

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>The E-Lib Mesh File Format</b>	<b>2</b>
2.1	The Mesh Definition File . . . . .	3
2.2	The Material File . . . . .	10
2.3	The Domain Decomposition File . . . . .	12
2.4	The Mesh Parameters File . . . . .	13
2.5	The Finite Element Error File . . . . .	14
2.6	The Finite Element Stress File . . . . .	15
2.7	The Element Geometric Definition File . . . . .	17
<b>3</b>	<b>The MESH Structure</b>	<b>18</b>
3.1	The MESH Structure . . . . .	21
3.2	The MAT Structure . . . . .	30
3.3	The COMPMAT1 Structure . . . . .	31
3.4	The DECOMP Structure . . . . .	32
3.5	The MESHPARAM Structure . . . . .	34
3.6	The FEERROR Structure . . . . .	36
3.7	The STRESSES Structure . . . . .	38
3.8	The GEOM Structure . . . . .	40
<b>4</b>	<b>The E-Lib Library</b>	<b>41</b>

# Chapter 1

## Introduction

The *E-Lib User's Guide v1.1* is the manual for the E-Lib functions library and defines the E-Lib mesh data file format. The library comprises two main types of function — those for manipulating mesh data and those of a more general nature for use in any type of program.

This work was motivated by the need to integrate the various finite element, mesh generation, mesh decomposition and plotting programs developed by the SECT Research Group. By imposing the use of common variable names throughout the group's programs and providing a standard library of functions, the process of integration will be greatly eased. The E-Lib library will also cut program development times and improve the portability and readability of source codes. Error checks performed by the E-Lib functions should also improve program robustness.

Therefore the requirements of the E-Lib format were that it should fully describe all features of the meshes previously in use by the group and that it should be simple to implement and the resulting data files easy to understand. It also had to be extendible so that it could accommodate new mesh descriptions but do so in such a way that it remained downwardly compatible with all previous versions. These requirements were best met by a format based on the use of keywords.

# Chapter 2

## The E-Lib Mesh File Format

The E-Lib format supports meshes of a single element type and mixed meshes. The E-Lib mesh file format takes the form of a series of keywords followed by one or more data items. For example,

```
NELEMENTS_TRIANG1    120
```

denotes that the mesh contains 120 elements of type **TRIANG1**.

**IMPORTANT** Most keywords are optional, but those which are declared must be in a strict, predefined order. This order automatically accounts for dependencies between the data.

The format also allows the use of file comments. If the first non-white character on a line is the **#** character, then all remaining text on that line is ignored. Two or more lines can be commented out by enclosing the lines inside a **/\*** and **\*/**. The opening **/\*** must be the first two characters on the first comment line and **\*/** the first two characters on the last comment line. All text on the line following **/\*** and **\*/** is ignored. The **/\*...\*/** comments must not be nested, although they can contain **#** comments. Blank lines are also permitted.

There are currently seven types of data file — the mesh definition (**.mdf**), geometric definition (**.gmf**), materials (**.mat**), domain decomposition (**.dom**), element mesh parameters (**.mpr**), stresses (**.ste**), finite element errors files (**.fee**). The mesh definition file contains the main description of the mesh. The other files describe additional properties of the mesh or they represent a state of the mesh.

The following sections describe the keywords and keyword data for the above data files. Each section contains a table which lists the keywords in the order they must be declared and defines the associated keyword data. The data type — whether it is an integer (i), real (r) or a character string (s) — is also given, where, for example, ‘i 3\*r’ denotes that the data consist of an integer followed by three reals. Units, where relevant, are enclosed in square brackets ([...]).

**IMPORTANT** Keyword data consisting of a single data item must follow the keyword on the same line. For vectors and matrices, each vector component or matrix row must start on a new line and each line must begin with an integer index. Unless otherwise stated, this index is either the vector component number or matrix row index. All keyword data strings must be enclosed in double quotes.

## 2.1 The Mesh Definition File

The mesh definition file (**.mdf**) contains the main description of the mesh.

