0,\, 1,\, 2,\, 5,\, 8,\, 9,\, 15\,] \tag{12.5}$$

In the previous Chapter, this array was called the Global Skyline Diagonal Location Table, or `GSDLT`.

Equations for which the displacement component is prescribed are identified by a *negative* diagonal location value. For example if u_3 and u_5 are prescribed displacement components in the test example, then

$$\texttt{p}: \quad [\,0,\ 1,\ 2,\ -5,\ 8,\ -9,\ 15\,] \tag{12.6}$$

REMARK 12.1

In Fortran it is convenient to dimension the diagonal location array as `p(0:n)` so that indexing begins at zero. In C this is the standard indexing.

§12.2.2 Factorization

The stiffness equations (12.1) are solved by a direct method that involves two basic phases: *factorization* and *solution*.

In the first stage, the skyline-stored symmetric stiffness matrix is factored as

$$\mathbf{K} = \mathbf{LDU} = \mathbf{LDL}^T = \mathbf{U}^T\mathbf{DU}, \tag{12.7}$$

where **L** is a unit lower triangular matrix, **D** is a nonsingular diagonal matrix, and **U** and **L** are the transpose of each other. The original matrix is overwritten by the entries of $\mathbf{D}^{-1}$ and **U**; details may be followed in the program implementation. No pivoting is used in the factorization process. This factorization is carried out by *Mathematica* module `SymmSkyMatrixFactor`, which is described later in this Chapter.

§12.2.3 Solution

Once **K** has been factored, the solution **u** for a given right hand side **f** is obtained by carrying out three stages:

$$\textit{Forward reduction}: \qquad \mathbf{Lz} = \mathbf{f}, \tag{12.8}$$

$$\textit{Diagonal scaling}: \qquad \mathbf{Dy} = \mathbf{z}, \tag{12.9}$$

$$\textit{Back substitution}: \qquad \mathbf{Uu} = \mathbf{y}, \tag{12.10}$$

where **y** and **z** are intermediate vectors. These stages are carried out by *Mathematica* modules `SymmSkyMatrixVectorSolve`, which is described later.

§12.2.4 Treating MFCs with Lagrange Multipliers

In the present implementation of `MathFET`, MultiFreedom Constraints (MFCs) are treated with Lagrange multiplier. There is one multiplier for each constraint. The multipliers are placed at the end of the solution vector.

Specifically, let the `nummul>0` MFCs be represented in matrix form as $\mathbf{Cu} = \mathbf{g}$, where $\mathbf{C}$ and $\mathbf{g}$ are given, and let the `nummul` multipliers be collected in a vector $\boldsymbol{\lambda}$. The multiplier-augmented master stiffness equations are

$$\begin{bmatrix} \mathbf{K} & \mathbf{C}^T \\ \mathbf{C} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} \mathbf{f} \\ \mathbf{g} \end{bmatrix} \tag{12.11}$$

or

$$\mathbf{Ax} = \mathbf{b}. \tag{12.12}$$

where the symmetric matrix $\mathbf{A}$, called a stiffness-bordered matrix, is of order `numdof+nummul`.

The stiffness bordered matrix is also stored in skyline form, and the previous solution procedure applies, as long as the skyline array is properly constructed as described in the previous Chapter.

The main difference with respect to the no-MFC case is that, because of the configuration (12.11), $\mathbf{A}$ can no longer be positive definite. In principle pivoting should be used during the factorization of $\mathbf{A}$ to forestall possible numerical instabilities. Pivoting can have a detrimental effect on solution efficiency because entries can move outside of the skyline template. However, by placing the $\boldsymbol{\lambda}$ at the end such difficulties will not be encountered if $\mathbf{K}$ is positive definite, and the constraints are linearly independent (that is, $\mathbf{C}$ has full rank). Thus pivoting is not necessary.

§12.3 A SKYSOLVER IMPLEMENTATION

The remaining sections of this revised Chapter describe a recent implementation of the skyline solver and related routines in *Mathematica* for *MathFET*. This has been based on similar Fortran codes used since 1967.

§12.3.1 Skymatrix Representation

In what follows the computer representation in *Mathematica* of a symmetric skymatrix will be generally denoted by the symbol `S`. Such a representation consists of a list of two numeric objects:

```
S = { p, s }
```
(12.13)

Here `p=GSDLT` is the Global Skyline Diagonal Location Table introduced in §11.6, and `s` is the array of skymatrix entries, arranged as described in the previous section. This array usually consists of floating-point numbers, but it may also contain exact integers or factions, or even symbolic entries.

For example, suppose that the numerical entries of the 6×6 skymatrix (12.10) are actually

$$\mathbf{K} = \begin{bmatrix} 11 & & 13 & & & 16 \\ & 22 & 0 & 24 & & 0 \\ & & 33 & 34 & & 0 \\ & & & 44 & & 46 \\ & & & & 55 & 56 \\ \mathit{symm} & & & & & 66 \end{bmatrix} \tag{12.14}$$

Cell 12.1 Factorization of a Symmetric SkyMatrix

```
SymmSkyMatrixFactor[S_,tol_]:= Module[
  {p,a,fail,i,j,k,l,m,n,ii,ij,jj,jk,jmj,d,s,row,v},
  row=SymmSkyMatrixRowLengths[S]; s=Max[row];
  {p,a}=S; n=Length[p]-1; v=Table[0,{n}]; fail=0;
  Do [jj=p[[j+1]]; If [jj<0|row[[j]]==0, Continue[]]; d=a[[jj]];
      jmj=Abs[p[[j]]]; jk=jj-jmj;
     Do [i=j-jk+k; v[[k]]=0; ii=p[[i+1]];
        If [ii<0, Continue[]]; m=Min[ii-Abs[p[[i]]],k]-1;
            ij=jmj+k; v[[k]]=a[[ij]];
            v[[k]]-=Take[a,{ii-m,ii-1}].Take[v,{k-m,k-1}];
            a[[ij]]=v[[k]]*a[[ii]],
     {k,1,jk-1}];
      d-=Take[a,{jmj+1,jmj+jk-1}].Take[v,{1,jk-1}];
      If [Abs[d]<tol*row[[j]], fail=j; a[[jj]]=Infinity; Break[] ];
      a[[jj]]=1/d,
  {j,1,n}];
  Return[{{p,a},fail}]
];

SymmSkyMatrixRowLengths[S_]:= Module[
  {p,a,i,j,n,ii,jj,m,d,row},
  {p,a}=S;  n=Length[p]-1; row=Table[0,{n}];
  Do [ii=p[[i+1]]; If [ii<0, Continue[]]; m=ii-i; row[[i]]=a[[ii]]^2;
     Do [If [p[[j+1]]>0, d=a[[m+j]]^2; row[[i]]+=d; row[[j]]+=d],
     {j,Max[1,Abs[p[[i]]]-m+1],Min[n,i]-1}],
  {i,1,n}];  Return[Sqrt[row]];
];
```

Its *Mathematica* representation, using the symbols (12.13) is

```
p= { 0,1,2,5,8,9,15};
s= { 11,22,13,0,33,24,34,44,55,16,0,0,46,56,66};          (12.15)
S= { p, s};
```

or more directly

```
S={ { 0,1,2,5,8,9,15},{11,22,13,0,33,24,34,44,55,16,0,0,46,56,66} };
                                                          (12.16)
```

§12.3.2 Skymatrix Factorization

Module `SymmSkyMatrixFactor`, listed in Cell 12.1, factors a symmetric skymatrix into the product **LDU** where **L** is the transpose of **U**. No pivoting is used. The module is invoked as

Cell 12.2 Factorization Test Input

```
ClearAll[n]; n=5; SeedRandom[314159];
p=Table[0,{n+1}]; Do[p[[i+1]]=p[[i]]+
    Max[1,Min[i,Round[Random[]*i]]],{i,1,n}];
    a=Table[1.,{i,1,p[[n+1]]}];
Print["Mean Band=",N[p[[n+1]]/n]];
S={p,a};
Sr=SymmSkyMatrixLDUReconstruct[S];
Print["Reconstructed SkyMatrix:"]; SymmSkyMatrixLowerTrianglePrint[Sr];
SymmSkyMatrixLowerTriangleMap[Sr];
Print["eigs=",Eigenvalues[SymmSkyMatrixConvertToFull[Sr]]];
x=Table[{N[i],3.,(-1)^i*N[n-i]},{i,1,n}];
Print["Assumed x=",x];
b=SymmSkyMatrixColBlockMultiply[Sr,x];
(*x=Transpose[x]; b=SymmSkyMatrixRowBlockMultiply[Sr,x];*)
Print["b=Ax=",b];
Print[Timing[{F,fail}=SymmSkyMatrixFactor[Sr,10.^(-12)]]];
If [fail!=0, Print["fail=",fail]; Abort[]];
Print["F=",F]; Print["fail=",fail];
Print["Factor:"];
SymmSkyMatrixLowerTrianglePrint[F];
x=SymmSkyMatrixColBlockSolve[F,b];
(*x=SymmSkyMatrixRowBlockSolve[F,b];*)
Print["Computed x=",x//InputForm];
```

```
{ Sf,fail } = SymmSkyMatrixFactor[S,tol]
```

The input arguments are

`S` The skymatrix to be factored, stored as the two-object list `{p,s}`; see previous subsection.

`tol` Tolerance for singularity test. The appropriate value of `tol` depends on the kind of skymatrix entries stored in `s`.

If the skymatrix entries are floating-point numbers handled by default in double precision arithmetic, `tol` should be set to $8\times$ or $10\times$ the machine precision in that kind of arithmetic. The factorization aborts if, when processing the `j-th` row, $d_j \le tol * r_j$, where d_j is the computed j^{th} diagonal entry of $\mathbf{D}$, and r_j is the Euclidean norm of the j^{th} skymatrix row.

If the skymatrix entries are exact (integers, fractions or symbols), `tol` should be set to zero. In this case exact singularity is detectable, and the factorization aborts only on that condition.

The outputs are:

Cell 12.3 Output from Program of Cells 12.1 and 12.2

```
Mean Band=1.6
Reconstructed SkyMatrix:

            Col   1    Col   2    Col   3    Col   4    Col   5
 Row   1     1.0000
 Row   2                1.0000
 Row   3                1.0000     2.0000
 Row   4                                     1.0000
 Row   5                           1.0000     1.0000     3.0000

     1 2 3 4 5

  1  +
  2    +
  3    + +
  4        +
  5      + + +
eigs={3.9563, 2.20906, 1., 0.661739, 0.172909}
Assumed x={{1., 3., -4.}, {2., 3., 3.}, {3., 3., -2.}, {4., 3., 1.},

   {5., 3., 0}}
b=Ax={{1., 3., -4.}, {5., 6., 1.}, {13., 12., -1.}, {9., 6., 1.},

   {22., 15., -1.}}
{0.0666667 Second, {{{0, 1, 2, 4, 5, 8},

    {1., 1., 1., 1., 1., 1., 1., 1.}}, 0}}
F={{0, 1, 2, 4, 5, 8}, {1., 1., 1., 1., 1., 1., 1., 1.}}
fail=0
Factor:

            Col   1    Col   2    Col   3    Col   4    Col   5
 Row   1     1.0000
 Row   2                1.0000
 Row   3                1.0000     1.0000
 Row   4                                     1.0000
 Row   5                           1.0000     1.0000     1.0000
Computed x={{1., 3., -4.}, {2., 3., 3.}, {3., 3., -2.}, {4., 3., 1.},

   {5., 3., 0.}}
```

`Sf` If `fail` is zero on exit, `Sf` is the computed factorization of `S`. It is a two-object list { p, du }, where du stores the entries of $\mathbf{D}^{-1}$ in the diagonal locations, and of $\mathbf{U}$ in its strict upper triangle.

`fail` A singularity detection indicator. A zero value indicates that no singularity was

Cell 12.4 Solving for a Single RHS

```
SymmSkyMatrixVectorSolve[S_,b_]:= Module[
  {p,a,n,i,j,k,m,ii,jj,bi,x},
  {p,a}=S; n=Length[p]-1; x=b;
  If [n!=Length[x], Print["Inconsistent matrix dimensions in",
    " SymmSkyMatrixVectorSolve"]; Return[Null]];
  Do [ii=p[[i+1]];If [ii>=0, Continue[]]; ii=-ii; k=i-ii+Abs[p[[i]]]+1;
      bi=x[[i]]; If [bi==0, Continue[]];
      Do [jj=p[[j+1]], If [jj<0, Continue[]];
         m=j-i; If [m<0, x[[j]]-=a[[ii+m]]*bi; Break[]];
         ij=jj-m; If [ij>Abs[p[[j]]], x[[j]]-=a[[ij]]*bi],
      {j,k,n}],
  {i,1,n}];
  Do [ii=p[[i+1]]; If [ii<0, x[[i]]=0; Continue[]];
     imi=Abs[p[[i]]]; m=ii-imi-1;
     x[[i]]-=Take[a,{imi+1,imi+m}].Take[x,{i-m,i-1}],
  {i,1,n}];
  Do [ii=Abs[p[[i+1]]]; x[[i]]*=a[[ii]], {i,1,n}];
  Do [ii=p[[i+1]]; If [ii<0, x[[i]]=b[[i]]; Continue[]];
     m=ii-Abs[p[[i]]]-1;
     Do [ x[[i-j]]-=a[[ii-j]]*x[[i]], {j,1,m}],
  {i,n,1,-1}];
  Return[x]
];
```

detected. If `fail` returns `j>0`, the factorization was aborted at the `j-th` row. In this case `Sf` returns the aborted factorization with ∞ stored in d_j.

A test of `SymmSkyMatrixFactor` on the matrix (12.14) is shown in Cells 12.2 and 12.3. The modules that print and produce skyline maps used in the test program are described later in this Chapter.

§12.3.3 Equation Solving

Module `SymmSkyMatrixVectorSolve`, listed in Cell 12.4, solves the linear system $\mathbf{Ax} = \mathbf{b}$ for $\mathbf{x}$, following the factorization of the symmetric skymatrix $\mathbf{A}$ by `SymmSkyMatrixFactor`. The module is invoked as

```
x = SymmSkyMatrixVectorSolve[Sf,b]
```

The input arguments are

`Sf`	The factored matrix returned by `SymmSkyMatrixFactor`.
`b`	The right-hand side vector to be solved for, stored as a single-level (one dimensional)

Cell 12.5 Solving for a Block of Righ Hand Sides

```
SymmSkyMatrixColBlockSolve[S_,b_]:= Module[
  {p,a,n,nrhs,i,j,k,m,r,ii,jj,bi,x},
  {p,a}=S; n=Length[p]-1; x=b;
  If [n!=Dimensions[x][[1]], Print["Inconsistent matrix dimensions in",
    " SymmSkyMatrixBlockColSolve"]; Return[Null]]; nrhs = Dimensions[x][[2]];
  Do [ii=p[[i+1]];If [ii>=0, Continue[]]; ii=-ii; k=i-ii+Abs[p[[i]]]+1;
     Do [bi=x[[i,r]]; If [bi==0, Continue[]];
        Do [jj=p[[j+1]], If [jj<0, Continue[]];
           m=j-i; If [m<0,x[[j,r]]-=a[[ii+m]]*bi; Break[]];
           ij=jj-m; If [ij>Abs[p[[j]]], x[[j,r]]-=a[[ij]]*bi],
        {j,k,n}],
     {r,1,nrhs}],
  {i,1,n}];
  Do [ii=p[[i+1]]; If [ii<0, Do[x[[i,r]]=0,{r,1,nrhs}];Continue[]];
      imi=Abs[p[[i]]]; m=ii-imi-1;
     Do [ Do [ x[[i,r]]-=a[[imi+j]]*x[[i-m+j-1,r]], {j,1,m}], {r,1,nrhs}],
  {i,1,n}];
  Do [ii=Abs[p[[i+1]]]; Do[x[[i,r]]*=a[[ii]], {r,1,nrhs}], {i,1,n}];
  Do [ii=p[[i+1]]; If [ii<0, Do[x[[i,r]]=b[[i,r]],{r,1,nrhs}];Continue[]];
     m=ii-Abs[p[[i]]]-1;
     Do [ Do [ x[[i-j,r]]-=a[[ii-j]]*x[[i,r]], {j,1,m}], {r,1,nrhs}],
  {i,n,1,-1}];
  Return[x]
];
SymmSkyMatrixRowBlockSolve[S_,b_]:= Module[
  {p,a,n,nrhs,i,j,k,m,r,ii,jj,bi,x},
  {p,a}=S; n=Length[p]-1; x=b;
  If [n!=Dimensions[x][[2]], Print["Inconsistent matrix dimensions in",
    " SymmSkyMatrixBlockRowSolve"]; Return[Null]]; nrhs = Dimensions[x][[1]];
  Do [ii=p[[i+1]];If [ii>=0, Continue[]]; ii=-ii; k=i-ii+Abs[p[[i]]]+1;
     Do [bi=x[[r,i]]; If [bi==0, Continue[]];
        Do [jj=p[[j+1]], If [jj<0, Continue[]];
           m=j-i; If [m<0,x[[j,r]]-=a[[ii+m]]*bi; Break[]];
           ij=jj-m; If [ij>Abs[p[[j]]], x[[r,j]]-=a[[ij]]*bi],
        {j,k,n}],
     {r,1,nrhs}],
  {i,1,n}];
  Do [ii=p[[i+1]]; If [ii<0, Do[x[[r,i]]=0,{r,1,nrhs}];Continue[]];
      imi=Abs[p[[i]]]; m=ii-imi-1;
     Do [ Do [ x[[r,i]]-=a[[imi+j]]*x[[r,i-m+j-1]], {j,1,m}], {r,1,nrhs}],
  {i,1,n}];
  Do [ii=Abs[p[[i+1]]]; Do[x[[r,i]]*=a[[ii]], {r,1,nrhs}], {i,1,n}];
  Do [ii=p[[i+1]]; If [ii<0, Do[x[[r,i]]=b[[r,i]],{r,1,nrhs}];Continue[]];
     m=ii-Abs[p[[i]]]-1;
     Do [ Do [ x[[r,i-j]]-=a[[ii-j]]*x[[r,i]], {j,1,m}], {r,1,nrhs}],
  {i,n,1,-1}];
  Return[x]
];
```

Cell 12.6 Multiplying Skymatrix by Individual Vector

```
SymmSkyMatrixVectorMultiply[S_,x_]:= Module[
  {p,a,n,i,j,k,m,ii,b},
  {p,a}=S; n=Length[p]-1;
  If [n!=Length[x], Print["Inconsistent matrix dimensions in",
    " SymmSkyMatrixVectorMultiply"]; Return[Null]];
  b=Table[a[[ Abs[p[[i+1]]] ]]*x[[i]], {i,1,n}];
  Do [ii=Abs[p[[i+1]]]; m=ii-Abs[p[[i]]]-1; If [m<=0,Continue[]];
     b[[i]]+=Take[a,{ii-m,ii-1}].Take[x,{i-m,i-1}];
     Do [b[[i-k]]+=a[[ii-k]]*x[[i]],{k,1,m}],
  {i,1,n}];
  Return[b]
 ];

 (*
 ClearAll[n]; n=10; SeedRandom[314159];
 p=Table[0,{n+1}]; Do[p[[i+1]]=p[[i]]+
   Max[1,Min[i,Round[Random[]*i]]],{i,1,n}];
 a=Table[1.,{i,1,p[[n+1]]}];
 Print["Mean Band=",N[p[[n+1]]/n]];
 S={p,a};
 Sr=SymmSkyMatrixLDUReconstruct[S];
 Print["Reconstructed SkyMatrix:"]; SymmSkyMatrixLowerTrianglePrint[Sr];
 SymmSkyMatrixLowerTriangleMap[Sr];
 x=Table[1.,{i,1,n}];
 b=SymmSkyMatrixVectorMultiply[Sr,x];
 Print["b=Ax=",b];*)
```

list. If the `i-th` entry of `x` is prescribed, the known value must be supplied in this vector.

The outputs are:

`x` The computed solution vector, stored as a single-level (one-dimensional) list. Prescribed solution components return the value of the entry in `b`.