At present twelve different elements types are supported – five one dimensional (**LINK1**, **LINK2**, **LINK3**, **LINK4**, **LINK5**), five triangular (**TRIANG1**, **TRIANG2**, **TRIANG3**, **TRIANG4**, **TRIANG5**), two quadrilateral (**QUAD1** and **QUAD3**), one tetrahedral (**TETRAH1**) and one three dimensional block element (**BLOCK1**). These element types are defined in Figure 2.1 which shows the order in which the element nodes must be labelled and in Table 2.1 which gives a short description as well. With the exception of the **TRIANG2** and **QUAD3** elements, individual elements have uniform material properties. **TRIANG2** and **QUAD3** elements are composed of layered composite materials referred to as composite materials of type 1.

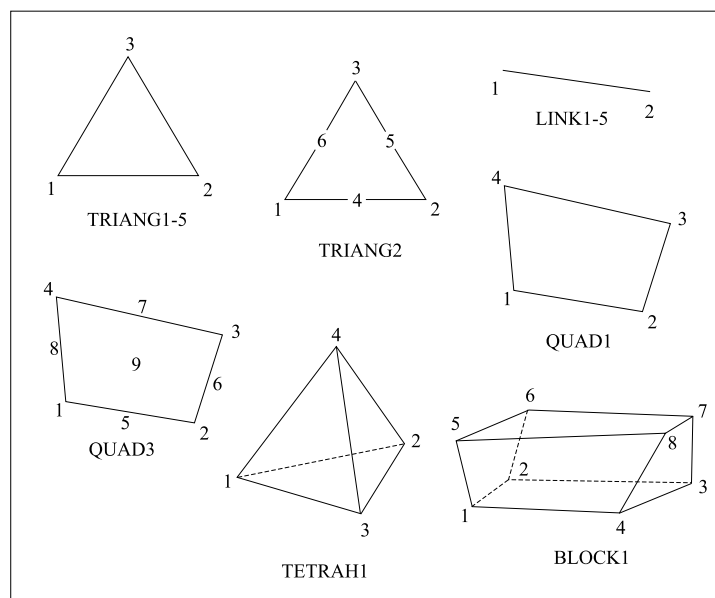


Figure 2.1: The element node orders

An example mesh (Figure 2.2) definition file is shown in the following

Element Type	Description	Number of Vertices	Number of Nodes
<b>LINK1</b>	Truss	2	2
<b>LINK2</b>	Cable	2	2
<b>LINK3</b>	Fixed tension 1-D link	2	2
<b>LINK4</b>	Fixed force density 1-D link	2	2
<b>LINK5</b>	Geodesic string	2	2
<b>TRIANG1</b>	Constant strain triangular	3	3
<b>TRIANG2</b>	A combined constant strain plane stress and constant moment plate bending simple facet triangular	3	6
<b>TRIANG3</b>	Solid triangular	3	3
<b>TRIANG4</b>	Membrane triangular	3	3
<b>TRIANG5</b>	Constant stress triangular	3	3
<b>QUAD1</b>	Plane stress quadrilateral	4	4
<b>QUAD3</b>	Mindlin plate quadrilateral	4	9
<b>TETRAH1</b>	Tetrahedral	4	4
<b>BLOCK1</b>	Block	8	8

Table 2.1: The element types

```
# An example mesh definition file

TITLE "An example mesh"
NMESHPOINTS      6
NNODES           6
NELEMENTS_TRIANG1 4

MESHPOINT_COORDINATES
1  0.000  0.000  0.000
2  3.000  0.000  0.000
3  6.000  0.000  0.000
4  0.000  3.000  0.000
5  3.000  3.000  0.000
6  6.000  3.000  0.000

NODES_TRIANG1
1  1  2  4
2  2  5  4
3  2  3  5
4  3  6  5
```

The first keyword **TITLE**, is compulsory and specifies the title of the mesh. The title can be used to annotate output such as the display in a plot program and, as with all string arguments, must be enclosed in double quotes.

A mesh is defined in terms of a set of *mesh-points* which are joined by straight lines to form the edges or surfaces of the elements.

**IMPORTANT** A distinction is made between the terms *mesh-point*, *vertex* and *finite element node*. A mesh-point is a point in space which may or may not form an element

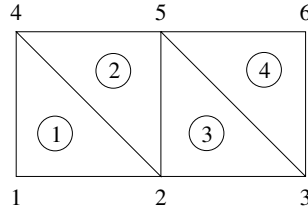


Figure 2.2: An example with TRIANG1 elements

vertex, whereas a finite element node (or simply *node*) is a point on an element used for function interpolation during the finite element analysis. Unlike mesh-points, nodes need not coincide with the element vertices. (See Figure 2.3.)