Sometimes it is necessary to solve linear systems for multiple ($m > 1$) right hand sides. One way to do that is to call `SymmSkyMatrixVectorSolve` repeatedly. Alternatively, if m right hand sides are collected as columns of a rectangular matrix **B**, module `SymmSkyMatrixColBlockSolve` may be invoked as

```
X = SymmSkyMatrixVectorSolve[Sf,B]
```

to provide the solution **X** of $\mathbf{SX} = \mathbf{B}$. This module is listed in Cell 12.5. The input arguments and function returns have the same function as those described for `SymmSkyMatrixVectorSolve`. The main difference is that `B` and `X` are matrices (two-dimensional lists) with the righ-hand side and

Cell 12.7 Multiplying Skymatrix by Vector Block

```
SymmSkyMatrixColBlockMultiply[S_,x_]:= Module[
  {p,a,n,nrhs,i,j,k,m,r,ii,aij,b},
  {p,a}=S; n=Length[p]-1;
  If [n!=Dimensions[x][[1]], Print["Inconsistent matrix dimensions in",
    " SymmSkyMatrixColBlockMultiply"]; Return[Null]];
  nrhs = Dimensions[x][[2]]; b=Table[0,{n},{nrhs}];
  Do [ii=Abs[p[[i+1]]]; m=ii-Abs[p[[i]]]-1;
     Do [b[[i,r]]=a[[ii]]*x[[i,r]], {r,1,nrhs}];
     Do [j=i-k; aij=a[[ii-k]]; If [aij==0, Continue[]];
       Do [b[[i,r]]+=aij*x[[j,r]]; b[[j,r]]+=aij*x[[i,r]], {r,1,nrhs}],
     {k,1,m}],
  {i,1,n}];
  Return[b]
 ];

SymmSkyMatrixRowBlockMultiply[S_,x_]:= Module[
  {p,a,n,nrhs,i,j,k,m,r,ii,aij,b},
  {p,a}=S; n=Length[p]-1;
  If [n!=Dimensions[x][[2]], Print["Inconsistent matrix dimensions in",
    " SymmSkyMatrixRowBlockMultiply"]; Return[Null]];
  nrhs = Dimensions[x][[1]]; b=Table[0,{nrhs},{n}];
  Do [ii=Abs[p[[i+1]]]; m=ii-Abs[p[[i]]]-1;
     Do [b[[r,i]]=a[[ii]]*x[[r,i]], {r,1,nrhs}];
     Do [j=i-k; aij=a[[ii-k]]; If [aij==0, Continue[]];
       Do [b[[r,i]]+=aij*x[[r,j]]; b[[r,j]]+=aij*x[[r,i]], {r,1,nrhs}],
     {k,1,m}],
  {i,1,n}];
  Return[b]
 ];
```

solution vectors as columns. There is a similar module `SymmSkyMatrixRowBlockSolve`, notlisted here, which solves for multiple right hand sides stored as rows of a matrix.

§12.3.4 Matrix-Vector Multiply

For various applications it is necessary to form the matrix-vector product

$$\mathbf{b} = \mathbf{S}\mathbf{x} \tag{12.17}$$

where **S** is a symmetric skymatrix and **x** is given.

This is done by module `SymmSkyMatrixVectorMultiply`, which is listed in Cell 12.6. Its arguments are the skymatrix `S` and the vector `x`. The function returns **Sx** in `b`.

Module `SymmSkyMatrixColBlockMultiply` implements the multiplication by a block of vectors

stored as columns of a rectangular matrix **X**:

$$\mathbf{B} = \mathbf{S}\mathbf{X} \tag{12.18}$$

This module is listed in Cell 12.7. Its arguments are the skymatrix `S` and the rectangular matrix `X`. The function returns **SX** in `B`.

There is a similar module `SymmSkyMatrixRowBlockMultiply`, also listed in Cell 12.7, which postmultiplies a vector block stored as rows.

§12.4 PRINTING AND MAPPING

Module `SymmSkyMatrixUpperTrianglePrint`, listed in Cell 12.8, prints a symmetric skymatrix in upper triangle form. Is is invoked as

```
SymmSkyMatrixUpperTrianglePrint[S]
```

where `S` is the skymatrix to be printed. For an example of use see Cells 122-3.

The print format resembles the configuration depicted in Section 12.1. This kind of print is useful for program development and debugging although of course it should not be attempted with a very large matrix.[1]

There is a similar module called `SymmSkyMatrixLowerTrianglePrint`, which displays the skymatrix entries in lower triangular form. This module is also listed in Cell 12.8.

Sometimes one is not interested in the actual values of the skymatrix entries but only on how the skyline template looks like. Such displays, called *maps*, can be done with just one symbol per entry. Module `SymmSkyMatrixUpperTriangleMap`, listed in Cell 12.9, produces a map of its argument. It is invoked as

```
SymmSkyMatrixUpperTriangleMap[S]
```

The entries within the skyline template are displayed by symbols `+`, `-` and `0`, depending on whether the value is positive, negative or zero, respectively. Entries outside the skyline template are blank.

As in the case of the print module, there is module `SymmSkyMatrixLowerTriangleMap` which is also listed in Cell 12.9.

---

[1] Computer oriented readers may notice that the code for the printing routine is substantially more complex than those of the computational modules. This is primarily due to the inadequacies of `Mathematica` in handling tabular format output. The corresponding Fortran or C implementations would be simpler because those languages provide much better control over low-level display.

Cell 12.8 Skymatrix Printing

```
SymmSkyMatrixLowerTrianglePrint[S_]:= Module[
 {p,a,cycle,i,ii,ij,it,j,jj,j1,j2,jref,jbeg,jend,jt,kcmax,kc,kr,m,n,c,t},
 {p,a}=S; n=Dimensions[p][[1]]-1; kcmax=5; jref=0;
 Label[cycle]; Print[" "];
   jbeg=jref+1; jend=Min[jref+kcmax,n]; kc=jend-jref;
   t=Table[" ",{n-jref+1},{kc+1}];
   Do [If [p[[j+1]]>0,c=" ",c="*"];
      t[[1,j-jref+1]]=StringJoin[c,"Col",ToString[PaddedForm[j,3]]],
     {j,jbeg,jend}];  it=1;
   Do [ii=Abs[p[[i+1]]]; m=ii-Abs[p[[i]]]-1; j1=Max[i-m,jbeg];j2=Min[i,jend];
      kr=j2-j1+1; If [kr<=0, Continue[]]; If [p[[i+1]]>0,c=" ",c="*"];
      it++; t[[it,1]]=StringJoin[c,"Row",ToString[PaddedForm[i,3]]];
      jt=j1-jbeg+2; ij=j1+ii-i;
      Do[t[[it,jt++]]=PaddedForm[a[[ij++]]//FortranForm,{7,4}],{j,1,kr}],
    {i,jbeg,n}];
   Print[TableForm[Take[t,it],TableAlignments->{Right,Right},
         TableDirections->{Column,Row},TableSpacing->{0,2}]];
   jref=jend; If[jref<n,Goto[cycle]];
];
SymmSkyMatrixUpperTrianglePrint[S_]:= Module[
 {p,a,cycle,i,ij,it,j,j1,j2,jref,jbeg,jend,kcmax,k,kc,m,n,c,t},
 {p,a}=S; n=Dimensions[p][[1]]-1; kcmax=5; jref=0;
  Label[cycle]; Print[" "];
   jbeg=jref+1; jend=Min[jref+kcmax,n]; kc=jend-jref;
   t=Table[" ",{jend+1},{kc+1}];
   Do [If [p[[j+1]]>0,c=" ",c="*"];
      t[[1,j-jref+1]]=StringJoin[c,"Col",ToString[PaddedForm[j,3]]],
   {j,jbeg,jend}];  it=1;
   Do [it++;  If [p[[i+1]]>0,c=" ",c="*"];
       t[[it,1]]=StringJoin[c,"Row",ToString[PaddedForm[i,3]]]; j=jref;
       Do [j++; If [j<i, Continue[]]; ij=Abs[p[[j+1]]]+i-j;
          If [ij<=Abs[p[[j]]], Continue[]];
          t[[it,k+1]]=PaddedForm[a[[ij]]//FortranForm,{7,4}],
       {k,1,kc}],
   {i,1,jend}];
   Print[TableForm[Take[t,it],TableAlignments->{Right,Right},
         TableDirections->{Column,Row},TableSpacing->{0,2}]];
   jref=jend; If[jref<n,Goto[cycle]];
];

Sr={{0, 1, 3, 6}, {1., 2., 7., 4., 23., 97.}};
SymmSkyMatrixLowerTrianglePrint[Sr];SymmSkyMatrixUpperTrianglePrint[Sr];
```

Cell 12.9 Skymatrix Mapping

```
SymmSkyMatrixLowerTriangleMap[S_]:=Module[
 {p,a,cycle,i,ii,ij,it,itop,j,jj,j1,j2,jref,jbeg,jend,jt,kcmax,kc,kr,m,n,c,t},
 {p,a}=S; n=Dimensions[p][[1]]-1; kcmax=40; jref=0;
 Label[cycle]; Print[" "];
   jbeg=jref+1; jend=Min[jref+kcmax,n]; kc=jend-jref;
   itop=2; If[jend>9,itop=3]; If[jend>99,itop=4]; If[jend>999,itop=5];
   t=Table[" ",{n-jref+itop},{kc+1}]; it=0;
   If [itop>=5, it++; Do [m=Floor[j/1000];
      If[m>0,t[[it,j-jref+1]]=ToString[Mod[m,10]]], {j,jbeg,jend}]];
   If [itop>=4, it++; Do [m=Floor[j/100];
      If[m>0,t[[it,j-jref+1]]=ToString[Mod[m,10]]], {j,jbeg,jend}]];
   If [itop>=3, it++; Do [m=Floor[j/10];
      If[m>0,t[[it,j-jref+1]]=ToString[Mod[m,10]]], {j,jbeg,jend}]];
   it++; Do[t[[it,j-jref+1]]=ToString[Mod[j,10]],{j,jbeg,jend}];
   it++; Do[If[p[[j+1]]<0,t[[it,j-jref+1]]="*"],{j,jbeg,jend}];
   Do [ii=Abs[p[[i+1]]]; m=ii-Abs[p[[i]]]-1; j1=Max[i-m,jbeg];j2=Min[i,jend];
      kr=j2-j1+1; If [kr<=0, Continue[]]; If [p[[i+1]]>0,c=" ",c="*"];
      it++; t[[it,1]]=StringJoin[ToString[PaddedForm[i,2]],c];
      jt=j1-jbeg+2; ij=j1+ii-i;
      Do [ c=" 0"; If[a[[ij]]>0,c=" +"]; If[a[[ij++]]<0,c=" -"];
          t[[it,jt++]]=c, {j,1,kr}],
      {i,jbeg,n}];
   Print[TableForm[Take[t,it],TableAlignments->{Right,Right},
         TableDirections->{Column,Row},TableSpacing->{0,0}]];
   jref=jend; If[jref<n,Goto[cycle]];
];
SymmSkyMatrixUpperTriangleMap[S_]:=Module[
 {p,a,cycle,i,ij,it,itop,j,j1,j2,jref,jbeg,jend,kcmax,k,kc,m,n,c,t},
 {p,a}=S; n=Dimensions[p][[1]]-1; kcmax=40; jref=0;
 Label[cycle]; Print[" "];
   jbeg=jref+1; jend=Min[jref+kcmax,n]; kc=jend-jref;
   itop=2; If[jend>9,itop=3]; If[jend>99,itop=4]; If[jend>999,itop=5];
   t=Table[" ",{jend+itop},{kc+1}]; it=0;
   If [itop>=5, it++; Do [m=Floor[j/1000];
      If[m>0,t[[it,j-jref+1]]=ToString[Mod[m,10]]], {j,jbeg,jend}]];
   If [itop>=4, it++; Do [m=Floor[j/100];
      If[m>0,t[[it,j-jref+1]]=ToString[Mod[m,10]]], {j,jbeg,jend}]];
   If [itop>=3, it++; Do [m=Floor[j/10];
      If[m>0,t[[it,j-jref+1]]=ToString[Mod[m,10]]], {j,jbeg,jend}]];
   it++; Do[t[[it,j-jref+1]]=ToString[Mod[j,10]],{j,jbeg,jend}];
   it++; Do[If[p[[j+1]]<0,t[[it,j-jref+1]]="*"],{j,jbeg,jend}];
   Do [it++;  If [p[[i+1]]>0,c=" ",c="*"];
       t[[it,1]]=StringJoin[ToString[PaddedForm[i,2]],c]; j=jref;
       Do [j++; If [j<i, Continue[]]; ij=Abs[p[[j+1]]]+i-j;
          If [ij<=Abs[p[[j]]], Continue[]]; c=" 0";
          If[a[[ij]]>0,c=" +"]; If[a[[ij++]]<0,c=" -"]; t[[it,k+1]]=c,
         {k,1,kc}],
   {i,1,jend}];
   Print[TableForm[Take[t,it],TableAlignments->{Right,Right},
          TableDirections->{Column,Row},TableSpacing->{0,0}]];
   jref=jend; If[jref<n,Goto[cycle]];
];
```

12–16

```
ClearAll[S,Sr];
S={{0,1,3,6,10,11,17,-20,22,30,-36},Table[1000.*i,{i,1,36}]};
SymmSkyMatrixLowerTriangleMap[S]; SymmSkyMatrixUpperTriangleMap[S];
```

Cell 12.10 Skymatrix Reconstruction from Factors

```
SymmSkyMatrixLDUReconstruct[S_]:= Module[
  {p,ldu,a,v,n,i,ii,ij,j,jj,jk,jmj,k,m},
  {p,ldu}=S; a=ldu; n=Length[p]-1; v=Table[0,{n}];
  Do [jmj=Abs[p[[j]]]; jj=p[[j+1]]; If [jj<0, Continue[]];
      jk=jj-jmj; v[[jk]]=ldu[[jj]];
     Do [ij=jmj+k; i=j+ij-jj; ii=p[[i+1]]; If [ii<0, v[[k]]=0; Continue[]];
         If [i!=j, v[[k]]=ldu[[ij]]*ldu[[ii]]];
         m=Min[ii-Abs[p[[i]]],k]; a[[ij]]= v[[k]];
         a[[ij]]+=Take[ldu,{ii-m+1,ii-1}].Take[v,{k-m+1,k-1}],
        {k,1,jk}],
     {j,1,n}];  Return[{p,a}];
  ];

SymmSkyMatrixLDinvUReconstruct[S_]:= Module[
  {p,ldu,a,v,n,i,ii,ij,j,jj,jk,jmj,k,m},
  {p,ldu}=S; a=ldu; n=Length[p]-1; v=Table[0,{n}];
  Do [jmj=Abs[p[[j]]]; jj=p[[j+1]]; If [jj<0, Continue[]];
      jk=jj-jmj; v[[jk]]=1/ldu[[jj]];
     Do [ij=jmj+k; i=j+ij-jj; ii=p[[i+1]]; If [ii<0, v[[k]]=0; Continue[]];
         If [i!=j, v[[k]]=ldu[[ij]]/ldu[[ii]]];
         m=Min[ii-Abs[p[[i]]],k]; a[[ij]]= v[[k]];
         a[[ij]]+=Take[ldu,{ii-m+1,ii-1}].Take[v,{k-m+1,k-1}],
        {k,1,jk}],
     {j,1,n}];  Return[{p,a}];
  ];

 p={0,1,2,5,8,9,15}; s={11,22,13,0,33,24,34,44,55,16,0,0,46,56,66};
 S={p,s};
Sr=SymmSkyMatrixLDinvUReconstruct[S]; Print[Sr//InputForm];
Print[SymmSkyMatrixFactor[Sr,0]];
```

§12.4.1 Reconstruction of SkyMatrix from Factors

In testing factorization and solving modules it is convenient to have modules that perform the "inverse process" of the factorization. More specifically, suppose that $\mathbf{U}$, $\mathbf{D}$ (or $\mathbf{D}^{-1}$) are given, and the problem is to reconstruct the skymatrix that have them as factors:

$$\mathbf{S} = \mathbf{LDU}, \qquad \text{or} \quad \mathbf{S} = \mathbf{LD}^{-1}\mathbf{U} \tag{12.19}$$

in which $\mathbf{L} = \mathbf{U}^T$. Modules `SymmSkyMatrixLDUReconstruction` and `SymmSkyMatrixLDinvUReconstruction` perform those operations. These modules are listed in Cell 12.10. Their argument is a factored form of $\mathbf{S}$: $\mathbf{U}$ and $\mathbf{D}$ in the first case, and $\mathbf{U}$ and $\mathbf{D}^{-1}$ in the second case.

Cell 12.11 Miscellaneous Skymatrix Utilities

```
SymmSkyMatrixConvertToFull[S_]:= Module[
  {p,a,aa,n,j,jj,jmj,k},
  {p,a}=S; n=Length[p]-1; aa=Table[0,{n},{n}];
  Do [jmj=Abs[p[[j]]]; jj=Abs[p[[j+1]]]; aa[[j,j]]=a[[jj]];
     Do [aa[[j,j-k]]=aa[[j-k,j]]=a[[jj-k]],{k,1,jj-jmj-1}],
  {j,1,n}];  Return[aa];
  ];

SymmSkyMatrixConvertUnitUpperTriangleToFull[S_]:= Module[
  {p,ldu,aa,n,j,jj,jmj,k},
  {p,ldu}=S; n=Length[p]-1; aa=Table[0,{n},{n}];
  Do [jmj=Abs[p[[j]]]; jj=Abs[p[[j+1]]]; aa[[j,j]]=1;
     Do [aa[[j-k,j]]=ldu[[jj-k]],{k,1,jj-jmj-1}],
  {j,1,n}];  Return[aa];
  ];

SymmSkyMatrixConvertDiagonalToFull[S_]:= Module[
  {p,ldu,aa,n,i,j,jj,jmj,k},
  {p,ldu}=S; n=Length[p]-1; aa=Table[0,{n},{n}];
  Do [jj=Abs[p[[j+1]]]; aa[[j,j]]=ldu[[jj]],
  {j,1,n}];  Return[aa];
  ];
```

§12.4.2 Miscellaneous Utilities

Finally, Cell 12.11 lists three miscellaneous modules. The most useful one is probably `SymmSkyMatrixConvertToFull`, which converts its skymatrix argument to a fully stored symmetric matrix. This is useful for things like a quick and dirty computation of eigenvalues:

```
Print[Eigenvalues[SymmSkyMatrixConvertToFull[S]]];
```

because *Mathematica* built-in eigensolvers require that the matrix be supplied in full storage form.

-

13

Skyline Assembly

§13.1 INTRODUCTION

The previous two Chapters explained preparatory operations for skyline solution, and the solution process itself. Sandwiched between preparation and solution there is the *assembly* process, in which the master stiffness equations are constructed.

This Chapter presents details of the assembly process of the master stiffness matrix **K** using the skyline format and describes the *MathFET* implementation in *Mathematica*.

§13.1.1 Recapitulation of Main DSM Steps

Figure 13.1 summarizes the basic steps of the Direct Stiffness Method using a skyline direct solver for the master stiffness equations. Chapters 12 and 13 dealt with the skyline preprocessing stage and the solution, respectively. This Chapter deals with the matrix assembly process.

Assembly is the most complex stage of a finite element program in terms of the data flow. The complexity is not due to the assembly per se, but to the combined effect of formation of individual elements and merge.. Forming an element requires access to node, element, constitutive and fabrication data. Merge requires access to the freedom and connection data. We see that this stage uses up all of the information studied in Chapters 2 through 12. Thus its coverage here.

The central role of the assembler is displayed in Figure 13.1.

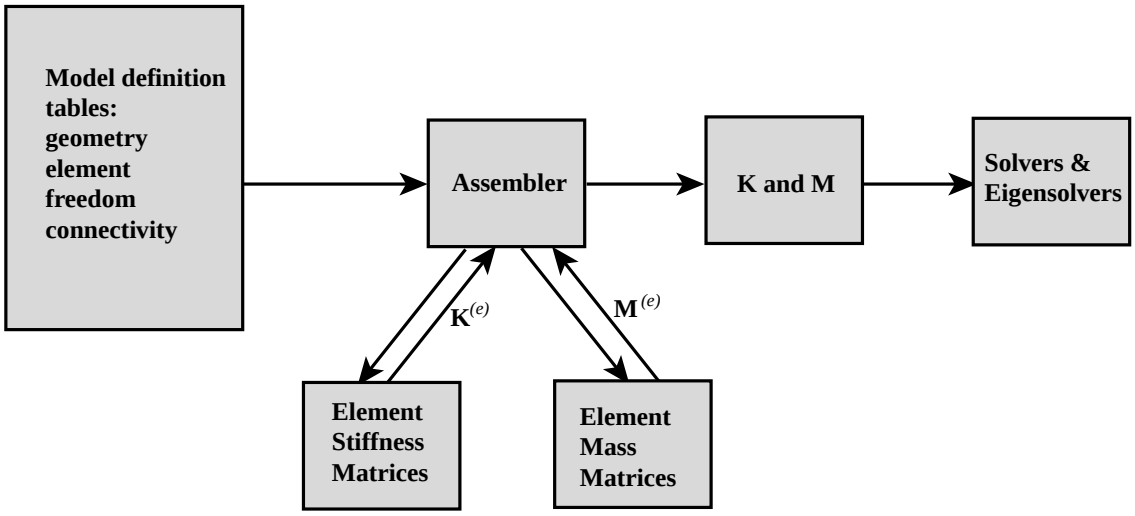
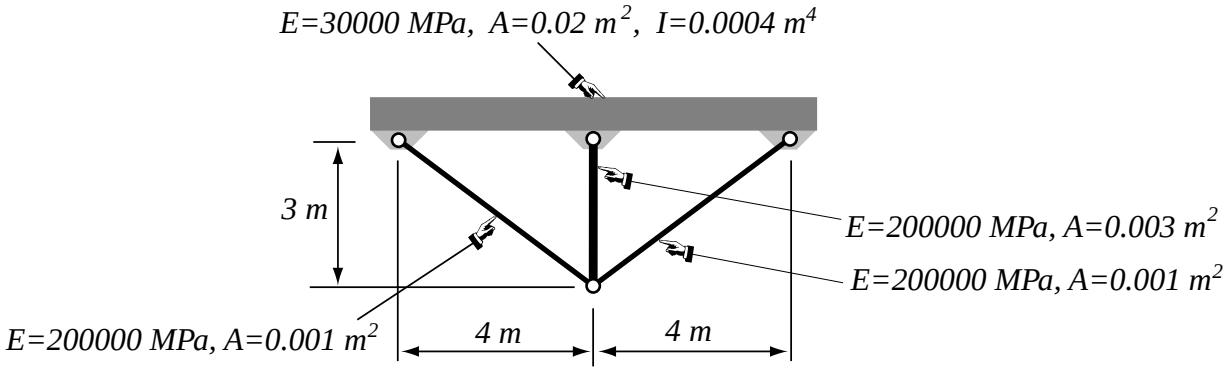


Figure 13.1. Role of assembler in FEM program.



SI Physical Units: m, MPa, s, ºC

Figure 13.2. Sample structure to illustrate assembly process.

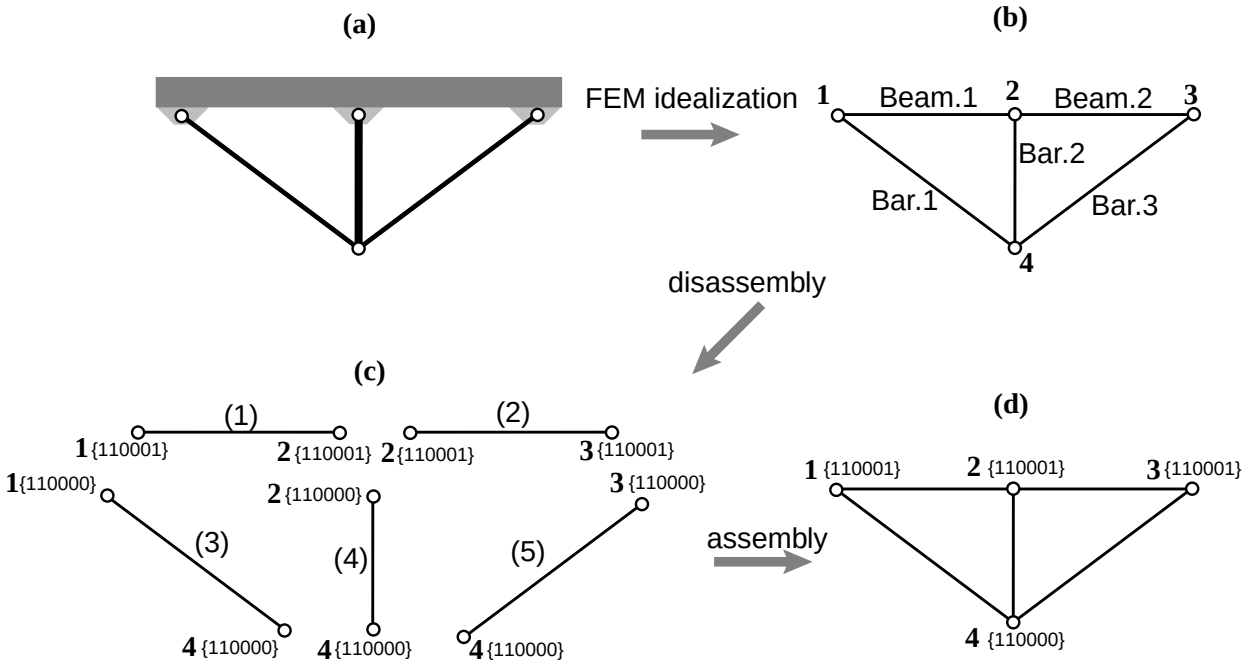


Figure 13.3. Finite element discretization, disassembly and assembly of example structure.

§13.2 STRUCTURAL ASSEMBLY EXAMPLE

We illustrate the assembly process with the structure shown in Figure 13.2. The finite element discretization, disassembly into elements and assembly are illustrated in Figure 13.3(a,b,c,d).

The assembled structure has 11 degrees of freedom, 3 at nodes 1, 2 and 3, and 2 at node 3. They are ordered

as follows:

GDOF #:	1	2	3	4	5	6	7	8	9	10	11
DOF:	u_{x1}	u_{y1}	θ_{z1}	u_{x2}	u_{y2}	θ_{z2}	u_{x3}	u_{y3}	θ_{z3}	u_{x4}	u_{y4}

(13.1)

The position of each freedom, as identified by the numbers in the first row, is called the *global freedom number*.

We now proceed to go over the assembly process by hand. The master stiffness $\mathbf{K}$ will be displayed as a fully stored matrix for visualization convenience. The process beging by initialization of $\mathbf{K} = \mathbf{0}$.

Element (1), "Beam.1". This is a plane beam-column element with freedoms

1	2	3	4	5	6
u_{x1}	u_{y1}	θ_{z1}	u_{x2}	u_{y2}	θ_{z2}

(13.2)

and its stiffness matrix is

$$\mathbf{K}^{(1)} = \begin{bmatrix} 150. & 0. & 0. & -150. & 0. & 0. \\ 0. & 22.5 & 45. & 0. & -22.5 & 45. \\ 0. & 45. & 120. & 0. & -45. & 60. \\ -150. & 0. & 0. & 150. & 0. & 0. \\ 0. & -22.5 & -45. & 0. & 22.5 & -45. \\ 0. & 45. & 60. & 0. & -45. & 120. \end{bmatrix} \tag{13.3}$$

The mapping between the element and global freedoms is specified by the Element Freedom Mapping Table or `eFMT`:

element:	1	2	3	4	5	6
assembly:	1	2	3	4	5	6

(13.4)

Upon merging this element the master stiffness matrix becomes

$$\mathbf{K} = \begin{bmatrix} 150. & 0. & 0. & -150. & 0. & 0. & 0 & 0 & 0 & 0 & 0 \\ 0. & 22.5 & 45. & 0. & -22.5 & 45. & 0 & 0 & 0 & 0 & 0 \\ 0. & 45. & 120. & 0. & -45. & 60. & 0 & 0 & 0 & 0 & 0 \\ -150. & 0. & 0. & 150. & 0. & 0. & 0 & 0 & 0 & 0 & 0 \\ 0. & -22.5 & -45. & 0. & 22.5 & -45. & 0 & 0 & 0 & 0 & 0 \\ 0. & 45. & 60. & 0. & -45. & 120. & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{13.5}$$

Element (2), "Beam.2" . This is a plane beam-column element with freedoms

1	2	3	4	5	6
u_{x2}	u_{y2}	θ_{z2}	u_{x3}	u_{y3}	θ_{z3}

(13.6)

The element stiffness matrix $\mathbf{K}^{(2)}$ is identical to (13.3). The mapping between the element and global freedoms is specified by the `eFMT`:

element:	1	2	3	4	5	6
assembly:	4	5	6	7	8	9

(13.7)

Upon merge the master stiffness matrix becomes

$$\mathbf{K} = \begin{bmatrix} 150. & 0. & 0. & -150. & 0. & 0. & 0 & 0 & 0 & 0 & 0 \\ 0. & 22.5 & 45. & 0. & -22.5 & 45. & 0 & 0 & 0 & 0 & 0 \\ 0. & 45. & 120. & 0. & -45. & 60. & 0 & 0 & 0 & 0 & 0 \\ -150. & 0. & 0. & 300. & 0. & 0. & -150. & 0. & 0. & 0 & 0 \\ 0. & -22.5 & -45. & 0. & 45. & 0. & 0. & -22.5 & 45. & 0 & 0 \\ 0. & 45. & 60. & 0. & 0. & 240. & 0. & -45. & 60. & 0 & 0 \\ 0 & 0 & 0 & -150. & 0. & 0. & 150. & 0. & 0. & 0 & 0 \\ 0 & 0 & 0 & 0. & -22.5 & -45. & 0. & 22.5 & -45. & 0 & 0 \\ 0 & 0 & 0 & 0. & 45. & 60. & 0. & -45. & 120. & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{13.8}$$

Element (3), "Bar.1". This is a plane bar element with freedoms

1	2	3	4
u_{x1}	u_{y1}	u_{x4}	u_{y4}

(13.9)

The element stiffness matrix is

$$\mathbf{K}^{(3)} = \begin{bmatrix} 25.6 & 19.2 & -25.6 & -19.2 \\ 19.2 & 14.4 & -19.2 & -14.4 \\ -25.6 & -19.2 & 25.6 & 19.2 \\ -19.2 & -14.4 & 19.2 & 14.4 \end{bmatrix} \tag{13.10}$$

The mapping between the element and global freedoms is specified by the `eFMT`:

element:	1	2	3	4
assembly:	1	2	10	11

(13.11)

Upon merge the master stiffness matrix becomes

$$\mathbf{K} = \begin{bmatrix} 175.6 & 19.2 & 0. & -150. & 0. & 0. & 0 & 0 & 0 & -25.6 & -19.2 \\ 19.2 & 36.9 & 45. & 0. & -22.5 & 45. & 0 & 0 & 0 & -19.2 & -14.4 \\ 0. & 45. & 120. & 0. & -45. & 60. & 0 & 0 & 0 & 0 & 0 \\ -150. & 0. & 0. & 300. & 0. & 0. & -150. & 0. & 0. & 0 & 0 \\ 0. & -22.5 & -45. & 0. & 45. & 0. & 0. & -22.5 & 45. & 0 & 0 \\ 0. & 45. & 60. & 0. & 0. & 240. & 0. & -45. & 60. & 0 & 0 \\ 0 & 0 & 0 & -150. & 0. & 0. & 150. & 0. & 0. & 0 & 0 \\ 0 & 0 & 0 & 0. & -22.5 & -45. & 0. & 22.5 & -45. & 0 & 0 \\ 0 & 0 & 0 & 0. & 45. & 60. & 0. & -45. & 120. & 0 & 0 \\ -25.6 & -19.2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 25.6 & 19.2 \\ -19.2 & -14.4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 19.2 & 14.4 \end{bmatrix} \tag{13.12}$$

Element (4), "Bar.2". This is a plane bar element with freedoms

1	2	3	4
u_{x2}	u_{y2}	u_{x4}	u_{y4}

(13.13)

The element stiffness matrix is

$$\mathbf{K}^{(4)} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 200. & 0 & -200. \\ 0 & 0 & 0 & 0 \\ 0 & -200. & 0 & 200. \end{bmatrix} \tag{13.14}$$

The mapping between the element and global freedoms is specified by the `eFMT`:

element:	1	2	3	4
assembly:	4	5	10	11

(13.15)

Upon merge the master stiffness matrix becomes

$$\mathbf{K} = \begin{bmatrix} 175.6 & 19.2 & 0. & -150. & 0. & 0. & 0 & 0 & 0 & -25.6 & -19.2 \\ 19.2 & 36.9 & 45. & 0. & -22.5 & 45. & 0 & 0 & 0 & -19.2 & -14.4 \\ 0. & 45. & 120. & 0. & -45. & 60. & 0 & 0 & 0 & 0 & 0 \\ -150. & 0. & 0. & 300. & 0. & 0. & -150. & 0. & 0. & 0 & 0 \\ 0. & -22.5 & -45. & 0. & 245. & 0. & 0. & -22.5 & 45. & 0 & -200. \\ 0. & 45. & 60. & 0. & 0. & 240. & 0. & -45. & 60. & 0 & 0 \\ 0 & 0 & 0 & -150. & 0. & 0. & 150. & 0. & 0 & 0 & 0 \\ 0 & 0 & 0 & 0. & -22.5 & -45. & 0. & 22.5 & -45. & 0 & 0 \\ 0 & 0 & 0 & 0. & 45. & 60. & 0. & -45. & 120. & 0 & 0 \\ -25.6 & -19.2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 25.6 & 19.2 \\ -19.2 & -14.4 & 0 & 0 & -200. & 0 & 0 & 0 & 0 & 19.2 & 214.4 \end{bmatrix} \tag{13.16}$$

Element (5), "Bar.3". This is a plane bar element with freedoms

1	2	3	4
u_{x2}	u_{y2}	u_{x4}	u_{y4}

(13.17)

The element stiffness matrix is

$$\mathbf{K}^{(4)} = \begin{bmatrix} 25.6 & -19.2 & -25.6 & 19.2 \\ -19.2 & 14.4 & 19.2 & -14.4 \\ -25.6 & 19.2 & 25.6 & -19.2 \\ 19.2 & -14.4 & -19.2 & 14.4 \end{bmatrix} \tag{13.18}$$

The mapping between the element and global freedoms is specified by the `eFMT`:

element:	1	2	3	4
assembly:	7	8	10	11

(13.19)

Upon merge the master stiffness matrix becomes

$$\mathbf{K} = \begin{bmatrix} 175.6 & 19.2 & 0. & -150. & 0. & 0. & 0 & 0 & 0 & -25.6 & -19.2 \\ 19.2 & 36.9 & 45. & 0. & -22.5 & 45. & 0 & 0 & 0 & -19.2 & -14.4 \\ 0. & 45. & 120. & 0. & -45. & 60. & 0 & 0 & 0 & 0 & 0 \\ -150. & 0. & 0. & 300. & 0. & 0. & -150. & 0. & 0. & 0 & 0 \\ 0. & -22.5 & -45. & 0. & 245. & 0. & 0. & -22.5 & 45. & 0 & -200. \\ 0. & 45. & 60. & 0. & 0. & 240. & 0. & -45. & 60. & 0 & 0 \\ 0 & 0 & 0 & -150. & 0. & 0. & 175.6 & -19.2 & 0. & -25.6 & 19.2 \\ 0 & 0 & 0 & 0. & -22.5 & -45. & -19.2 & 36.9 & -45. & 19.2 & -14.4 \\ 0 & 0 & 0 & 0. & 45. & 60. & 0. & -45. & 120. & 0 & 0 \\ -25.6 & -19.2 & 0 & 0 & 0 & 0 & -25.6 & 19.2 & 0 & 51.2 & 0. \\ -19.2 & -14.4 & 0 & 0 & -200. & 0 & 19.2 & -14.4 & 0 & 0. & 228.8 \end{bmatrix} \tag{13.20}$$

For storage as a skyline matrix the connection between nodes 1 and 3 would be out of the skyline envelope. Omitting the lower triangle the skyline template looks like

$$\mathbf{K} = \begin{bmatrix} 175.6 & 19.2 & 0. & -150. & 0. & 0. & & & & -25.6 & -19.2 \\ & 36.9 & 45. & 0. & -22.5 & 45. & & & & -19.2 & -14.4 \\ & & 120. & 0. & -45. & 60. & & & & 0 & 0 \\ & & & 300. & 0. & 0. & -150. & 0. & 0. & 0 & 0 \\ & & & & 245. & 0. & 0. & -22.5 & 45. & 0 & -200. \\ & & & & & 240. & 0. & -45. & 60. & 0 & 0 \\ & & & & & & 175.6 & -19.2 & 0. & -25.6 & 19.2 \\ & & & & & & & 36.9 & -45. & 19.2 & -14.4 \\ & & & & & & & & 120. & 0 & 0 \\ & & & & & & & & & 51.2 & 0. \\ \textit{symm} & & & & & & & & & & 228.8 \end{bmatrix} \tag{13.21}$$

The diagonal location of (13.21) are defined by the table

$$\texttt{DLT = \{ 0,1,3,6,10,15,21,25,30,36,46,57 \}} \tag{13.22}$$

Examination of the skyline template (13.21) reveals that some additional zero entries could be removed from the template; for example K_{13}. The fact that those entries are exactly zero is, however, fortuitous. It comes from the fact that some elements such as the beams are aligned along x, which decouples axial and bending stiffnesses.

§13.2.1 Imposing a MultiFreedom Constraint

To see the effect of imposing an MFC on the configuration of the master stiffness matrix, suppose that the example structure is subjected to the constraint that nodes 1 and 3 must move vertically by the same amount. That is,

$$u_{y1} = u_{y3} \qquad \text{or} \qquad u_{y1} - u_{y3} = 0. \tag{13.23}$$

The constraint (13.23) is defined as a sixth element, named `"MFC.1"`, and applied by the Lagrange Multiplier method. The degrees of freedom of the assembled structure increase by one to 12, which are ordered as follows:

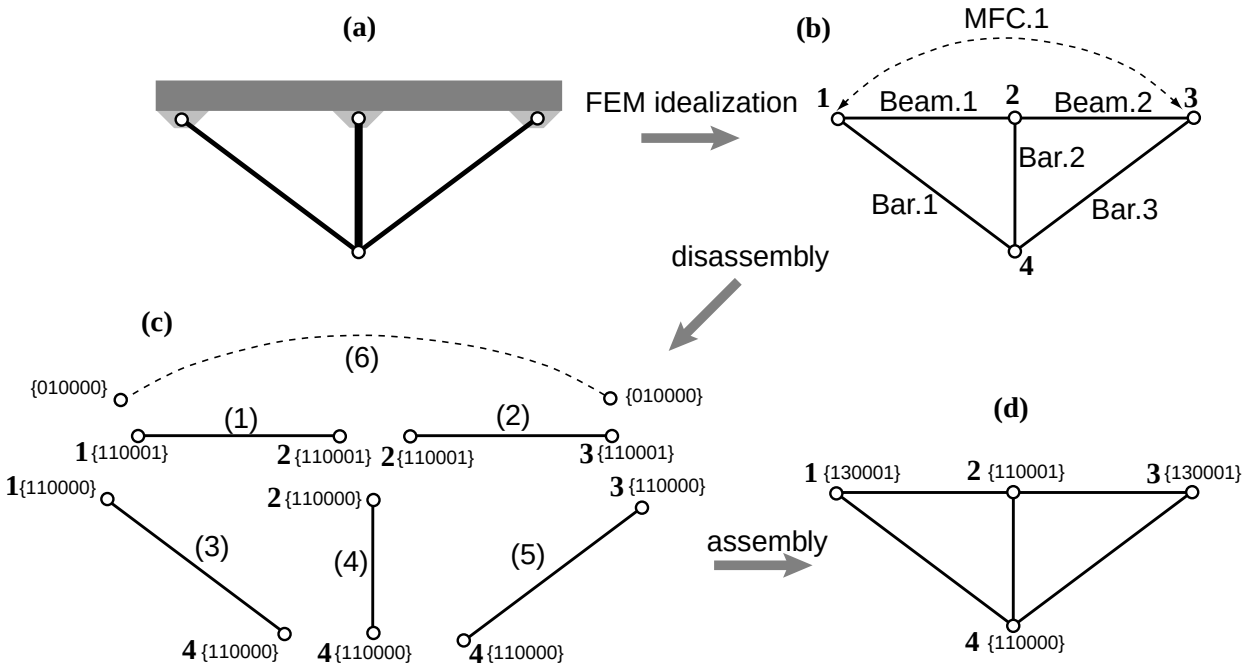


Figure 13.4. Finite element discretization, disassembly and assembly of example structure with MFC $u_{y1} = u_{y3}$.