**NMESHPOINTS** is the keyword for the number of mesh-points defined in the mesh definition file and **NNODES** defines the total number of finite element nodes in the mesh. The coordinates of the mesh-points follow the keyword **MESHPOINT\_COORDS**. All three of the keywords **NMESHPOINTS**, **NNODES** and **MESHPOINT\_COORDS** are compulsory. The keyword **NELEMENTS\_TRIANG1** act both to declare the type of elements in the mesh and to give the number of the element. If this number is zero then the keyword should be omitted. At least one of the keywords which defines the element type must be declared and for each such keyword present there must also be the corresponding keyword like **NODES\_TRIANG1** etc. The data for these keywords define the node indices of each element in turn in the order given in Figure 2.1. Element vertex nodes are always labelled before non-vertex nodes and for two dimensional elements, labelling occurs in an anti-clockwise direction.

Another example is presented to demonstrate the difference between *mesh-point*, *vertex* and *finite element node*. The geometric arrangement of the elements can be seen in Figure 2.3 while the generated mesh file is the following.

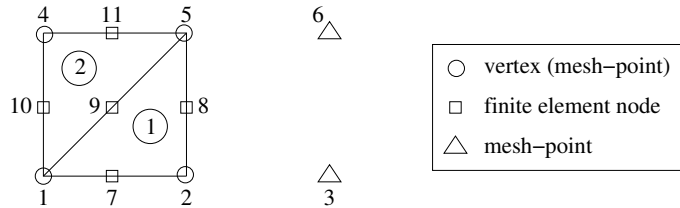


Figure 2.3: An example with TRIANG2 elements

```

# Example with TRIANG 2 elements

TITLE "An example mesh"
NMESHPOINTS      6
NNODES           10
NELEMENTS_TRIANG2 2

MESHPOINT_COORDINATES
  1   0.000   0.000   0.000
  2   3.000   0.000   0.000
  3   6.000   0.000   0.000
  4   0.000   3.000   0.000
  5   3.000   3.000   0.000
  6   6.000   3.000   0.000

NODES_TRIANG2
  1   1       2       5       7       8       9
  2   1       5       4       9      11      10

```

**NOTE** *All other keywords are optional* except in cases where the declaration of one keyword implies that one or more further keywords will be declared later. For example, if the number of boundary condition nodes is defined (**NBOUNDARY\_CONDITION\_NODES**) then so must the nodal boundary conditions (**BOUNDARY\_CONDITIONS**), see example below.

```

# Example with extra keywords

TITLE "An example mesh"
NMESHPOINTS      3
NNODES           3
NELEMENTS_TRIANG1 1
NBOUNDARY_CONDITION_NODES 2

MESHPOINT_COORDINATES
  1   0.000   0.000   0.000
  2   6.000   0.000   0.000
  3   6.000   6.000   0.000

NODES_TRIANG2
  1   1       2       3

BOUNDARY_CONDITIONS
  1   "FIXED" 0.000 "FIXED" 0.000 "FREE"
  2   "FREE"   "FIXED" 0.000 "FREE"

```

An exhaustive list of possible keywords in an **.mdf** file is presented in Table 2.2a,b,c .

For the use of keywords not discussed here see the relevant section in the documentation.