GDOF #:	1	2	3	4	5	6	7	8	9	10	11	12
DOF:	u_{x1}	u_{y1}	θ_{z1}	u_{x2}	u_{y2}	θ_{z2}	u_{x3}	u_{y3}	θ_{z3}	u_{x4}	u_{y4}	$\lambda^{(6)}$

(13.24)

The assembly of the first five element proceeds as before, and produces the same master stiffness as (13.20), except for an extra zero row and column. Processing the sixth element yields

$$\mathbf{K} = \begin{bmatrix} 175.6 & 19.2 & 0. & -150. & 0. & 0. & 0 & 0 & 0 & -25.6 & -19.2 & 0 \\ 19.2 & 36.9 & 45. & 0. & -22.5 & 45. & 0 & 0 & 0 & -19.2 & -14.4 & 1. \\ 0. & 45. & 120. & 0. & -45. & 60. & 0 & 0 & 0 & 0 & 0 & 0 \\ -150. & 0. & 0. & 300. & 0. & 0. & -150. & 0. & 0. & 0 & 0 & 0 \\ 0. & -22.5 & -45. & 0. & 245. & 0. & 0. & -22.5 & 45. & 0 & -200. & 0 \\ 0. & 45. & 60. & 0. & 0. & 240. & 0. & -45. & 60. & 0 & 0 & 0 \\ 0 & 0 & 0 & -150. & 0. & 0. & 175.6 & -19.2 & 0. & -25.6 & 19.2 & 0 \\ 0 & 0 & 0 & 0. & -22.5 & -45. & -19.2 & 36.9 & -45. & 19.2 & -14.4 & -1. \\ 0 & 0 & 0 & 0. & 45. & 60. & 0. & -45. & 120. & 0 & 0 & 0 \\ -25.6 & -19.2 & 0 & 0 & 0 & 0 & -25.6 & 19.2 & 0 & 51.2 & 0. & 0 \\ -19.2 & -14.4 & 0 & 0 & -200. & 0 & 19.2 & -14.4 & 0 & 0. & 228.8 & 0 \\ 0 & 1. & 0 & 0 & 0 & 0 & 0 & -1. & 0 & 0 & 0 & 0 \end{bmatrix} \tag{13.25}$$

in which the coefficients 1 and -1 associated with the MFC (13.23) end up in the last row and column. The

asembled stiffness (13.25) can be represented as the symmetric skymatrix:

$$\mathbf{K} = \begin{bmatrix} 175.6 & 19.2 & 0. & -150. & 0. & 0. & & & & -25.6 & -19.2 & \\ & 36.9 & 45. & 0. & -22.5 & 45. & & & & -19.2 & -14.4 & 1. \\ & & 120. & 0. & -45. & 60. & & & & 0 & 0 & 0 \\ & & & 300. & 0. & 0. & -150. & 0. & 0. & 0 & 0 & 0 \\ & & & & 245. & 0. & 0. & -22.5 & 45. & 0 & -200. & 0 \\ & & & & & 240. & 0. & -45. & 60. & 0 & 0 & 0 \\ & & & & & & 175.6 & -19.2 & 0. & -25.6 & 19.2 & 0 \\ & & & & & & & 36.9 & -45. & 19.2 & -14.4 & -1. \\ & & & & & & & & 120. & 0 & 0 & 0 \\ & & & & & & & & & 51.2 & 0. & 0 \\ & & & & & & & & & & 228.8 & 0 \\ \textit{symm} & & & & & & & & & & & 0 \end{bmatrix} \tag{13.26}$$

The diagonal locations of (13.26) are defined by the table

$$\texttt{DLT = \{ 0,1,3,6,10,15,21,25,30,36,46,57,68 \}} \tag{13.27}$$

Cell 13.1 Assembling the Master Stiffness Matrix

```
MakeSkylineStiffnessMatrix[MNDT_,MEDT_,MELT_,MCPT_,MFPT_,
   MNFT_,MMST_,GNFAT_,GSDLT_]:= Module[
  {a,c,cc,ccold,coef,dimeNL,dofNL,e,eCL,edofNL,eFL,eNFAL,eNL,
   ename,eqsNL,eTL,fc,fcold,i,ii,j,jj,k,m,mfc,mFCL,
   n,numdof,numele=Length[MEDT],numeNL,numeqs,
   numfab=Length[MFPT],nummat=Length[MCPT],nummul=Length[MMST],
   numnod=Length[MNDT],numrcc,numrfc,Nx2i,p,s,tc,tcold,xn,
   cNL,ef,eFMT,etag,etagold,neldof,nf,ntag,ntagold},

   (*  Initialization *)

   p=GSDLT; numeqs=Length[p]-1; numdof=numeqs-nummul;
   lensky=Abs[p[[numeqs+1]]]; s=Table[0,{lensky}];
   k=0; Do [k=Max[MNDT[[n,1]],k], {n,1,numnod}];
   Nx2i=Table[0,{k}]; Do [xn=MNDT[[n,1]]; Nx2i[[xn]]=n, {n,1,numnod}];

   numrcc=numrfc=0;
   Do [If [Length[MCPT[[cc]]]==0, Break[]]; numrcc++, {cc,1,nummat}];
   Do [If [Length[MFPT[[fc]]]==0, Break[]]; numrfc++, {fc,1,numfab}];
   tcold=ccold=fcold=mfc=0; eCL=eFL=eNL=eTL={};

   (*  Cycle over elements *)

   Do [

   (*  Gather element data structures  *)

      ename= MEDT[[e,1]];
      {etype,tc}= MEDT[[e,2]];
      If [tc==0, Continue[]];  (* do nothing element *)
      eNL=        MEDT[[e,3]];
      {cc,fc}=    MEDT[[e,4]]; (* NB: zero codes retrieve {} *)
      eNLf=  Flatten[eNL];
      If [tc!=tcold, eTL=MELT[[tc]];          tcold=tc];
      If [cc!=ccold, eCL=MCPT[[cc+numrcc+1]]; ccold=cc];
      If [fc!=fcold, eFL=MFPT[[fc+numrfc+1]]; fcold=fc];
      numeNL=Length[eNLf];
      xyzNL=dirNL=dofNL=edofNL=eNFAL=Table[0,{numeNL}];
      Do [xn=eNLf[[i]]; If [xn<=0,Continue[]];
          n=Nx2i[[xn]];
          xyzNL[[i]]=MNDT[[n,2]]; dirNL[[i]]=MNFT[[n,4]];
          dofNL[[i]]=MNFT[[n,2]]; eNFAL[[i]]=GNFAT[[n]],
      {i,1,numeNL}];
      deNL= {0,0,0,0};
      Do [deNL[[i]]=Length[eNL[[i]]],{i,1,Length[eNL]}]; k=0;
      Do [Do [edofNL[[++k]]=eTL[[8,i]],{j,1,deNL[[i]]}],{i,1,4}];
```

§13.3 IMPLEMENTATION IN MATHFET

Module `MakeSkylineStiffnessMatrix` assembles the master stiffness matrix of an arbitrary linear structure. The code is fairly lengthly compared with other modules, and is split listed in Cells 13.1 and 1a. The module begins by performing various initializations, including that of the skyline stored master stiffness array. Then it loops over the elements, gets element stiffnesses from `MakeSkylineStiffnessMatrix`, forms the `EFMT` and merges it into the master stiffness. MultiFreedom Constraints (MFC), which are treated as special elements

`MakeSkylineStiffnessMatrix` interfaces with element stiffness formation modules through `GetLinearElementStiffness`, which is listed in Cell 13.2.

The test statement for the example structure of Figures 13.2-3 are shown in Cell 3, and those for the example structure with an MFC in Cell 4.

For the example structure the element stiffness of the bar and beam column discussed in Chapter 15 are used. These are reproduced in Cell 5 for convenience.

The assembler uses certain utility modules to pack and unpack freedom tags, already covered in Chapter 7. These modules are listed in Cell 6. Also shown in this cell is a stand-alone module called `ElementFreedomMapTable`, which builds the `eFMT` that maps local to global freedoms. The logic of this module is intricate because it covers many possibilities, inclkuding

1. Different assigned freedoms per node
2. Element freedoms has no “receiver” global freedoms
3. Global freedoms have no contributing element freedom

A stand alone configuration for this module is convenient for testing. However, the logic of `ElementFreedomMapTable` has been actually inlined into the assembler module listed in Cells 13.1-1a for efficiency. (The inlined statements appear in Cell 1a.)

Cell 7 lists two utility skymatrix modules used by the test programs. for displaying **K** in skymatrix format, and for converting a skymatrix to a full matrix.

Finally, Cell 8 reproduces the `MELT` builder discussed in Chapter 14. This is used to built a 3-entry `MELT`, which is printed in `InputForm` and transported, via cut and paste, to the test programs of Cells 13.3 and chapno.4.

Cell 13.1a Assembling the Master Stiffness Matrix (continued)

```
   (*  MultiFreedom Constraint needs special processing *)

       If [etype=="MFC", mfc++; k=numdof+mfc;
         mFCL=eFL[[5]]; m=Length[mFCL]; ii=p[[k+1]];
         Print["mfc=",mfc," mFCL=",mFCL];
         Do [{xn,j,coef}=mFCL[[i]]; n=Nx2i[[xn]];
              a=GNFAT[[n]]+j; s[[ii-k+a]]=coef,
         {i,1,m}];
         Continue[]
       ];

   (*  Ordinary element: form element stiffness matrix  *)

      Ke=GetLinearElementStiffness[ename,etype,eTL,deNL,eNLf,
                 xyzNL,dirNL,eCL,eFL];

   (*  Form element to global addressing table:
       MakeElementFreedomTable inlined for efficiency *)

      c=neldof=etagold=ntagold=0; ef=nf={0,0,0,0,0,0};
      cNL=Table[0,{numeNL}];
      Do[etag=edofNL[[i]];
        If [etag!=etagold, c=CountAssignedNodeFreedoms[etag];
        etagold=etag]; cNL[[i]]=c; neldof=neldof+c,
      {i,numeNL}];
      eFMT=Table[0,{neldof}]; k=etagold=0;
      Do [c=cNL[[i]]; a=eNFAL[[i]]; etag=edofNL[[i]]; ntag=dofNL[[i]];
      If [etag==ntag, Do [eFMT[[++k]]=++a, {j,1,c}]; Continue[]];
      If [etag!=etagold, ef=UnpackNodeFreedomTags[etag]; etagold=etag];
      If [ntag!=ntagold, nf=UnpackNodeFreedomTags[ntag]; ntagold=ntag];
         Do [If [nf[[j]]!=0, a++];
             If [ef[[j]]!=0, k++; If [nf[[j]]!=0, eFMT[[k]]=a]],
         {j,1,6}],
      {i,1,numeNL}];
      Print["eFMT=",eFMT];

    (*  Merge into global master stiffness *)

      Do [jj=eFMT[[j]]; If[jj==0, Continue[]];
         k=Abs[p[[jj+1]]]-jj;
         Do [ii=eFMT[[i]]; If [ii==0 || jj<ii, Continue[]];
            s[[k+ii]]+=Ke[[i,j]],
         {i,1,neldof}],
      {j,1,neldof}],
   {e,1,numele}];
   Return[{p,s}];
];
```

Cell 13.2 Assembler Interface to Element Stiffness Modules

```
GetLinearElementStiffness[ename_,etype_,eTL_,deNL_,eNLf_,
      xyzNL_,dirNL_,eCL_,eFL_]:= Module[{Ke},

 (*Print["\nEnter GetLinearElementStiffness"];
  Print["ename=",ename];
  Print["etype=",etype];
  Print["descriptor eTL=",eTL];
  Print["deNL=",deNL];
  Print["flattened eNL=",eNLf];
  Print["xyzNL=",xyzNL];
  Print["dirNL=",dirNL];
  Print["constitution eCL=",eCL];
  Print["fabrication  fCL=",eFL];*)

  If [etype=="Bar2D.2",
      Ke=Stiffness2DBar[{Take[xyzNL[[1]],2],Take[xyzNL[[2]],2]},
            {eCL[[3,1]],eCL[[3,3]]},{eFL[[3,1]]} ];

  Print["Ke of ",ename," is ",Ke//TableForm];
      Return[Ke]
  ];
  If [etype=="BeamC2D.2",
      Ke=Stiffness2DBeamColumn[{Take[xyzNL[[1]],2],Take[xyzNL[[2]],2]},
                                 eCL[[3]],eFL[[3]]];
  Print["Ke of ",ename," is ",Ke//TableForm];
      Return[Ke]
  ];

  Print["*** GetLinearElementStiffness: unknown element type ",
        etype]; Abort[];
];
```

Cell 13.3 Assembling the Uncnstrained Example Structure of Figures 13.2-3

```
MNDT={{1,{0,0,0},{}}, {2,{4,0,0},{}}, {3,{8,0,0},{}},{4,{4,3,0},{}}};
MNFT={{1,110001,3,{}}, {2,110001,3,{}}, {3,110001,3,{}},{4,110000,2,{}}};
MEDT={{"Beam.1",{"BeamC2D.2",2}, {{1,2}},{1,1}},
      {"Beam.2",{"BeamC2D.2",2}, {{2,3}},{1,1}},
      {"Bar.1", {"Bar2D.2",1},   {{1,4}},{2,2}},
      {"Bar.2", {"Bar2D.2",1},   {{2,4}},{2,3}},
      {"Bar.3", {"Bar2D.2",1},   {{3,4}},{2,2}}};
MELT={{"Bar2D.2",   "STM", 1, "BAR",    "MOM", "SEG", 1000, {110000}},
      {"BeamC2D.2", "STM", 1, "PBEAC1", "MOM", "SEG", 1000, {110001}},
      {"MFC",       "STM", 0, "CON",    "EXT", "ARB", 1000, {0}}};
MCPT={ {},
       {"Concrete",  {"Isot.L.E.S","SI"},{ 30000,.15,2320.,.000010}},
       {"ASTM-Steel",{"Isot.L.E.S","SI"},{200000,.30,7860.,.000012}}
       };
MFPT={{},{"DeckBeam", {"A.Iz","SI"},{.02,.004}},
         {"BarDiags", {"A","SI"},   {.001,.0004}},
         {"BarStrut", {"A","SI"},   {.003,.0004}}};
MMST={};
GNFAT={0,3,6,9,11};
GSDLTX={0,1,3,6,10,15,21,25,30,36,46,57};
{p,s}=MakeSkylineStiffnessMatrix[MNDT,MEDT,MELT,MCPT,MFPT,
      MNFT,MMST,GNFAT,GSDLTX];
 Print["\ndiag locators=",p];
 Print["s=",s];
 SymmSkyMatrixUpperTrianglePrint[{p,s}];
 Print["\nEigenvalues of K=",Chop[Eigenvalues[
       SymmSkyMatrixConvertToFull[{p,s}]],10.^(-12)]];
 Print["\nFull K=",SymmSkyMatrixConvertToFull[{p,s}]];
```

Cell 13.4 Assembling the Constrained Example Structure of Figure 13.4

```
MNDT={{1,{0,0,0},{}}, {2,{4,0,0},{}}, {3,{8,0,0},{}},{4,{4,3,0},{}}};
MNFT={{1,110001,3,{}}, {2,110001,3,{}}, {3,110001,3,{}},{4,110000,2,{}}};
MEDT={{"Beam.1",{"BeamC2D.2",2}, {{1,2}},{1,1}},
      {"Beam.2",{"BeamC2D.2",2}, {{2,3}},{1,1}},
      {"Bar.1", {"Bar2D.2",  1}, {{1,4}},{2,2}},
      {"Bar.2", {"Bar2D.2",  1}, {{2,4}},{2,3}},
      {"Bar.3", {"Bar2D.2",  1}, {{3,4}},{2,2}},
      {"MFC.1", {"MFC",      3}, {{1,3}},{0,4}}};
MELT={{"Bar2D.2",   "STM", 1, "BAR",    "MOM", "SEG", 1000, {110000}},
      {"BeamC2D.2", "STM", 1, "PBEAC1", "MOM", "SEG", 1000, {110001}},
      {"MFC",       "STM", 0, "CON",    "EXT", "ARB", 1000, {0}}};
MCPT={ {},
       {"Concrete",  {"Isot.L.E.S","SI"},{ 30000,.15,2320.,.000010}},
       {"ASTM-Steel",{"Isot.L.E.S","SI"},{200000,.30,7860.,.000012}}};
MFPT={{},{"DeckBeam", {"A.Iz","SI"},{.02,.004}},
         {"BarDiags", {"A","SI"},   {.001,.0004}},
         {"BarStrut", {"A","SI"},   {.003,.0004}},
         {"ty1=ty3",  {"MFC","SI"}, {},{},{{1,2,1.0},{3,2,-1.0}}}};
MMST={{6,0,0}};
GNFAT={0,3,6,9,11,12};
GSDLTX={0,1,3,6,10,15,21,25,30,36,46,57,68};
{p,s}=MakeSkylineStiffnessMatrix[MNDT,MEDT,MELT,MCPT,MFPT,
      MNFT,MMST,GNFAT,GSDLTX];
 Print["\ndiag locators=",p];
 Print["s=",s];
 SymmSkyMatrixUpperTrianglePrint[{p,s}];
 Print["\nEigenvalues of K=",Chop[Eigenvalues[
       SymmSkyMatrixConvertToFull[{p,s}]],10.^(-12)]];
 Print["\nFull K=",SymmSkyMatrixConvertToFull[{p,s}]];
```

Cell 13.5 Bae and Beam Element Stiffness Modules Used in Assembler Test

```
Stiffness2DBar[{{x1_,y1_},{x2_,y2_}},{Em_,rho_},{A_}]:= Module[
   {dx,dy,len3,Ke}, dx=x2-x1; dy=y2-y1;
    len3=(dx^2+dy^2)*PowerExpand[Sqrt[dx^2+dy^2]];
    Ke=(Em*A/len3)*{{ dx^2,  dx*dy,-dx^2, -dx*dy},
                   { dx*dy, dy^2, -dy*dx,-dy^2},
                   {-dx^2, -dy*dx, dx^2,  dy*dx},
                   {-dy*dx,-dy^2,  dy*dx, dy^2}};
    Return[Ke]
 ];
Ke=Stiffness2DBar[{{0,0},{L,0}},{Em,rho},{A}];
Print[Ke//MatrixForm];

Stiffness2DBeamColumn[{{x1_,y1_},{x2_,y2_}},{Em_,nu_,rho_,alpha_},
    {A_,Iz_}]:= Module[{dx,dy,c,s,L,LL,ra,rb,Kebar,Ke},
    dx=x2-x1; dy=y2-y1; LL=dx^2+dy^2; L=PowerExpand[Sqrt[LL]];
    c=dx/L; s=dy/L; ra=Em*A/L; rb= Em*Iz/L^3;
    Kebar= ra*{
     { 1,0,0,-1,0,0},{0,0,0,0,0,0},{0,0,0,0,0,0},
     {-1,0,0, 1,0,0},{0,0,0,0,0,0},{0,0,0,0,0,0}} +
           rb*{
     { 0,0,0,0,0,0},{0, 12, 6*L,0,-12,6*L},{0,6*L,4*LL,0,-6*L,2*LL},
     { 0,0,0,0,0,0},{0,-12,-6*L,0,12,-6*L},{0,6*L,2*LL,0,-6*L,4*LL}};
     T={{c,s,0,0,0,0},{-s,c,0,0,0,0},{0,0,1,0,0,0},
        {0,0,0,c,s,0},{0,0,0,-s,c,0},{0,0,0,0,0,1}};
    Ke=Transpose[T].Kebar.T;
    Return[Ke]
 ];

 Ke=Stiffness2DBeamColumn[{{0,0},{0,L}},{Em,nu,rho,alpha},{A,Iz}];
 Print[Ke];
```

Cell 13.6 Utility Modules Used by Assembler

```
UnpackNodeFreedomTags[p_]:= Module[{f1,g2,f2,g3,f3,g4,f4,g5,f5,f6},
   If [p==0, Return[{0,0,0,0,0,0}]];
   f1=Floor[p/100000];
   g2=p -100000*f1;   f2=Floor[g2/10000]; g3=g2-10000*f2;
   f3=Floor[g3/1000]; g4=g3-1000*f3;      f4=Floor[g4/100];
   g5=g4-100*f4;      f5=Floor[g5/10];    f6=g5-10*f5;
   Return[{f1,f2,f3,f4,f5,f6}];
 ];
UnpackAssignedNodeFreedomTags[p_]:= Module[
  {f1,g2,f2,g3,f3,g4,f4,g5,f5,f6,f,fx,i,j},
   If [p==0, Return[{}]];
   f1=Floor[p/100000];
   g2=p -100000*f1;   f2=Floor[g2/10000]; g3=g2-10000*f2;
   f3=Floor[g3/1000]; g4=g3-1000*f3;      f4=Floor[g4/100];
   g5=g4-100*f4;      f5=Floor[g5/10];    f6=g5-10*f5;
   f={f1,f2,f3,f4,f5,f6}; fx={0,0,0,0,0,0};
   j=1; Do [If [f[[i]]>0, fx[[i]]=j++],{i,1,6}];
   Return[{f,fx}];
 ];
 CountAssignedNodeFreedoms[p_]:= Module[{f1,g2,f2,g3,f3,g4,f4,g5,f5,f6},
   If [p==0, Return[0]];
   f1=Floor[p/100000];
   g2=p -100000*f1;   f2=Floor[g2/10000]; g3=g2-10000*f2;
   f3=Floor[g3/1000]; g4=g3-1000*f3;      f4=Floor[g4/100];
   g5=g4-100*f4;      f5=Floor[g5/10];    f6=g5-10*f5;
   Return[Min[1,f1]+Min[1,f2]+Min[1,f3]+Min[1,f4]+Min[1,f5]+Min[1,f6]];
 ];
ElementFreedomMapTable[edofNL_,dofNL_,eNFAL_]:= Module[
  {a,c,cNL,ef,eFMT,etag,etagold, i,j,k,
   numeNL=Length[edofNL],neldof,nf,ntag,ntagold},
  c=neldof=etagold=ntagold=0; ef=nf={0,0,0,0,0,0}; cNL=Table[0,{numeNL}];
  Do[etag=edofNL[[i]];
     If [etag!=etagold, c=CountAssignedNodeFreedoms[etag];
     etagold=etag]; cNL[[i]]=c; neldof=neldof+c,
  {i,numeNL}];
  eFMT=Table[0,{neldof}]; k=etagold=0;
  Do [c=cNL[[i]]; a=eNFAL[[i]]; etag=edofNL[[i]]; ntag=dofNL[[i]];
     If [etag==ntag, Do [eFMT[[++k]]=++a, {j,1,c}]; Continue[]];
     If [etag!=etagold, ef=UnpackNodeFreedomTags[etag]; etagold=etag];
     If [ntag!=ntagold, nf=UnpackNodeFreedomTags[ntag]; ntagold=ntag];
     Do [If [nf[[j]]!=0, a++];
         If [ef[[j]]!=0, k++; If [nf[[j]]!=0, eFMT[[k]]=a]],
     {j,1,6}],
  {i,1,numeNL}];
  Return[eFMT]
];
```

Cell 13.7 Skymatrix Utility Modules Used in Assembler Test

```
SymmSkyMatrixUpperTrianglePrint[S_]:= Module[
 {p,a,cycle,i,ij,it,j,j1,j2,jref,jbeg,jend,kcmax,k,kc,m,n,c,t},
 {p,a}=S; n=Dimensions[p][[1]]-1; kcmax=5; jref=0;
  Label[cycle]; Print[" "];
   jbeg=jref+1; jend=Min[jref+kcmax,n]; kc=jend-jref;
   t=Table[" ",{jend+1},{kc+1}];
   Do [If [p[[j+1]]>0,c=" ",c="*"];
      t[[1,j-jref+1]]=StringJoin[c,"Col",ToString[PaddedForm[j,3]]],
   {j,jbeg,jend}];  it=1;
   Do [it++;  If [p[[i+1]]>0,c=" ",c="*"];
       t[[it,1]]=StringJoin[c,"Row",ToString[PaddedForm[i,3]]]; j=jref;
       Do [j++; If [j<i, Continue[]]; ij=Abs[p[[j+1]]]+i-j;
          If [ij<=Abs[p[[j]]], Continue[]];
          t[[it,k+1]]=PaddedForm[a[[ij]]//FortranForm,{7,4}],
       {k,1,kc}],
   {i,1,jend}];
   Print[TableForm[Take[t,it],TableAlignments->{Right},
         TableDirections->{Column,Row},TableSpacing->{0,2}]];
   jref=jend; If[jref<n,Goto[cycle]];
];
SymmSkyMatrixConvertToFull[S_]:= Module[
  {p,a,aa,n,j,jj,jmj,k},
  {p,a}=S; n=Length[p]-1; aa=Table[0,{n},{n}];
  Do [jmj=Abs[p[[j]]]; jj=Abs[p[[j+1]]]; aa[[j,j]]=a[[jj]];
     Do [aa[[j,j-k]]=aa[[j-k,j]]=a[[jj-k]],{k,1,jj-jmj-1}],
  {j,1,n}];  Return[aa];
  ];
```

Cell 13.8 Module to Generate the MELT Used in Assembler Test

```
DefineElementType[MELT_,type_,descriptor_]:= Module [
  {elt=MELT,etype,t,tt=0,numtyp},
   etype=Prepend[descriptor,type];
   Do [ If [elt[[t,1]]==type, tt=t; Break[] ], {t,1,Length[elt]}];
   If [tt==0, AppendTo[elt,etype], elt[[tt]]=etype];
   Return[elt];
 ];

MELT= {};
MELT= DefineElementType[MELT,
      "Bar2D.2",  {"STM",1,"BAR",   "MOM","SEG",1000,{110000}}];
MELT= DefineElementType[MELT,
      "BeamC2D.2",{"STM",1,"PBEAC1","MOM","SEG",1000,{110001}}];
MELT= DefineElementType[MELT,
      "MFC",      {"STM",0,"CON",   "EXT","ARB",1000,{000000}}];
Print["MELT=",MELT//InputForm];
```

-

14

The Master Element Library

All finite elements available to the program are described through a Master Element Library Table, or MELT. The present Chapter describes the format of this table, and how it is constructed.

§14.1 THE MASTER ELEMENT LIBRARY TABLE

The MELT is a built-in table prepared by the program developer. It may be viewed as a catalog of elements available to the program.

§14.2 ELEMENT TYPE

The element *type* describes its function in sufficient detail to the FE program so that it can process the element through the appropriate modules or subroutines.

The MELT has the configuration sketched in Figure 14.1. It consists of (type,descriptor) pairs. There is one pair for each defined element type.

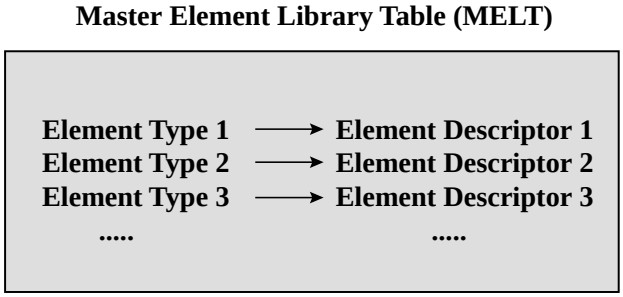


Figure 14.1. Configuration of the Master Element Library Table or MELT.

The element type is concisely defined by the user through an *element type name*. This name specifies the element externally. It serves as an identifier by the user to specify which element is to be used. The descriptor is an internal representation of the element. The following subsections specify these two objects in more detail.

§14.2.1 Element Type Name

The *element type name,* or simply *element name,* is a character string through which the user specifies the function of the element.

The complexity of this name depends on the level of generality of the underlying FE code. As an extreme case, consider a simple code that uses one and only one element type. No name is then required.

Most finite element codes, however, implement several element types and names appear. Some ancient programs use numeric codes such as 103 or 410. More common nowadays is the use of character strings with some mnemonic touches, such as "QUAD9" or "BEAM2".

In the present scheme an element type name is assumed. The choice of names is up to the program developer. The name implicitly define the element type descriptor, as illustrated in Figure 14.2.

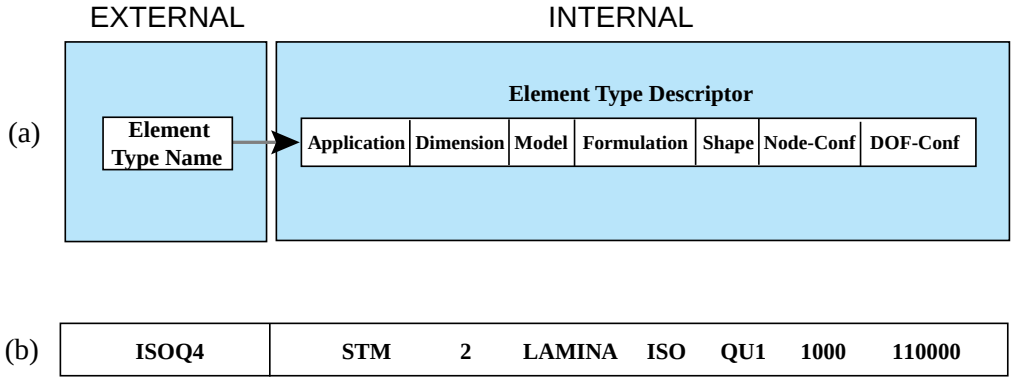


Figure 14.2. Element type name and descriptor: (a) the name is external and specified by the user whereas the descriptor is internal and only seen by the program; (b) an example.

Table 14.1 Application specification in element type descriptor (examples)

Identifier	Application
`"ACF"`	Acoustic fluid
`"EMM"`	Electromagnetic medium
`"EUF"`	Euler fluid
`"NSF"`	Navier-Stokes fluid
`"RBM"`	Rigid body mechanics
`"STM"`	Structural mechanics
`"THM"`	Thermomechanical

§14.2.2 Element Type Descriptor

The element *type descriptor* or simply *element descriptor* is a list that contains seven components that collectively classify the element in sufficient detail to be routed, during the processing phases, to the appropriate element formation routines. The descriptor does *not* include information processed within the routine, such as node coordinates, materials and fabrication data.

The seven components are either items or sublists, and appear in the following order:

```
descriptor = {application, dimensionality, model, formulation,
              geometric-shape, node-configuration, freedom-configuration }
```
(14.1)

See Figure 14.2. In programming, these may be identified by names such as:

```
eDL = { eDapp, eDdim, eDmod, eDfrm, eDsha, eDnod, eDdof }
```
(14.2)

or similar conventions.

As noted above, the connection between element type names and descriptor is done through a built-in data structure called the Master Element Library Table or `MELT`. This data structure is updated by the program developer as elements are added to the library. The organization of the `MELT` is described in Chapter 4.

§*14.2.2.1 Application*

The *application* item is a 3-character identification tag that explains what kind of physical problem the element is used for. The most important ones are listed in Table 14.1. The data structures described here emphasize Structural Mechanics elements, with tag `"STM"`. However, many of the attributes apply equally to the other applications of Table 14.1, if those applications ared modeled by the finite element method.

§*14.2.2.2 Dimensionality*

The *dimensionality* item is an integer that defines the number of intrinsic space dimensions of a mechanical element: 0, 1, 2 or 3. Multipoint contraint and multifreedom constraint elements are conventionally assigned a dimensionality of 0.

This attribute should not be confused with spatial dimensionality, which is 2 or 3 for 2D or 3D analysis, respectively. Spatial dimensionality is the same for all elements. For example a bar or beam element has intrinsic dimensionality of 1 but may be used in a 2D or 3D context.

§*14.2.2.3 Model*

The *model* item is linked to the application and dimensionality attributes to define the mathematical model used in the element derivation. It is specified by a character string containing up to 6 characters. Some common models used in Structural Mechanics applications are alphabetically listed in Table 14.1.

In that Table, 0/1 after bending models identifies the so-called C^0 and C^1 formulations, respectively. For example `"PLATE1"` is another moniker for Kirchhoff plate or thin plate. `"LAMINA"` and `"SLICE"` are names for 2D elements in plane stress and plane strain, respectively. `"FRUS0"` and `"FRUS1"` are axisymmetric shell elements obtained by rotating C^0 and C^1 beams, respectively, 360 degrees. `"RING"` is an axisymmetric solid element.

Other mathematical model identifiers can of course be added to the list as program capabilities expand. One common expansion is through the addition of *composite elements*; for example a stiffened shell or a girder box.

Table 14.1 Structural Mechanics model specification in element type descriptor

Appl	Dim	Identifier	Mathematical model
"STM"	0	"POINT"	Point
"STM"	0	"CON"	Multipoint constraint
"STM"	0	"CON"	Multifreedom constraint
"STM"	1	"BAR"	Bar
"STM"	1	"PBEAM0"	C^0 plane beam
"STM"	1	"PBEAM1"	C^1 plane beam
"STM"	1	"PBEAC0"	C^0 plane beam-column
"STM"	1	"PBEAC1"	C^1 plane beam-column
"STM"	1	"SBEAM0"	C^0 space beam
"STM"	1	"SBEAM1"	C^1 space beam
"STM"	1	"CABLE"	Cable
"STM"	1	"ARCH"	Arch
"STM"	1	"FRUST0"	C^0 axisymmetric shell
"STM"	1	"FRUST1"	C^1 axisymmetric shell
"STM"	2	"LAMIN"	Plane stress (membrane)
"STM"	2	"SLICE"	Plane strain slice
"STM"	2	"RING"	Axisymmetric solid
"STM"	2	"PLATE0"	C^0 (Reissner-Mindlin) plate
"STM"	2	"PLATE1"	C^1 (Kirchhoff) plate
"STM"	2	"SHELL0"	C^0 shell
"STM"	2	"SHELL1"	C^1 shell
"STM"	3	"SOLID"	Solid

§*14.2.2.4 Formulation*

The *formulation* item indicates, through a three-letter string, the methodology used for deriving the element. Some of the most common formulation identifiers are listed in Table 14.2.

§*14.2.2.5 Geometric Shape*

The geometric shape, in conjunction with the intrinsic dimensionality attribute, identifies the element geometry by a 3-character string. Allowable identifiers, paired with element dimensionalities, are listed in Table 14.3.

Not much geometrical variety can be expected of course in 0 and 1 dimensions. Actually there are two possibility for zero dimension: "DOT" identifies a point element (for example, a concentrated mass or a spring-to-ground) whereas "CON" applies to MPC (MultiPoint Constraint) and MFC (MultiFreedom Constraint) elements.

In two dimensions triangles and quadrilaterals are possible, with identifiers "TR" and "QU" respectively, followed by a number. In three dimensions the choice is between tetrahedra, wedges and bricks, which are identified by "TE", "WE" and "BR", respectively, also followed by a number. The

Table 14.2 Structural Mechanics formulation specification in element type descriptor

Identifier	Formulation
"ANS"	Assumed Natural Strain
"AND"	Assumed Natural Deviatoric Strain
"EFF"	Extended Free Formulation
"EXT"	Externally supplied: used for MFCs
"FRF"	Free Formulation
"FLE"	Flexibility (assumed force)
"HET"	Heterosis
"HYB"	Hybrid
"HYP"	Hyperparametric
"ISO"	Isoparametric
"MOM"	Mechanics of Materials
"SEM"	Semi-Loof
"TEM"	Template (most general of all)
"TMX"	Transfer Matrix

number following the shape identifier specifies whether the element is constructed as a unit, or as a macroelement assembly.

The "ARB" identifier is intended as a catch-all for shapes that are too complicated to be described by a short sequence.

Table 14.3 Shape specification in element type descriptor

Dim	Identifier	Geometric shape
0	"DOT"	Point; e.g. a concentrated mass
0	"CON"	Constraint element (conventionally)
1	"SEG"	Line segment
2	"TR1"	Triangle
2	"QU1"	Quadrilateral
2	"QU2"	Quadrilateral built of 2 triangles
2	"QU4"	Quadrilateral built of 4 triangles
3	"TE1"	Tetrahedron
3	"WE1"	Wedge
3	"WE3"	Wedge built of 3 tetrahedra
3	"BR1"	Brick
3	"BR5"	Brick built of 5 tetrahedra
3	"BR6"	Brick built of 6 tetrahedra
2-3	"VOR"	Voronoi cell
1-3	"ARB"	Arbitrary shape: defined by other attributes

§*14.2.2.6 Element Nodal Configuration*

The node configuration of the element is specified by a list of four integer items:

```
node-configuration := { eCNX, eSNX, eFNX, eINX }          (14.3)
```

Here `eCNX`, `eSNX`, `eFNX` and `eINX`, are abbreviations for element-corner-node-index, element-side-node-index, element-face-node-index and element-internal-node-index, respectively. Their value ranges from 0 through 2 with the meaning explained in Table 14.4. For storage and display convenience, the four indices are often decimally packed as one integer:

```
eCSFIX = 1000*eCNX + 100*eSNX + 10*eFNX + eINX          (14.4)
```

Note that the full set of indices only applies to elements with intrinsic dimensionality of 3. If dimensionality is 2 or less, `eINX` is ignored. If dimensionality is 1 or less, `eFNX` and `eINX` are ignored. Finally if dimensionality is 0, all except `eCNX` are ignored. Some specific examples:

20-node triquadratic brick	`ECSFI=1100`
27-node triquadratic brick	`ECSFI=1111`
64-node tricubic brick	`ECSFI=1222 (64=8*1+12*2+6*4+8)`
8-node serendipity quad	`ECSFI=1100`
9-node biquadratic quad	`ECSFI=1110`
10-node cubic lamina (plane stress) triangle	`ECSFI=1210 (10=3*1+3*2+1)`

Table 14.4 Node configuration in element type descriptor

Dim	Indices	Meaning/remarks
1–3	`eCNX=0,1`	element has `eCNX` nodes per corner
1–3	`eSNX=0,1,2`	element has `eSNX` nodes per side
2–3	`eFNX=0,1,2`	if `eFNX=0`, no face nodes
		if `eFNX=1`, one node per face
		if `eFNX=2`, as many face nodes as face corners
3	`eFNX=0,1,2`	if `eINX=0`, no face nodes
		if `eINX=1`, one node per face
		if `eINX=2`, as many face nodes as face corners
0	`eCSFIX=1000`	all nodes are treated as corners
Some exotic node configurations are omitted; cf. Remark 14.2		

REMARK 14.1

Note that the meaning of ”interior nodes” here does not agree with the usual FEM definition except for 3D elements. A plane stress element with one center node has `eFNX=1` and `eINX=0` instead of `eFNX=0` and `eINX=1`. In other words, that *center node is considered a face node*. Similarly a bar or beam element with two nodes at its third-points has `eSNX=2`, `eFNX=eINX=0`; that is, those nodes are considered side nodes. As explained next, the reason for this convention is mixability.

Why use indices instead of actual node counts? Mixability tests (connecting different elements together) are considerably simplified using this item. More specifically, elements with same application, model, formulation, `eCNX`, `eSNX`, `eFNX`, (in 3D) and `eCNX`, `eSNX` (in 2D) can be mixed if, in addition, some geometric compatibility conditions are met, while dimensionality and geometric shape attributes may be different. [Note that `eINX` and `eFNX` are irrelevant in 3D and 2D, respectively.] Two examples:

(i) Two solid elements with ECSFI=1220, 1221 and 1222 are candidates for mixing because interior nodes are irrelevant to such mating. This is true regardless of whether those elements are bricks, wedges or tetrahedra. Of course the common faces must have the same geometric shape; this represents another part of the mixability test.

(ii) Mixability across different dimensionalities is often of interest. For example, a 3-node bar with ECSFI=1100 attached along the edge of a quadrilateral or triangular lamina with `ECSFI=1100` or `1110`. Similarly a 2-node beam, which has `ECSFI=1000`, may be attached to a plate with `ECSFI=1000` but not to one with `ECSFI=1100`.

Values other than those listed in Table 14.5 are reserved for exotic configurations. For example, elements with ”double corner nodes” (`eCNX=2`) are occassionally useful to represent singularities in fracture mechanics.

§*14.2.2.7 Element Freedom Configuration*

The element freedom configuration defines the *default freedom assignation* at element nodes. Here “default” means that the assignation may be ovewriden on a node by node basis by the Freedom Definition data studied in another Chapter. A brief introduction to the concept of freedom assignment is needed here. More details are given in that chapter. The discussion below is limited to Structural Mechanics elements.