Keyword	Keyword Data	Data Type
<b>TITLE</b>	Mesh title	s
<b>NMESHPOINTS</b>	Number of mesh-points	i
<b>NNODES</b>	Number of nodes	i
<b>NELEMENTS_LINK1</b>	Number of <b>LINK1</b> elements	i
<b>NELEMENTS_LINK2</b>	Number of <b>LINK2</b> elements	i
<b>NELEMENTS_LINK3</b>	Number of <b>LINK3</b> elements	i
<b>NELEMENTS_LINK4</b>	Number of <b>LINK4</b> elements	i
<b>NELEMENTS_LINK5</b>	Number of <b>LINK5</b> elements	i
<b>NELEMENTS_TRIANG1</b>	Number of <b>TRIANG1</b> elements	i
<b>NELEMENTS_TRIANG2</b>	Number of <b>TRIANG2</b> elements	i
<b>NELEMENTS_TRIANG3</b>	Number of <b>TRIANG3</b> elements	i
<b>NELEMENTS_TRIANG4</b>	Number of <b>TRIANG4</b> elements	i
<b>NELEMENTS_TRIANG5</b>	Number of <b>TRIANG5</b> elements	i
<b>NELEMENTS_QUAD1</b>	Number of <b>QUAD1</b> elements	i
<b>NELEMENTS_QUAD3</b>	Number of <b>QUAD3</b> elements	i
<b>NELEMENTS_TETRAH1</b>	Number of <b>TETRAH1</b> elements	i
<b>NELEMENTS_BLOCK1</b>	Number of <b>BLOCK1</b> elements	i
<b>NBOUNDARY_CONDITION_NODES</b>	Number of boundary condition nodes	i
<b>NLOADED_NODES</b>	Number of loaded nodes	i
<b>NMATERIALS</b>	Total number of materials including those declared in composite materials	i
<b>NCOMP_MATERIALS_TYPE1</b>	Number of type 1 composite materials	i
<b>NNURBS_CURVES</b>	Number of NURBS curves	i
<b>TIMESTEP</b>	The value of the timestep	r
<b>NTIMESTEPS</b>	Number of time-steps	i
	This keyword and the pair of keywords <b>NINTERNAL_TIMESTEPS</b> and <b>NEXTERNAL_TIMESTEPS</b> are mutually exclusive	
<b>NINTERNAL_TIMESTEPS</b>	Number internal and external time-steps	i
<b>NEXTERNAL_TIMESTEPS</b>	These data enable time-stepping to be broken down into a series of <i>NExternalTimesteps</i> sets of <i>NinternalTimesteps</i> time-steps (See <b>NTIMESTEPS</b> )	
<b>DAMPING_FACTOR</b>	Viscous damping factor	r
<b>BETA</b>	Newmark's $\beta$ integration constant	r
<b>GAMMA</b>	Newmark's $\gamma$ integration constant	r

Table 2.2: (a) The mesh definition file keywords and keyword data (cont.)



Keyword	Keyword Data	Data Type
<b>NSTRESS_POINTS_LINK1</b>	Number of stress points in a <b>LINK1</b> element	i
<b>NSTRESS_POINTS_LINK2</b>	Number of stress points in a <b>LINK2</b> element	i
<b>NSTRESS_POINTS_LINK3</b>	Number of stress points in a <b>LINK3</b> element	i
<b>NSTRESS_POINTS_LINK4</b>	Number of stress points in a <b>LINK4</b> element	i
<b>NSTRESS_POINTS_LINK5</b>	Number of stress points in a <b>LINK5</b> element	i
<b>NSTRESS_POINTS_TRIANG1</b>	Number of stress points in a <b>TRIANG1</b> element	i
<b>NSTRESS_POINTS_TRIANG2</b>	Number of stress points in a <b>TRIANG2</b> element	i
<b>NSTRESS_POINTS_TRIANG3</b>	Number of stress points in a <b>TRIANG3</b> element	i
<b>NSTRESS_POINTS_TRIANG4</b>	Number of stress points in a <b>TRIANG4</b> element	i
<b>NSTRESS_POINTS_TRIANG5</b>	Number of stress points in a <b>TRIANG5</b> element	i
<b>NSTRESS_POINTS_QUAD1</b>	Number of stress points in a <b>QUAD1</b> element	i
<b>NSTRESS_POINTS_QUAD3</b>	Number of stress points in a <b>QUAD3</b> element	i
<b>NSTRESS_POINTS_TETRAH1</b>	Number of stress points in a <b>TETRAH1</b> element	i
<b>NSTRESS_POINTS_BLOCK1</b>	Number of stress points in a <b>BLOCK1</b> element	i
<b>MESHPPOINT_COORDS</b>	x-,y- and z-coordinates of the mesh-point	i 3*r
<b>NODES_LINK1</b>	Node indices of each <b>LINK1</b> element	i 2*i
<b>NODES_LINK2</b>	Node indices of each <b>LINK2</b> element	i 2*i
<b>NODES_LINK3</b>	Node indices of each <b>LINK3</b> element	i 2*i
<b>NODES_LINK4</b>	Node indices of each <b>LINK4</b> element	i 2*i
<b>NODES_LINK5</b>	Node indices of each <b>LINK5</b> element	i 2*i
<b>NODES_TRIANG1</b>	Node indices of each <b>TRIANG1</b> element	i 3*i
<b>NODES_TRIANG2</b>	Node indices of each <b>TRIANG2</b> element	i 6*i
<b>NODES_TRIANG3</b>	Node indices of each <b>TRIANG3</b> element	i 3*i
<b>NODES_TRIANG4</b>	Node indices of each <b>TRIANG4</b> element	i 3*i
<b>NODES_TRIANG5</b>	Node indices of each <b>TRIANG5</b> element	i 3*i
<b>NODES_QUAD1</b>	Node indices of each <b>QUAD1</b> element	i 4*i
<b>NODES_QUAD3</b>	Node indices of each <b>QUAD3</b> element	i 9*i
<b>NODES_TETRAH1</b>	Node indices of each <b>TETRAH1</b> element	i 4*i
<b>NODES_BLOCK1</b>	Node indices of each <b>BLOCK1</b> element	i 8*i
<b>BOUNDARY_CONDITIONS</b>	Nodal boundary conditions  For each boundary condition node, the node index followed by, for each degree of freedom, the string " <b>FREE</b> " if no boundary conditions are imposed, " <b>FIXED</b> " followed by a real displacement [m], or " <b>SPRING</b> " followed by a non-negative spring-constant [Nm <sup>-1</sup> ]. The displacement argument specifies an initial displacement of a node. The spring constant enables elastic forces to be modelled.	←

Table 2.2: (b) The mesh definition file keywords and keyword data (cont.)

Keyword	Keyword Data	Data Type
<b>LOADS</b>	Nodal loads For each loaded node, the node index followed by the applied load for each degree of freedom	i r
<b>MATERIALS.LINK1</b>	Material name of each <b>LINK1</b> element	i s
<b>MATERIALS.LINK2</b>	Material name of each <b>LINK2</b> element	i s
<b>MATERIALS.LINK3</b>	Material name of each <b>LINK3</b> element	i s
<b>MATERIALS.LINK4</b>	Material name of each <b>LINK4</b> element	i s
<b>MATERIALS.LINK5</b>	Material name of each <b>LINK5</b> element	i s
<b>MATERIALS.TRIANG1</b>	Material name of each <b>TRIANG1</b> element	i s
<b>COMP_MATERIALS.TRIANG2</b>	Composite material name of each <b>TRIANG2</b> element	i s
<b>MATERIALS.TRIANG3</b>	Material name of each <b>TRIANG3</b> element	i s
<b>MATERIALS.TRIANG4</b>	Material name of each <b>TRIANG4</b> element	i s
<b>MATERIALS.TRIANG5</b>	Material name of each <b>TRIANG5</b> element	i s
<b>MATERIALS.QUAD1</b>	Material name of each <b>QUAD1</b> element	i s
<b>COMP_MATERIALS.QUAD3</b>	Composite material name of each <b>QUAD3</b> element	i s
<b>MATERIALS.TETRAH1</b>	Material name of each <b>TETRAH1</b> element	i s
<b>MATERIALS.BLOCK1</b>	Material name of each <b>BLOCK1</b> element	i s
<b>REMESH_DATA</b>	Defines the beginning of NURBS definition	

Table 2.2: (c) The mesh definition file keywords and keyword data

## 2.2 The Material File

The materials file (**.mat**) contains the properties of the material and composite material declared in the mesh definition file. Any number of materials can be defined and not just those used by the current mesh. This enables a number of different meshes to use a single materials file. The materials and composite materials can be defined in any order.

Each definition of material properties begins with the keyword **MATERIAL** and ends with the keyword **END**. Type 1 composite material properties begin with the keyword **COMP\_MATERIAL\_TYPE1** and must also end with the keyword **END**. Tables 2.3 and 2.4 define the property keywords.