Table 14.6 Freedom Signatures in element descriptor

Signature	Meaning
0	Freedom is not assigned ("off")
1	Freedom is assigned ("on")

At each node n a local Cartesian reference system $(\bar{x}_n, \bar{y}_n, \bar{z}_n)$ may be used to define freedom directions. If these directions are not explicitly specified, they are assumed to coincide with the global system (x, y, z). (This is in fact the case in the present implementation).

In principle up to six degrees of freedom can be assigned at a node. These are identified by the symbols

```
tx, ty, tz, rx, ry, rz
```  (14.5)

Symbols `tx`, `ty` and `tz` denote translations along axes $\bar{x}_n$, $\bar{y}_n$ and $\bar{z}_n$, respectively, whereas `rx`, `ry` and `rz` denotes the rotations about those axes. If a particular freedom, say `rz`, appears in the finite element equations of the element it is said to be *assigned*. Otherwise it is *unassigned*. For example, consider a flat thin plate element located in the global (x, y) plane, with all local reference systems aligned with that system. Then `tz`, `rx`, and `ry` are assigned whereas `tx`, `ty` and `rz` are not.

The freedom configuration at corner, side, face and interior nodes of an element is defined by four integers denoted by

```
eFS = { eFSC, eFSS, eFSF, eFSI }
```  (14.6)

which are called *element freedom signatures*. Omitted values are assumed zero.

Each of the integers in (14.6) is formed by decimally packing six *individual freedom signature* values:

```
eFSC = 100000*txC + 10000*tyC + 1000*tzC + 100*rxC + 10*ryC + rzC
eFSS = 100000*txS + 10000*tyS + 1000*tzS + 100*rxS + 10*ryC + rzS
eFSF = 100000*txF + 10000*tyF + 1000*tzF + 100*rxF + 10*ryC + rzF
eFSI = 100000*txI + 10000*tyI + 1000*tzI + 100*rxI + 10*ryC + rzI
```  (14.7)

The freedom signature `txC` is associated with the translation `tx` at corner nodes. Similarly, `rxS` is associated with the rotation `rx` at side nodes. And so on. The signature value specifies whether the associated freedom is on or off, as indicated in Table 14.6. Signatures other than 0 or 1 are used in the Master Freedom Definition Table described in Chapter 7 to incorporate the attribute called *freedom activity*.

REMARK 14.2

If the element lacks node of a certain type the corresponding signature is zero. For example, if the element has only corner nodes, `eFSS = eFSF = eFSI = 0`, and these may be omitted from the list (14.6).

REMARK 14.3

For an MFC element the signatures, if given, are conventionally set to zero, since the node configuration is obtained from other information.

Cell 14.1 Defining an Element Type in MELT

```
DefineElementType[MELT_,type_,descriptor_]:= Module [
  {elt=MELT,etype,t,tt=0,numtyp},
   etype=Prepend[descriptor,type];
   Do [ If [elt[[t,1]]==type, tt=t; Break[] ], {t,1,Length[elt]}];
   If [tt==0, AppendTo[elt,etype], elt[[tt]]=etype];
   Return[elt];
 ];

MELT= {};
MELT= DefineElementType[MELT,
      "Bar2D.2",  {"STM",1,"BAR",   "MOM","SEG",1000,{110000}}];
MELT= DefineElementType[MELT,
      "BeamC2D.2",{"STM",1,"PBEAC1","MOM","SEG",1000,{110001}}];
MELT= DefineElementType[MELT,
      "MFC",      {"STM",0,"CON",   "EXT","ARB",1000,{000000}}];
Print["MELT=",MELT//InputForm];
PrintMasterElementLibraryTable[MELT];
```

Cell 14.2 Output from the Program of Cell 14.1

```
MELT={{"Bar2D.2", "STM", 1, "BAR", "MOM", "SEG", 1000, {110000}},

   {"BeamC2D.2", "STM", 1, "PBEAC1", "MOM", "SEG", 1000, {110001}},

   {"MFC", "STM", 0, "CON", "MOM", "ARB", 1000, {0}}}
TypeId     App   Dim   Model   Form   Sha  Nod-con  DOF-con
Bar2D.2    STM   1     BAR     MOM    SEG  1000     {110000}
BeamC2D.2  STM   1     PBEAC1  MOM    SEG  1000     {110001}
MFC        STM   0     CON     MOM    ARB  1000     {0}
```

§14.3 MATHEMATICA MODULES FOR BUILDING THE MELT

§14.3.1 Defining an Individual Node

Module `DefineElementType` is displayed in Cell 14.1. It enters a new element type into the `MELT`. It is invoked as follows:

```
MELT = DefineElementType [MELT,Type,Descriptor]
```

The input arguments are the existing `MELT`, the element type name, and the element descriptor. The

Cell 14.5 Printing the Master Node Definition Table

```
PrintMasterElementLibraryTable[MELT_]:= Module[
  {numtyp=Length[MELT],t,n},
   t=Table["    ",{numtyp+1},{8}];
   Do [
       t[[n+1,1]]=MELT[[n,1]];
       t[[n+1,2]]=MELT[[n,2]];
       t[[n+1,3]]=ToString[MELT[[n,3]] ];
       t[[n+1,4]]=MELT[[n,4]];
       t[[n+1,5]]=MELT[[n,5]];
       t[[n+1,6]]=MELT[[n,6]];
       t[[n+1,7]]=ToString[MELT[[n,7]] ];
       t[[n+1,8]]=ToString[MELT[[n,8]] ],
   {n,1,numtyp}];
   t[[1]] = {"TypeId","App ","Dim ","Model","Form ","Sha",
             "Nod-con","DOF-con"};
   Print[TableForm[t,TableAlignments->{Left},
           TableDirections->{Column,Row},TableSpacing->{0,2}]];
   ];

MELT={{"BAR2D.2","STM",1,"BAR",  "MOM","SEG",1000,{110000}},
      {"BAR3D.2","STM",1,"BAR",  "MOM","SEG",1000,{111000}},
      {"QLAM.4","STM",2,"LAMINA","ISO","QU1",1000,{110000}}};
PrintMasterElementLibraryTable[MELT];
```

Cell 14.4 Output from the Program of Cell 14.3

```
TypeId   App   Dim   Model   Form   Sha  Nod-con  DOF-con
BAR2D.2  STM   1     BAR     MOM    SEG  1000     {110000}
BAR3D.2  STM   1     BAR     MOM    SEG  1000     {111000}
QLAM.4   STM   2     LAMINA  ISO    QU1  1000     {110000}
```

latter is supplied as a list of seven items. Before entering the first element type, the `MELT` should be initialized by a statement such as `MELT={ }`.

The statements following `DefineIndividualNode` in Cell 14.1 test the module by building a `MELT` with several element types, which is then printed with the module `PrintMasterElementLibraryTable` described below. The output is shown in Cell 14.2.

§14.3.2 Printing the MELT

Module `PrintMasterElementLibraryTable`, listed in Cell 14.3, prints the `MELT` in a tabular

format. Its only input argument is the `MELT`. The module is tested by the statements that follow it, which build an MNDT directly and print it.

15

Implementation of One-Dimensional Elements

The present Chapter illustrates, through specific examples, the programming of one-dimensional elements: bars and beams, using *Mathematica* as implementation language.

§15.1 2D BAR ELEMENT

The two-node, prismatic, two-dimensional bar element is shown in Figure 15.1. It has two nodes and four degrees of freedom. The element node displacements and congugate forces are

$$\mathbf{u}^{(e)} = \begin{bmatrix} u_{x1} \\ u_{y1} \\ u_{x2} \\ u_{y2} \end{bmatrix}, \quad \mathbf{f}^{(e)} = \begin{bmatrix} f_{x1} \\ f_{y1} \\ f_{x2} \\ f_{y2} \end{bmatrix}. \tag{15.1}$$

The element geometry is described by the coordinates x_i, y_i, $i = 1, 2$ of the two end nodes. The two material properties involved in the stiffness and mass computations are the modulus of elasticity E and the mass density per unit volume ρ. The only fabrication property required is the cross section area A. All of these properties are constant over the element.

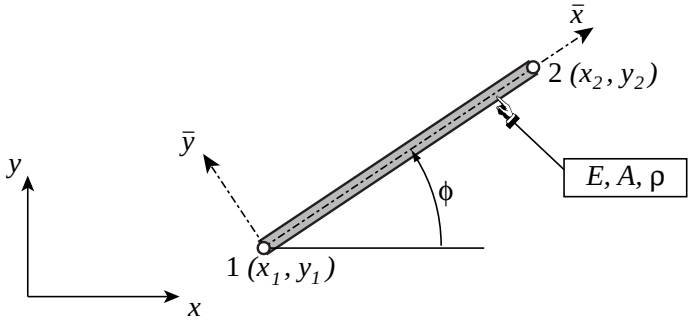


Figure 1. Plane bar element.

§15.1.1 Element Formulation

The element stiffness matrix for the 2B bar element is given by (cf. IFEM Chapter 3):

$$\begin{aligned} \mathbf{K}^{(e)} &= \frac{EA}{\ell} \begin{bmatrix} c^2 & sc & -c^2 & -sc \\ sc & s^2 & -sc & -s^2 \\ -c^2 & -sc & c^2 & sc \\ -sc & -s^2 & sc & s^2 \end{bmatrix} \\ &= \frac{E^{(e)}A^{(e)}}{\ell^3} \begin{bmatrix} \Delta x^2 & \Delta x\,\Delta y & -\Delta x^2 & -\Delta x\,\Delta y \\ \Delta x\,\Delta y & \Delta y^2 & -\Delta x\,\Delta y & -\Delta y^2 \\ -\Delta x^2 & -\Delta x\,\Delta y & \Delta x^2 & \Delta x\,\Delta y \\ -\Delta x\,\Delta y & -\Delta y^2 & \Delta x\,\Delta y & \Delta y^2 \end{bmatrix}. \end{aligned} \tag{15.2}$$

Here $c = \cos\phi = \Delta x/\ell$, $s = \sin\phi = \Delta y/\ell$, in which $\Delta x = x_2 - x_1$, $\Delta y = y_2 - y_1$, $\ell = \sqrt{\Delta x^2 + \Delta y^2}$, and ϕ is the angle formed by $\bar{x}$ and x, measured from x positive counterclockwise

Cell 15.1 *Mathematica* **Modules to Form Stiffness and Mass Matrices of a 2-Node, 2D Prismatic Bar Element**

```
Stiffness2DBar[{{x1_,y1_},{x2_,y2_}},{Em_,rho_},{A_}]:= Module[
   {dx,dy,len3,Ke}, dx=x2-x1; dy=y2-y1;
    len3=(dx^2+dy^2)*PowerExpand[Sqrt[dx^2+dy^2]];
    Ke=(Em*A/len3)*{{ dx^2,  dx*dy,-dx^2, -dx*dy},
                    { dx*dy, dy^2, -dy*dx,-dy^2},
                    {-dx^2, -dy*dx, dx^2,  dy*dx},
                    {-dy*dx,-dy^2,  dy*dx, dy^2}};
    Return[Ke]
 ];

ConsistentMass2DBar[{{x1_,y1_},{x2_,y2_}},{Em_,rho_},{A_}]:= Module[
   {dx,dy,len,Me}, dx=x2-x1; dy=y2-y1;
    len=PowerExpand[Sqrt[dx^2+dy^2]];
    Me=(rho*A*len/6)*{{2,0,1,0},{0,2,0,1},{1,0,2,0},{0,1,0,2}};
    Return[Me]
 ];

LumpedMass2DBar[{{x1_,y1_},{x2_,y2_}},{Em_,rho_},{A_}]:= Module[
   {dx,dy,len,Me}, dx=x2-x1; dy=y2-y1;
    len=PowerExpand[Sqrt[dx^2+dy^2]];
    Me=(rho*A*len/2)*{{1,0,0,0},{0,1,0,0},{0,0,1,0},{0,0,0,1}};
    Return[Me]
];
```

(see Figure 15.1). The last matrix expression is useful in symbolic work, because it enhances simplification possibilities.

The consistent and lumped mass matrix are given by

$$\mathbf{M}_C^{(e)} = \frac{\rho\ell}{6}\begin{bmatrix} 2 & 0 & 1 & 0 \\ 0 & 2 & 0 & 1 \\ 1 & 0 & 2 & 0 \\ 0 & 1 & 0 & 2 \end{bmatrix}, \qquad \mathbf{M}_L^{(e)} = \frac{\rho\ell}{2}\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{15.3}$$

Both mass matrices are independent of the orientation of the element. For the derivation see, e.g., the book by Przemieniecki, *Matrix Structural Analysis*, Dover, 1968.

§15.1.2 Element Formation Modules

Cell 15.1 lists three *Mathematica* modules:

`Stiffness2DBar` returns the element stiffness matrix $\mathbf{K}^{(e)}$ of a 2-node, 2-dimensional, prismatic bar element, given by (15.2).

Cell 15.2 Test of 2D Bar Element Modules with Numerical Inputs

```
Ke= Stiffness2DBar[{{0.,0.},{10.,10.}},{1000.,1/6},{2*Sqrt[2.]}];
Print["\nNumerical Elem Stiff Matrix: \n"];Print[Ke//TableForm];
Print["Eigenvalues of Ke=",Chop[Eigenvalues[N[Ke]]]];
Print["Symmetry check=",Simplify[Transpose[Ke]-Ke]];

MeC= ConsistentMass2DBar[{{0.,0.},{10.,10.}},{1000.,1/6},{2*Sqrt[2.]}];
Print["\nNumerical Consistent Mass Matrix: \n"]; Print[MeC//TableForm];
Print["Eigenvalues of MeC=",Eigenvalues[MeC]];
Print["Symmetry check=",Simplify[Transpose[MeC]-MeC]];

MeL= LumpedMass2DBar[{{0.,0.},{10.,10.}},{1000.,1/6},{2*Sqrt[2.]}];
Print["\nNumerical Lumped Mass Matrix: \n"]; Print[MeL//TableForm];
Print["Eigenvalues of MeL=",Eigenvalues[MeL]];
```

Cell 15.3 Output from Test Statements of Cell 15.2

```
Numerical Elem Stiff Matrix:

100.    100.    -100.   -100.
100.    100.    -100.   -100.
-100.   -100.   100.    100.
-100.   -100.   100.    100.

Eigenvalues of Ke={400., 0, 0, 0}
Symmetry check={{0., 0., 0., 0.}, {0., 0., 0., 0.}, {0., 0., 0., 0.},
   {0., 0., 0., 0.}}

Numerical Consistent Mass Matrix:

2.22222   0         1.11111   0
0         2.22222   0         1.11111
1.11111   0         2.22222   0
0         1.11111   0         2.22222

Eigenvalues of MeC={3.33333, 3.33333, 1.11111, 1.11111}
Symmetry check={{0., 0., 0., 0.}, {0., 0., 0., 0.}, {0., 0., 0., 0.},
   {0., 0., 0., 0.}}

Numerical Lumped Mass Matrix:

3.33333   0         0         0
0         3.33333   0         0
0         0         3.33333   0
0         0         0         3.33333

Eigenvalues of MeL={3.33333, 3.33333, 3.33333, 3.33333}
```

Cell 15.4 Test of 2D Bar Element Modules with Symbolic Inputs

```
Ke=  Stiffness2DBar[{{0,0},{L,0}},{Em,rho},{A}]; Ke=Simplify[Ke];
Print["Symbolic Elem Stiff Matrix: \n"]; Print[Ke//InputForm];
Print["Eigenvalues of Ke=",Eigenvalues[Ke]//InputForm];

MeC= ConsistentMass2DBar[{{0,0},{L,0}},{Em,rho},{A}]; MeC= Simplify[MeC];
Print["\nSymbolic Consistent Mass Matrix: \n"]; Print[MeC//InputForm];
Print["\nEigenvalues of MeC=",Eigenvalues[MeC]//InputForm];
Print["Squared frequencies=",Simplify[Eigenvalues[Inverse[MeC].Ke]]//
        InputForm];

MeL= LumpedMass2DBar[{{0,0},{L,0}},{Em,rho},{A}]; MeL= Simplify[MeL];
Print["\nSymbolic Lumped Mass Matrix: \n"]; Print[MeL//InputForm];
Print["\nEigenvalues of MeL=",Eigenvalues[MeL]//InputForm];
Print["Squared frequencies=",Simplify[Eigenvalues[Inverse[MeL].Ke]]//
        InputForm];
```

Cell 15.5 Output from Test Statements of Cell 15.4

```
Symbolic Elem Stiff Matrix:

{{(A*Em)/L, 0, -((A*Em)/L), 0}, {0, 0, 0, 0},
  {-((A*Em)/L), 0, (A*Em)/L, 0}, {0, 0, 0, 0}}

Eigenvalues of Ke={0, 0, 0, (2*A*Em)/L}

Symbolic Consistent Mass Matrix:

{{(A*L*rho)/3, 0, (A*L*rho)/6, 0}, {0, (A*L*rho)/3, 0, (A*L*rho)/6},
  {(A*L*rho)/6, 0, (A*L*rho)/3, 0}, {0, (A*L*rho)/6, 0, (A*L*rho)/3}}

Eigenvalues of MeC={(A*L*rho)/6, (A*L*rho)/6, (A*L*rho)/2, (A*L*rho)/2}

Squared frequencies={0, 0, 0, (12*Em)/(L^2*rho)}

Symbolic Lumped Mass Matrix:

{{(A*L*rho)/2, 0, 0, 0}, {0, (A*L*rho)/2, 0, 0}, {0, 0, (A*L*rho)/2, 0},
  {0, 0, 0, (A*L*rho)/2}}

Eigenvalues of MeL={(A*L*rho)/2, (A*L*rho)/2, (A*L*rho)/2, (A*L*rho)/2}

Squared frequencies={0, 0, 0, (4*Em)/(L^2*rho)}
```

`ConsistentMass2DBar` returns the consistent mass matrix $\mathbf{M}_C^{(e)}$ of a 2-node, 2-dimensional, prismatic bar element, given by (15.3).

`LumpedMass2DBar` returns the lumped mass matrix $\mathbf{M}_L^{(e)}$ of a 2-node, 2-dimensional, prismatic bar element, given by (15.3).

The argument sequence of the three modules is the same:

```
[ {{x1,y1},{x2,y2}}, {Em,rho}, {A} ]
```

Here `x1,y1` and `x2,y2` are the coordinates of the end nodes, and `Em`, `rho` and `A` stand for E, ρ and A, respectively. Note that the arguments are grouped into three lists:

coordinates, material properties, fabrication properties

This arrangement will be consistent followed for other elements.

§15.1.3 Testing the Bar Element Modules

The modules are tested by the statements collected in Cells 15.2 and 15.4.

Cell 15.2 tests a numerically defined element with end nodes located at $(0, 0)$ and $(10, 10)$, with $E = 1000$, $A = 2\sqrt{2}$ and $\rho = 1/6$. Executing the statements in Cell 15.2 produces the results listed in Cell 15.3. The tests consist of the following operations:

Testing $\mathbf{K}^{(e)}$. The stiffness matrix returned in `Ke` is printed. Its four eigenvalues are computed and printed. As expected three eigenvalues, which correspond to the three independent rigid body motions of the element, are zero. The remaining eigenvalue is positive and equal to EA/ℓ. The symmetry of `KE` is also checked.

Testing $\mathbf{M}_C^{(e)}$. The consistent mass matrix returned in `MeL` is printed. Its four eigenvalues are computed and printed. As expected they are all positive and form two pairs. The symmetry is also checked.

Testing $\mathbf{M}_L^{(e)}$. The lumped mass matrix returned in `MeL` is printed. This is a diagonal matrix and consequently its eigenvalues are the same as the diagonal entries.

Cell 15.4 tests a symbolically defined element with end nodes located at $(0, 0)$ and $(L, 0)$, which is aligned with the x axis. The element properties E, ρ and A are kept symbolic. Executing the statements in Cell 15.4 produces the results shown in Cell 15.5.

The sequence of tests on the symbolic element is essentially the same carried out before, but a frequency test has been added. This consists of solving the one-element vibration eigenproblem

$$\mathbf{K}^{(e)}\mathbf{v}_i = \omega_i^2\mathbf{M}_C^{(e)}\mathbf{v}_i, \qquad \mathbf{K}^{(e)}\mathbf{v}_i = \omega_i^2\mathbf{M}_L^{(e)}\mathbf{v}_i, \tag{15.4}$$

where ω_i are circular frequencies and $\mathbf{v}_i$ the associated eigenvectors or vibration mode shapes. Since the 3 rigid body modes are solution of (15.4), three zero frequencies are expected, which is borned out by the tests. The single positive nonzero frequency $\omega_a > 0$ corresponds to the vibration mode of axial extension and contraction. The consistent mass yields $\omega_a^2 = 12E/(\rho L^2)$ whereas the lumped mass yields $\omega_a^2 = 4E/(\rho L^2)$. The exact continuum solution for this axial mode is $\omega_a^2 = \pi^2 E/(\rho L^2)$. Hence the consistent mass overestimates the true frequency whereas the lumped mass underestimates it.

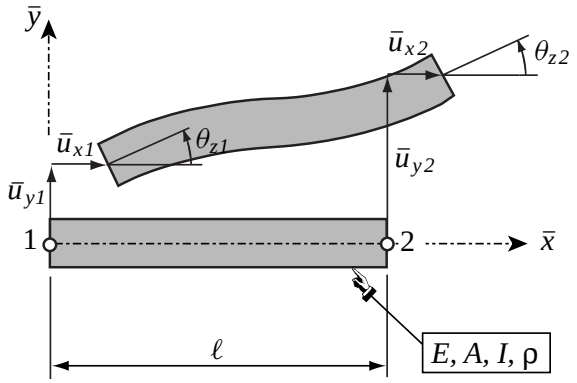


Figure 15.2. Plane beam-column element in its local system.

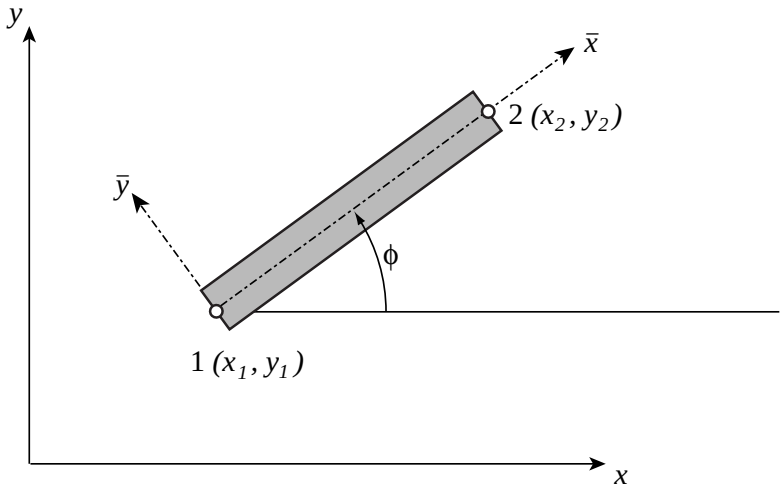


Figure 15.3. Plane beam-column element in the global system.

§15.1.4 MELT Definition

The 2D bar element is entered in the Master Element Library Table (`MELT`) described in Chapter 20, by the statement

```
MELT = DefineElementType[MELT, "Bar2D.2"
      {"STM", 1, "BAR", "MOM", "SEG",1000, {110000}}];
```

The element type name is `"Bar2D.2"`. The descriptor fields specify:

`"STM"`	element is of structural type
`1`	intrinsic dimensionality is one
`"BAR"`	mathematical model
`"MOM"`	formulation is by Mechanics of Materials
`"SEG"`	geometry is line segment
`1000`	element has corner nodes only
`{110000}`	initialized freedom signature for corner nodes

§15.2 2D BEAM-COLUMN ELEMENT

Beam-column elements model structural members that resist both axial and bending actions. This is the case in skeletal structures such as buildings. A plane beam-column element is a combination of a plane bar (such as that considered in §20.1), and a plane beam.

We consider such an element in its local system $(\bar{x}, \bar{y})$ as shown in Figure 15.2, and then in the global system (x, y) as shown in Figure 15.3. The element The six degrees of freedom and conjugate node forces of the elements are:

$$\bar{\mathbf{u}}^{(e)} = \begin{bmatrix} \bar{u}_{x1} \\ \bar{u}_{y1} \\ \theta_{z1} \\ \bar{u}_{x2} \\ \bar{u}_{y2} \\ \theta_{z2} \end{bmatrix}, \quad \bar{\mathbf{f}}^{(e)} = \begin{bmatrix} \bar{f}_{x1} \\ \bar{f}_{y1} \\ m_{z1} \\ \bar{u}_{x2} \\ \bar{u}_{y2} \\ m_{z2} \end{bmatrix}, \qquad \mathbf{u}^{(e)} = \begin{bmatrix} u_{x1} \\ u_{y1} \\ \theta_{z1} \\ u_{x2} \\ u_{y2} \\ \theta_{z2} \end{bmatrix}, \quad \mathbf{f}^{(e)} = \begin{bmatrix} f_{x1} \\ f_{y1} \\ m_{z1} \\ f_{x2} \\ f_{y2} \\ m_{z2} \end{bmatrix}. \tag{15.5}$$

The rotation angles θ and the nodal moments m are the same in the local and the global systems.

The element geometry is described by the coordinates $x_i, y_i, i = 1, 2$ of the two end nodes. The two material properties involved in the stiffness and mass computations are the modulus of elasticity E and the mass density per unit volume ρ. The fabrication properties required are the cross section area A and the bending moment of inertia I. All of these properties are taken to be constant over the element.

§15.2.1 Element Formulation

The stiffness matrix of prismatic beam column element in the local system $\bar{x}$, $\bar{y}$ is given by

$$\bar{\mathbf{K}}^{(e)} = \frac{EI}{\ell^3}\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ & 12 & 6\ell & 0 & -12 & 6\ell \\ & & 4\ell^2 & 0 & -6\ell & 2\ell^2 \\ & & & 0 & 0 & 0 \\ & & & & 12 & -6\ell \\ symm & & & & & 4\ell^2 \end{bmatrix} + \frac{EA}{\ell}\begin{bmatrix} 1 & 0 & 0 & -1 & 0 & 0 \\ & 0 & 0 & 0 & 0 & 0 \\ & & 0 & 0 & 0 & 0 \\ & & & 1 & 0 & 0 \\ & & & & 0 & 0 \\ symm & & & & & 0 \end{bmatrix} \tag{15.6}$$

The first matrix on the right is the contribution from the bending (beam) stiffness, as derived in IFEM Chapter 12, and the second is the contribution from the bar stiffness.

The consistent mass matrix in the local system $\bar{x}$, $\bar{y}$

$$\bar{\mathbf{M}}_C^{(e)} = \frac{\rho A\ell}{420}\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ & 156 & 22\ell & 0 & 54 & -13\ell \\ & & 4\ell^2 & 0 & 13\ell & -3\ell^2 \\ & & & 0 & 0 & 0 \\ & & & & 156 & -22\ell \\ symm & & & & & 4\ell^2 \end{bmatrix} + \frac{\rho A\ell}{6}\begin{bmatrix} 2 & 0 & 0 & 1 & 0 & 0 \\ & 0 & 0 & 0 & 0 & 0 \\ & & 0 & 0 & 0 & 0 \\ & & & 2 & 0 & 0 \\ & & & & 0 & 0 \\ symm & & & & & 0 \end{bmatrix} \tag{15.7}$$

Cell 15.6 *Mathematica* **Modules to Form Stiffness and Mass Matrices of a 2-Node, 2D Prismatic Beam-Column Element**

```
Stiffness2DBeamColumn[{{x1_,y1_},{x2_,y2_}},{Em_,nu_,rho_,alpha_},
    {A_,Iz_}]:= Module[{dx,dy,c,s,L,LL,ra,rb,Kebar,Ke},
    dx=x2-x1; dy=y2-y1; LL=dx^2+dy^2; L=PowerExpand[Sqrt[LL]];
    c=dx/L; s=dy/L; ra=Em*A/L; rb= Em*Iz/L^3;
    Kebar= ra*{
     { 1,0,0,-1,0,0},{0,0,0,0,0,0},{0,0,0,0,0,0},
     {-1,0,0, 1,0,0},{0,0,0,0,0,0},{0,0,0,0,0,0}} +
           rb*{
     { 0,0,0,0,0,0},{0, 12, 6*L,0,-12,6*L},{0,6*L,4*LL,0,-6*L,2*LL},
     { 0,0,0,0,0,0},{0,-12,-6*L,0,12,-6*L},{0,6*L,2*LL,0,-6*L,4*LL}};
     T={{c,s,0,0,0,0},{-s,c,0,0,0,0},{0,0,1,0,0,0},
        {0,0,0,c,s,0},{0,0,0,-s,c,0},{0,0,0,0,0,1}};
    Ke=Transpose[T].Kebar.T;
    Return[Ke]
 ];

ConsistentMass2DBeamColumn[{{x1_,y1_},{x2_,y2_}},{Em_,nu_,rho_,alpha_},
   {A_,Iz_}]:= Module[{dx,dy,c,s,L,LL,m,MeCbar,MeC},
    dx=x2-x1; dy=y2-y1; LL=dx^2+dy^2; L=PowerExpand[Sqrt[LL]];
    c=dx/L; s=dy/L; m=rho*L*A;
    MeCbar= (m/6)*{
     {2,0,0,1,0,0},{0,0,0,0,0,0},{0,0,0,0,0,0},
     {1,0,0,2,0,0},{0,0,0,0,0,0},{0,0,0,0,0,0}}+
         +(m/420)*{
     {0,0,0,0,0,0},{0,156,22*L,0,54,-13*L},{0,22*L,4*LL,0,13*L,-3*LL},
     {0,0,0,0,0,0},{0,54,13*L,0,156,-22*L},{0,-13*L,-3*LL,0,-22*L,4*LL}};
     T={{c,s,0,0,0,0},{-s,c,0,0,0,0},{0,0,1,0,0,0},
        {0,0,0,c,s,0},{0,0,0,-s,c,0},{0,0,0,0,0,1}};
     MeC=Transpose[T].MeCbar.T;
    Return[MeC]
 ];

LumpedMass2DBeamColumn[{{x1_,y1_},{x2_,y2_}},{Em_,nu_,rho_,alpha_},
   {A_,Iz_}]:= Module[{dx,dy,dxdx,dxdy,dydy,L,MeL},
    dx=x2-x1; dy=y2-y1; L=PowerExpand[Sqrt[dx^2+dy^2]];
    MeL=(rho*A*L/2)*{{1,0,0,0,0,0},{0,1,0,0,0,0},{0,0,0,0,0,0},
                     {0,0,0,1,0,0},{0,0,0,0,1,0},{0,0,0,0,0,0}};
    Return[MeL]
];
```

The first matrix on the RHS is the contribution from the bending (beam) inertia and the second from the axial (bar) inertia. These expressions are given in Przemieniecki, *loc cit.*.

Cell 15.7 Test of 2D Beam-Column Element Modules with Numerical Inputs

```
geom={{0,0},{5,0}}; mat={200,0,1/30,0}; fab={1/4,3};
Ke= Stiffness2DBeamColumn[geom,mat,fab];
Print["\nNumerical Elem Stiff Matrix: \n"];Print[Ke//InputForm];
Print["Eigenvalues of Ke=",Chop[Eigenvalues[N[Ke]]]];
Print["Symmetry check=",Simplify[Transpose[Ke]-Ke]];

MeC= ConsistentMass2DBeamColumn[geom,mat,fab];
Print["\nNumerical Consistent Mass Matrix: \n"]; Print[MeC//InputForm];
Print["Eigenvalues of MeC=",Eigenvalues[N[MeC]]];
Print["Squared vibration frequencies (consistent)=",
       Chop[Eigenvalues[Inverse[MeC].N[Ke]],10^(-7)]];
Print["Symmetry check=",Simplify[Transpose[MeC]-MeC]];

MeL= LumpedMass2DBeamColumn[geom,mat,fab];
Print["\nNumericaL Lumped Mass Matrix: \n"]; Print[MeL//InputForm];
Print["Eigenvalues of MeL=",Eigenvalues[N[MeL]]];
MeL[[3,3]]=MeL[[6,6]]=1/10000000;
Print["Squared vibration frequencies (lumped)=",
       Chop[Eigenvalues[Inverse[MeL].N[Ke]],10^(-7)]];
```

The displacement transformation matrix between local and global system is

$$\bar{\mathbf{u}}^{(e)} = \begin{bmatrix} \bar{u}_{x1} \\ \bar{u}_{y1} \\ \theta_{z1} \\ u_{x2} \\ \bar{u}_{y2} \\ \theta_{z2} \end{bmatrix} = \begin{bmatrix} c & s & 0 & c & s & 0 \\ -s & c & 0 & -s & c & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ c & s & 0 & c & s & 0 \\ -s & c & 0 & -s & c & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_{x1} \\ u_{y1} \\ \theta_{z1} \\ u_{x2} \\ u_{y2} \\ \theta_{z2} \end{bmatrix} = \mathbf{T}\mathbf{u}^{(e)} \tag{15.8}$$

where $c = \cos\phi$, $s = \sin\phi$, and ϕ is the angle between $\bar{x}$ and x, measured positive-counterclockwise from x; see Figure 15.3. The stiffness and consistent mass matrix in the global system are obtained through the congruential transformation $\mathbf{K}^{(e)} = \mathbf{T}^T\bar{\mathbf{K}}^{(e)}\mathbf{T}$, $\quad \mathbf{M}_C^{(e)} = \mathbf{T}^T\bar{\mathbf{M}}^{(e)}\mathbf{T}$.

The lumped mass matrix is diagonal and is the same in the local and global systems:

$$\mathbf{M}_L^{(e)} = \bar{\mathbf{M}}_L^{(e)} = \frac{\rho\ell}{2}\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}. \tag{15.9}$$

This form has zero rotational mass and hence singular. Sometimes it is desirable to invert $\mathbf{M}_L^{(e)}$ explicitly. If so it is useful to replace the zeros in locations (3, 3) and (6, 6) by a small number ϵ.

Cell 15.8 Output from Test Statements of Cell 15.7

```
Numerical Elem Stiff Matrix:
{{5058/125, -2856/125, -576/5, -5058/125, 2856/125, -576/5},
  {-2856/125, 3392/125, 432/5, 2856/125, -3392/125, 432/5},
  {-576/5, 432/5, 480, 576/5, -432/5, 240},
  {-5058/125, 2856/125, 576/5, 5058/125, -2856/125, 576/5},
  {2856/125, -3392/125, -432/5, -2856/125, 3392/125, -432/5},
  {-576/5, 432/5, 240, 576/5, -432/5, 480}}
Eigenvalues of Ke={835.2, 240., 20., 0, 0, 0}
Symmetry check={{0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0},
   {0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0},
   {0, 0, 0, 0, 0, 0}}

Numerical Consistent Mass Matrix:
{{313/21000, -2/2625, -11/1260, 83/14000, 2/2625, 13/2520},
  {-2/2625, 911/63000, 11/1680, 2/2625, 803/126000, -13/3360},
  {-11/1260, 11/1680, 5/504, -13/2520, 13/3360, -5/672},
  {83/14000, 2/2625, -13/2520, 313/21000, -2/2625, 11/1260},
  {2/2625, 803/126000, 13/3360, -2/2625, 911/63000, -11/1680},
  {13/2520, -13/3360, -5/672, 11/1260, -11/1680, 5/504}}
Eigenvalues of MeC={0.0365449, 0.0208333, 0.0121748, 0.00694444,
   0.00164952, 0.000424401}
Squared vibration frequencies (consistent)=
  {967680., 82944., 2880., 0, 0, 0}
Symmetry check={{0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0},
   {0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0},
   {0, 0, 0, 0, 0, 0}}

NumericaL Lumped Mass Matrix:
{{1/48, 0, 0, 0, 0, 0}, {0, 1/48, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0},
  {0, 0, 0, 1/48, 0, 0}, {0, 0, 0, 0, 1/48, 0}, {0, 0, 0, 0, 0, 0}}
Eigenvalues of MeL={0.0208333, 0.0208333, 0.0208333, 0.0208333, 0., 0.}
Squared vibration frequencies (lumped)=
             9        9
  {7.20001 10 , 2.4 10 , 960., 0, 0, 0}
```

§15.2.2 Element Formation Modules

Cell 15.6 lists three *Mathematica* modules:

`Stiffness2DBarColumn` returns the element stiffness matrix $\mathbf{K}^{(e)}$ of a 2-node, 2-dimensional, prismatic bar element, given by (15.8).

`ConsistentMass2DBarColumn` returns the consistent mass matrix $\mathbf{M}_C^{(e)}$ of a 2-node, 2-dimensional, prismatic bar element, given by (15.9).

`LumpedMass2DBarColumn` returns the lumped mass matrix $\mathbf{M}_L^{(e)}$ of a 2-node, 2-dimensional, prismatic bar element, given by (15.9).

Cell 15.9 Test of 2D Beam-Column Element Modules with Symbolic Inputs

```
ClearAll[L,Em,rho,A,Iz];
geom={{0,0},{L,0}}; mat={Em,0,rho,0}; fab={A,Iz};
Ke=  Stiffness2DBeamColumn[geom,mat,fab]; Ke=Simplify[Ke];
Print["Symbolic Elem Stiff Matrix: \n"]; Print[Ke//InputForm];
Print["Eigenvalues of Ke=",Eigenvalues[Ke]//InputForm];

MeC= ConsistentMass2DBeamColumn[geom,mat,fab]; MeC= Simplify[MeC];
Print["\nSymbolic Consistent Mass Matrix: \n"]; Print[MeC//InputForm];
Print["\nEigenvalues of MeC=",Eigenvalues[MeC]//InputForm];
Print["Squared frequencies=",Simplify[Eigenvalues[Inverse[MeC].Ke]]//
        InputForm];

MeL= LumpedMass2DBeamColumn[geom,mat,fab];
MeL= Simplify[MeL]; MeL[[3,3]]=MeL[[6,6]]=eps;
Print["\nSymbolic Lumped Mass Matrix: \n"]; Print[MeL//InputForm];
Print["\nEigenvalues of MeL=",Eigenvalues[MeL]//InputForm];
Print["Squared frequencies=",Simplify[Eigenvalues[Inverse[MeL].Ke]]//
        InputForm];
```

The argument sequence of the three modules is the same:

```
[ {{x1,y1},{x2,y2}}, {Em,nu,rho,alpha}, {A,Iz} ]
```

Here `x1,y1` and `x2,y2` are the coordinates of the end nodes, and `Em`, `nu`, `rho`, `alpha` `A` and `Iz` stand for E, ν, ρ, α A and I, respectively. Of these `nu` and `alpha` are not used but are included for future use. As usual the arguments are grouped into three lists:

coordinates, material properties, fabrication properties

This arrangement is consistent followed for all elements.

§15.2.3 Testing the Bar-Column Element Modules

The modules are tested by the statements collected in Cells 15.7 and 15.9.

Cell 15.7 tests a numerically defined element with end nodes located at (0, 0) and (5, 0) respectivelt, with $E = 200$, $\rho = 1/30$, $A = 1/4$ and $I = 3$. Executing the statements in Cell 15.7 produces the results listed in Cell 15.8. The tests consist of the following operations:

Testing $\mathbf{K}^{(e)}$. The stiffness matrix returned in `Ke` is printed. Its 6 eigenvalues are computed and printed. As expected three eigenvalues, which correspond to the three independent rigid body motions of the element, are zero. The remaining three eigenvalues are positive. The symmetry of `Ke` is also checked.