Keyword	Keyword Data	Data Type
<b>MATERIAL</b>	Material name	s
<b>MAT.TYPE</b>	Material model index	i
<b>DENSITY</b>	Density	r
<b>YMOD</b>	Isotropic Young's modulus This keyword and the x-,y- and z-direction Young's moduli keywords are mutually exclusive	r
<b>YMOD_X</b>	x-direction Young's modulus	r
<b>YMOD_Y</b>	y-direction Young's modulus	r
<b>YMOD_Z</b>	z-direction Young's modulus	r
<b>THICKNESS</b>	Thickness	r
<b>POISSONS_RATIO</b>	Poisson ratio	r
<b>IPARAM<math>n</math></b>	$n$ -th integer property, $n = 1, \dots, 6$	i
<b>DPARAM<math>n</math></b>	$n$ -th double precision property, $n = 1, \dots, 20$	r
<b>END</b>		

Table 2.3: The material property keywords and keyword data

Keyword	Keyword Data	Data Type
<b>COMP_MATERIAL_TYPE1</b>	Type 1 composite material name	s
<b>NLAYERS</b>	Number of layers	i
<b>LAYER_MATERIALS</b>	Material name of each layer	i s
<b>LAYER_THICKNESSES</b>	Thickness of each layer	i r
<b>END</b>		

Table 2.4: The type 1 composite material property keywords and keyword data

The structure offers a possibility to introduce new material properties which are not listed among the keywords. For integer values **IPARAM $n$**  and for real values **DPARAM $n$**  should be used.

**IMPORTANT** When new material property is defined, do not forget to document that which parameters correspond to which material property.

In the case of the composite material the thicknesses can have any value and their sum is not checked.

An example **.mdf** with materials would be the following.

```
# Example with TRIANG 2 elements and materials

TITLE "An example mesh"
NMESHPOINTS      6
NNODES           10
NELEMENTS_TRIANG2  2
NMATERIALS       2
NCOMP_MATERIALS_TYPE1 1

MESHPOINT_COORDINATES
  1  0.000  0.000  0.000
  2  3.000  0.000  0.000
  3  6.000  0.000  0.000
  4  0.000  3.000  0.000
  5  3.000  3.000  0.000
  6  6.000  3.000  0.000

NODES_TRIANG2
  1  1      2      5      7      8      9
  2  1      5      4      9     11     10

COMP_MATERIALS_TRIANG2
  1  "layered_material"
  2  "layered_material"
```

While the corresponding **.mat** file looks like this.

```
MATERIAL  "substance1"
DENSITY   2000.0
YMOD      210000.0
# Yielding stress
DPARAM1   20
END

MATERIAL  "substance2"
DENSITY   1000.0
YMOD      50000.0
# Yielding stress
DPARAM1   10
END

COMP_MATERIAL_TYPE1  "layered_material"
NLAYERS  2
LAYER_MATERIALS
  1  "substance1"
  2  "substance2"
LAYER_THICKNESSES
  1  0.12
  2  0.5
END
```

## 2.3 The Domain Decomposition File

The domain decomposition file (**.dom**) specifies the subdomains into which the elements have been partitioned (Table 2.5). The keywords **NSUBDOMAINS** and **SUBDOMAINS\_TRIANG1** etc. specify the number of subdomains to be created and the partition indices for the various element types. The indices must lie between 1 and the number of subdomains inclusive and the number of each element type must be consistent with the number of that type defined in the mesh definition file.

Keyword	Keyword Data	Data Type
<b>NSUBDOMAINS</b>	Number of subdomains	i
<b>SUBDOMAINS_LINK1</b>	Subdomain index of each <b>LINK1</b> element	i i
<b>SUBDOMAINS_LINK2</b>	Subdomain index of each <b>LINK2</b> element	i i
<b>SUBDOMAINS_LINK3</b>	Subdomain index of each <b>LINK3</b> element	i i
<b>SUBDOMAINS_LINK4</b>	Subdomain index of each <b>LINK4</b> element	i i
<b>SUBDOMAINS_LINK5</b>	Subdomain index of each <b>LINK5</b> element	i i
<b>SUBDOMAINS_TRIANG1</b>	Subdomain index of each <b>TRIANG1</b> element	i i
<b>SUBDOMAINS_TRIANG2</b>	Subdomain index of each <b>TRIANG2</b> element	i i
<b>SUBDOMAINS_TRIANG3</b>	Subdomain index of each <b>TRIANG3</b> element	i i
<b>SUBDOMAINS_TRIANG4</b>	Subdomain index of each <b>TRIANG4</b> element	i i
<b>SUBDOMAINS_TRIANG5</b>	Subdomain index of each <b>TRIANG5</b> element	i i
<b>SUBDOMAINS_QUAD1</b>	Subdomain index of each <b>QUAD1</b> element	i i
<b>SUBDOMAINS_QUAD3</b>	Subdomain index of each <b>QUAD3</b> element	i i
<b>SUBDOMAINS_TETRAH1</b>	Subdomain index of each <b>TETRAH1</b> element	i i
<b>SUBDOMAINS_BLOCK1</b>	Subdomain index of each <b>BLOCK1</b> element	i i