Testing $\mathbf{M}_C^{(e)}$. The consistent mass matrix returned in `MeC` is printed. Its symmetry is checked. The

Cell 15.10 Output from Test Statements of Cell 15.9

```
Symbolic Elem Stiff Matrix:
{{(A*Em)/L, 0, 0, -((A*Em)/L), 0, 0},
  {0, (12*Em*Iz)/L^3, (6*Em*Iz)/L^2, 0, (-12*Em*Iz)/L^3, (6*Em*Iz)/L^2},
  {0, (6*Em*Iz)/L^2, (4*Em*Iz)/L, 0, (-6*Em*Iz)/L^2, (2*Em*Iz)/L},
  {-((A*Em)/L), 0, 0, (A*Em)/L, 0, 0},
  {0, (-12*Em*Iz)/L^3, (-6*Em*Iz)/L^2, 0, (12*Em*Iz)/L^3,
   (-6*Em*Iz)/L^2}, {0, (6*Em*Iz)/L^2, (2*Em*Iz)/L, 0, (-6*Em*Iz)/L^2,
   (4*Em*Iz)/L}}
Eigenvalues of Ke={0, 0, 0, (2*A*Em)/L, (2*Em*Iz)/L,
   (6*Em*Iz*(4 + L^2))/L^3}

Symbolic Consistent Mass Matrix:
{{(A*L*rho)/3, 0, 0, (A*L*rho)/6, 0, 0},
  {0, (13*A*L*rho)/35, (11*A*L^2*rho)/210, 0, (9*A*L*rho)/70,
   (-13*A*L^2*rho)/420}, {0, (11*A*L^2*rho)/210, (A*L^3*rho)/105, 0,
   (13*A*L^2*rho)/420, -(A*L^3*rho)/140},
  {(A*L*rho)/6, 0, 0, (A*L*rho)/3, 0, 0},
  {0, (9*A*L*rho)/70, (13*A*L^2*rho)/420, 0, (13*A*L*rho)/35,
   (-11*A*L^2*rho)/210}, {0, (-13*A*L^2*rho)/420, -(A*L^3*rho)/140, 0,
   (-11*A*L^2*rho)/210, (A*L^3*rho)/105}}
Eigenvalues of MeC={(A*L*rho)/6, (A*L*rho)/2,
   (-(A*L*(-2040 - 20*L^2)*rho) -
      (-33600*A^2*L^4*rho^2 + A^2*L^2*(-2040 - 20*L^2)^2*rho^2)^(1/2))/
    16800, (-(A*L*(-2040 - 20*L^2)*rho) +
      (-33600*A^2*L^4*rho^2 + A^2*L^2*(-2040 - 20*L^2)^2*rho^2)^(1/2))/
    16800, (-(A*L*(-360 - 12*L^2)*rho) -
      (-2880*A^2*L^4*rho^2 + A^2*L^2*(-360 - 12*L^2)^2*rho^2)^(1/2))/1440,
     (-(A*L*(-360 - 12*L^2)*rho) +
     (-2880*A^2*L^4*rho^2 + A^2*L^2*(-360 - 12*L^2)^2*rho^2)^(1/2))/1440}
Squared frequencies={0, 0, 0, (720*Em*Iz)/(A*L^4*rho),
   (8400*Em*Iz)/(A*L^4*rho), (12*Em)/(L^2*rho)}

Symbolic Lumped Mass Matrix:
{{(A*L*rho)/2, 0, 0, 0, 0, 0}, {0, (A*L*rho)/2, 0, 0, 0, 0},
  {0, 0, eps, 0, 0, 0}, {0, 0, 0, (A*L*rho)/2, 0, 0},
  {0, 0, 0, 0, (A*L*rho)/2, 0}, {0, 0, 0, 0, 0, eps}}
Eigenvalues of MeL={eps, eps, (A*L*rho)/2, (A*L*rho)/2, (A*L*rho)/2,
   (A*L*rho)/2}
Squared frequencies={0, 0, 0, (2*Em*Iz)/(eps*L), (4*Em)/(L^2*rho),
   (6*Em*Iz)/(eps*L) + (48*Em*Iz)/(A*L^4*rho)}
```

six eigenvalues are computed and printed. As expected they are all positive. A frequency test is also carried out.

Testing $\mathbf{M}_L^{(e)}$. The lumped mass matrix returned in `MeL` is printed. This is a diagonal matrix and consequently its eigenvalues are the same as the diagonal entries. A frequency test is carried out.

Cell 15.9 tests a symbolically defined element with end nodes located at $(0, 0)$ and $(L, 0)$, which is aligned with the x axis. The element properties E, ρ, A and I are kept in symbolic form. Executing the statements in Cell 15.9 produces the results shown in Cell 15.10.

The sequence of tests on the symbolic element is similar to that in Cell 15.7. The vibration eigenproblems are

$$\mathbf{K}^{(e)}\mathbf{v}_i = \omega_i^2\mathbf{M}_C^{(e)}\mathbf{v}_i, \qquad \mathbf{K}^{(e)}\mathbf{v}_i = \omega_i^2\mathbf{M}_L^{(e)}\mathbf{v}_i, \tag{15.10}$$

where ω_i are circular frequencies and $\mathbf{v}_i$ the associated eigenvectors or vibration mode shapes. Because the 3 rigid body modes are solution of (15.10), three zero frequencies are expected for the consistent mass eigenproblem, which is borned out by the tests. The three positive nonzero frequencies corresponds to one free-free axial and two free-free bending modes. The consistent mass yields for the latter $\omega^2 = 720EI/(\rho AL^4)$ and $\omega^2 = 8400EI/(\rho AL^4)$ whereas the exact continuum solution for the first two free-free bending frequencies are $502EI/(\rho AL^4)$ and $1382EI/(\rho AL^4)$.

The lumped mass vibration eigenproblem yields three zero frequencies, two infinite frequencies (associated with zero rotational mass) and one finite positive axial vibration frequency, which is the same as that provided by the bar element. No finite frequency bending modes are obtained; that requires a discretization of two or more elements.

§15.2.4 MELT Definition

The 2D beam-column element is entered in the Master Element Library Table (`MELT`) described in Chapter 20, by the statement

```
MELT = DefineElementType[MELT, "BeamC2D.2",
      {"STM", 1, "PBEAC1", "MOM", "SEG", 1000, {110001}}];
```

The element type name is `"BeamCol2D.2"`. The descriptor fields specify:

`"STM"`	element is of structural mechanics type
`1`	intrinsic dimensionality is one
`"PBEAC1"`	mathematical model: C^1 plane beam-column
`"MOM"`	formulation is by Mechanics of Materials
`"SEG"`	geometry is line segment
`1000`	element has corner nodes only
`{110001}`	initialized freedom signature for corner nodes

ASEN 4519/5519 Sec IV Finite Element Programming with *Mathematica*

Homework Assignment for Chapters 1-2

Due Monday September 22, 1997

See back of this page for homework guidelines and course material access.

EXERCISE 1

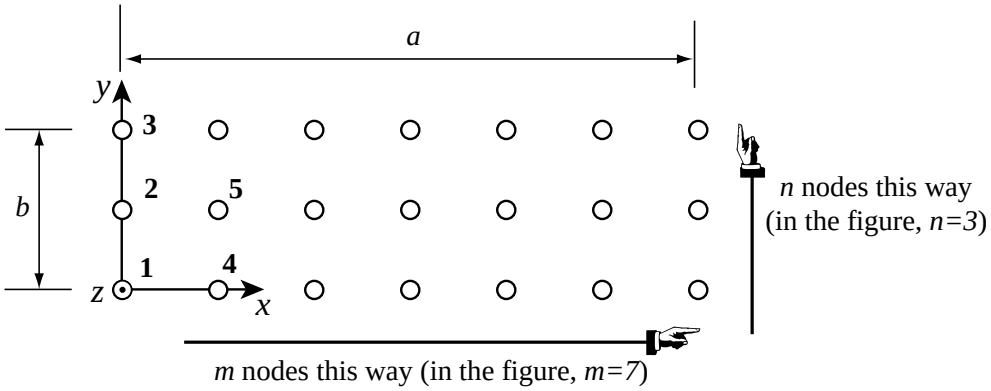
Read Chapter 1, concentrating on the description of data structures. Explain why a list, as used for example in Mathematica, is more general than a Fortran or C array.

EXERCISE 2 *Note: Do this one before Monday Sep 14 to show me how it works at the ITLL during class*

Download from the Web (see homepage address on the back) the Notebook files for Chapter 2.* Execute the programs in Cells 2.1, 2.3, and 2.5 of the Notes, which are on separate Notebook cells, and verify that the answer is that given in Cells 2.2, 2.4 and 2.6. [Don't pay attention to 2.7-2.8]. As answer to this exercise document any problems or glitches you may have experienced in the process.

EXERCISE 3

Write a Mathematica module `GenerateTwoDimNodeLattice[a,b,m,n]` that generates the Master Node Definition Table (MNDT) for a regular $m \times n$ two-dimensional lattice grid that looks like



Place the coordinate axes and number the nodes as illustrated in the diagram. Once the module works, run it for $a = 18$, $b = 6$, $m = 7$, $n = 3$ and print the `MNDT` by calling `PrintMasterNodeDefinitionTable`.

Then write a module `PlotTwoDimNodeLattice[MNDT]` that plots the grid by marking the location of the nodes as small circles, as in the figure above. [Use of `ListPlot` is recommended]. Run it to produce a grid plot for the previous numerical values.

OVER

---

* The format of Notebooks for Mathematica 2.2 and 3.0 is incompatible, although version 3.0 can convert 2.2 Notebooks. Since some of you may have version 2.2 on your PC or Mac, both formats will be posted on the Web.

HOMEWORK GUIDELINES

Normally one assignment will be given every two weeks on Mondays, and it will be due two weeks from the assignment date.

Staple HW solutions securely, and write your initials on each page. *Attach the HW assignment page as cover, with full name(s).*

GROUP HOMEWORK IS ENCOURAGED. The group may contain 2 or 3 students. Homework done as a group will be graded as a group. Submit only one solution per group, with everybody's name on it.

LATE HOMEWORKS WILL NOT BE ACCEPTED FOR CREDIT. This is a consequence of the fact that group homework is the rule. If you expect to miss a class when a HW is due, please make appropriate arrangements with the instructor for delivering the HW on time. If you are in a group, this situation should rarely happen.

PRESENTATION OF COMPUTER RESULTS. Results from computer work should be submitted on 8.5 x 11 printed pages stapled to the rest of the homework. Because all input and output for the computer coursework will be stored in *Mathematica* Notebook files, you will need to get familiar with procedures for printing selected cells and graphics. If, as often happens, the expected output is voluminous the HW assignment will specify which part is essential for presentation.

ACCESSING THE COURSE MATERIAL

All course material, including homework assignments and solutions, will be posted on the Web at the homepage:

```
http://caswww.colorado.edu/courses.d/MIFEM.d/Home.html
```

Click on the appropriate entry. You will need a PDF viewer, for example that supplied with Netscape 3.0, to view and print the files.

Homework assignments will also be posted on the Web, with links from the above home page.

ASEN 4519/5519 Sec IV Finite Element Programming with *Mathematica*

Homework Assignment for Chapters 3-6

Due Monday October 6, 1997

EXERCISE 1

Write a couple of paragraphs explaining the `MEDT` and the `MELT`, and what kind of information is stored there. Details regarding element descriptors as well as constitutive and fabrication properties need not be included.

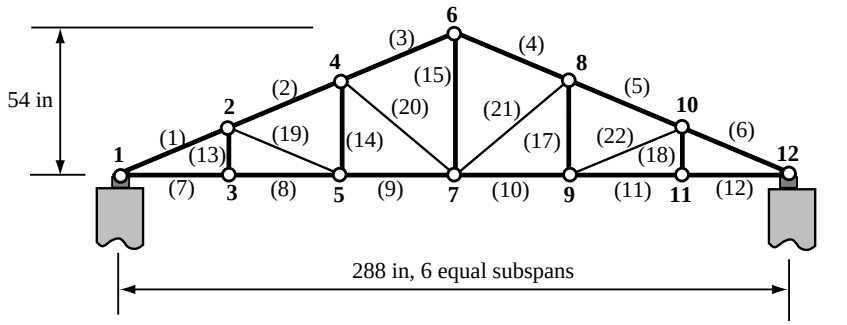
EXERCISE 2

Download from the Web the Notebook files for Chapters 4, 5 and 6.* Execute the programs in the input Cells of Chapters 4, 5 and 6 of the Notes, which are on separate Notebook cells, and verify that the answer is that given in the output Cells. Report by e-mail any discrepancies or difficulties.

EXERCISE 3

Write a Mathematica program that defines the nodes and elements of the plane roof truss depicted in Figure 1, and print the `MNDT`, `MEDT` and `MELT` data structures. Get the elastic moduli from manuals. You will need to built a `MELT` that accomododates this element type (the 2-node bar), following the guidelines of the Notes. Call the element type `"BAR2D.2"`.

Geometric and member-fabrication properties of the truss are shown in the Figure. Do not worry about support conditions for now; these will be handled as part of the freedom data (Chapter 7).



Numbers enclosed in parentheses are element numbers. Use "Bar.1" as external name for member (1), etc.

Member fabrication:

1- 12	Douglas fir	rectangular section: 2 in x 6 in
13-18 (struts)	Douglas fir	rectangular section: 2 in x 4 in
19-22 (diagonals)	ASTM A-36 steel	full circular section: 1/2 in diameter

Figure 1. Truss structure for Exercise 3.

OVER

* The format of ".ma" Notebooks for Mathematica 2.2 and ".nb" Notebooks 3.0 is incompatible, although version 3.0 can convert 2.2 Notebooks. Because some of you may have version 2.2 on your PC or Mac, both formats are posted on the Web.

EXERCISE 4

Write a Mathematica module `GenerateTwoDimElementLattice[a,b,me,ne]` that generates the Master Element Definition Table (`MEDT`) for the regular $m_e \times n_e$ mesh of 4-node quadrilateral plane-stress elements illustrated in Figure 2. Call the element type `QUAD4.LAM`. All elements are identical. Identify them as `"QUAD.1"`, `"QUAD.2"` ,etc. Leave the constitutive and fabrication properties in symbolic form, accessible via constitutive and fabrication codes of 1.

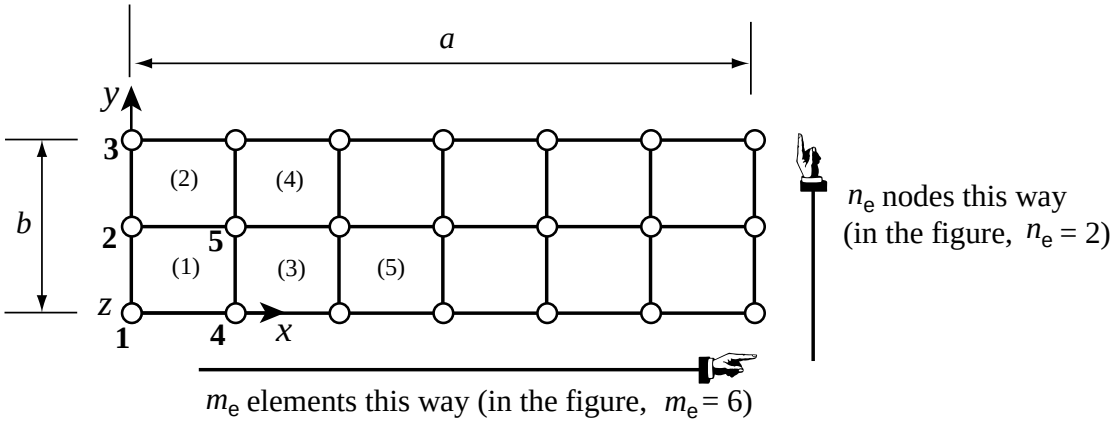


Figure 2. Mesh for Exercise 4.

Place the axes, and number element and nodes as illustrated in the diagram. Once the module works, run it for $a = 18$, $b = 6$, $m_e = 7$, $n_e = 3$ and print the `MEDT` and `MNDT`. [For the latter use the module developed in the previous homework; note that `m=me+1` and `n=ne+1`.] Then write a module `PlotTwoDimElemLattice[MEDT,MELT,MNDT]` that draws the mesh element by element and depicts the element sides nodes. You will need to use *Mathematica* graphic primitives for this task.

ASEN 4519/5519 Sec IV Finite Element Programming with *Mathematica*

Homework Assignment for Chapter 7 and completion of last one

Due Monday October 20, 1997

EXERCISE 0 (not graded)

Download from the Web the Notebook files of Chapter 7. Execute the input Cells, and verify that the answer is that given in the output Cells of the Notes. Report any discrepancies or difficulties.

EXERCISE 1

For the plane truss structure of the second homework, construct realistic Constitutive and Fabrication Tables (use any format you like for the property lists), and include a table printout.

EXERCISE 2

For the plane truss structure of the second homework, reuse the `MNDT` and `MEDT` from that Exercise and the `MFPT` from the above one. Also construct an appropriate `MELT` that includes the definition of a truss element.

Feed into `MNFT=InitializeMasterNodeFreedomTable[MNDT,MEDT,MELT,MFPT]` and print the `MNFT`. All nodes should come up with signature `110000`. If they dont, work backwards to fix the input tables until that happens.

EXERCISE 3

Use `GenerateToDimNodeLattice[a,b,ne+1,me+1]` of the first homework to build the MNDT and `GenerateTwoDimElementLattice[a,b,me,ne]` of the second homework to build the `MEDT` of a regular FE quad mesh for $a = 18$, $b = 6$, $m_e = 7$, $n_e = 3$.

Feed to `MNFT=InitializeMasterNodeFreedomTable[MNDT,MEDT,MELT,MFPT]`. All nodes should come out with freedom signature of `110000`.

Write a module that fixes the two assigned freedoms `tx` and `ty` of the n_e nodes `1,2,... ne+1` of the left end, and prescribes a uniform load of $w = 120$ (force per unit length) acting in the x direction on the n_e nodes of the right end. You will need to compute the node forces by lumping the contributions wb/n_e of each element to those nodes. Verify the settings by printing the `MNFT` and `MNST`.

EXERCISE 4

Investigate whether *Mathematica* could draw symbols near nodes to mark boundary conditions of 2D problems given the `nDL` and `fDL` of that node. For example, symbols shown in the figure below. Think of other possible symbols that may show better and/or may be easier to draw. [Let you imagination run wild].

Translation tx known (two possible symbols)

Force qx known and nonzero (two possible symbols)

Rotation rx known (two possible symbols)

Moment rx known and nonzero (two possible symbols)

ASEN 4519/5519 Sec IV Finite Element Programming with *Mathematica*

Homework Assignment for Master Stiffness Assembler - Chapter 13

Due Monday November 24, 1997 - Note one week span!

For the following exercises, use the files posted in the Web site for Chapter 13 and related *Mathematica* files.

EXERCISE 1

Test the assembler module for the structure of Figures 13.2-3, and check that it produces the answers obtained by hand computations in §13.2.

EXERCISE 2

As above, for the structure of Figure 13.4, which includes the MFC $u_{y1} = u_{y3}$.

EXERCISE 3

Add a second MFC to the structure of Figure 13.4: $u_{x1} + u_{x2} + u_{x3} + u_{x4} + u_{x5} = 0$, which expresses the fact that the mean horizontal displacement is zero. This constraint connects all nodes. The number of global freedoms increase to 13. The skyline array size increases to $68 + 13 = 81$. Incorporate that constraint in the `MEDT` and `MFCT`, adjust the other connectivity tables accordingly,* run the assembler and report results.

* For this you need to study the inputs of the assembler

ASEN 4519/5519 Sec IV Finite Element Programming with *Mathematica*

Homework Assignment for Running Complete Beam-Truss Problem - Chapter 16

Due Monday December 1, 1997 - Note one week span!

For the following exercises, download the *Mathematica 2.2* or *Mathematica 3.0* Notebook file `CompleteProgBeamTruss.ma` or `CompleteProgBeamTruss.nb` posted in the course Web site as part of Chapter 16.

EXERCISE 1

Comment on the overall program organization, especially that of the last 3 input cells.

EXERCISE 2

Convert to Mathematica 3.0 if necessary, and run the problem by executing the last input cell (all preceding module cells must be initialized).

EXERCISE 3

Comment on the solution output printout produced in the last output cell.

ASEN 4519/5519 Sec IV Finite Element Programming with *Mathematica*

Homework Assignment for Beam-Truss Vibration Analysis - Chapter 17

Due Monday December 8, 1997 - Last Homework Assignment

For the following exercises, download the *Mathematica 2.2* or *Mathematica 3.0* Notebook file `ComplProgBeamTrussVibration.ma` or `ComplProgBeamTrussVibration.nb` posted in the course Web site as part of Chapter 17.

EXERCISE 1

Comment on how the vibration eigenproblem has been set up and solved using the *Mathematica* built-in `Eigensystem` function.

EXERCISE 2

The animation of vibration mode shapes only works if one displays a single mode (mode number 10 in the sample Notebook). If one tried to display 2 modes, say 10 and 11, the sequence of 6 graphic cells gets intermixed when one tries to animate a group. Is there a way to separate the animations?

Take Home Exam

I will consult on the form of the take-home exam to be assigned on December 8th. This may be posted to the Web earlier.