Table 2.5: The domain decomposition file keywords and keyword data

An example domain decomposition file is shown for Figure 2.2

```
# An example domain decomposition file
NSUBDOMAINS 2

SUBDOMAINS_TRIANG1
1 1
2 1
3 2
4 2
```

## 2.4 The Mesh Parameters File

The mesh parameters file (**.mpr**) defines the element mesh parameters of each element according to its element type (Table 2.6). The number of mesh parameters for each element type must equal the number of elements defined in the mesh definition file.

There is a possibility to define the mesh parameters on a nodal basis. In this case the **NODAL\_MESHPARAMS** keyword should be defined and then the mesh parameter for each mesh-point.

Keyword	Keyword Data	Data Type
<b>NODAL_MESHPARAMS</b>	Mesh parameters for each node	i r
<b>MESHPARAMS_LINK1</b>	Element mesh parameter of each <b>LINK1</b> element	i r
<b>MESHPARAMS_LINK2</b>	Element mesh parameter of each <b>LINK2</b> element	i r
<b>MESHPARAMS_LINK3</b>	Element mesh parameter of each <b>LINK3</b> element	i r
<b>MESHPARAMS_LINK4</b>	Element mesh parameter of each <b>LINK4</b> element	i r
<b>MESHPARAMS_LINK5</b>	Element mesh parameter of each <b>LINK5</b> element	i r
<b>MESHPARAMS_TRIANG1</b>	Element mesh parameter of each <b>TRIANG1</b> element	i r
<b>MESHPARAMS_TRIANG2</b>	Element mesh parameter of each <b>TRIANG2</b> element	i r
<b>MESHPARAMS_TRIANG3</b>	Element mesh parameter of each <b>TRIANG3</b> element	i r
<b>MESHPARAMS_TRIANG4</b>	Element mesh parameter of each <b>TRIANG4</b> element	i r
<b>MESHPARAMS_TRIANG5</b>	Element mesh parameter of each <b>TRIANG5</b> element	i r
<b>MESHPARAMS_QUAD1</b>	Element mesh parameter of each <b>QUAD1</b> element	i r
<b>MESHPARAMS_QUAD3</b>	Element mesh parameter of each <b>QUAD3</b> element	i r
<b>MESHPARAMS_TETRAH1</b>	Element mesh parameter of each <b>TETRAH1</b> element	i r
<b>MESHPARAMS_BLOCK1</b>	Element mesh parameter of each <b>BLOCK1</b> element	i r

Table 2.6: The element mesh parameters file keywords and keyword data

An example element mesh parameters file is shown for Figure 2.2

```
# An example element mesh parameters file

MESHPARAMS_TRIANG1
1 5.3456
2 3.4557
3 3.4556
4 0.456
```

## 2.5 The Finite Element Error File

The finite element errors are stored in the finite element error file (**.fee**). One error value must be defined for each element (Table 2.7). Even in the case of **TRIANG2** and **QUAD3** elements which are layered finite elements only one value should be defined for one element.

Keyword	Keyword Data	Data Type
<b>FEERRORS_LINK1</b>	Finite element error of each <b>LINK1</b> element	i r
<b>FEERRORS_LINK2</b>	Finite element error of each <b>LINK2</b> element	i r
<b>FEERRORS_LINK3</b>	Finite element error of each <b>LINK3</b> element	i r
<b>FEERRORS_LINK4</b>	Finite element error of each <b>LINK4</b> element	i r
<b>FEERRORS_LINK5</b>	Finite element error of each <b>LINK5</b> element	i r
<b>FEERRORS_TRIANG1</b>	Finite element error of each <b>TRIANG1</b> element	i r
<b>FEERRORS_TRIANG2</b>	Finite element error of each <b>TRIANG2</b> element	i r
<b>FEERRORS_TRIANG3</b>	Finite element error of each <b>TRIANG3</b> element	i r
<b>FEERRORS_TRIANG4</b>	Finite element error of each <b>TRIANG4</b> element	i r
<b>FEERRORS_TRIANG5</b>	Finite element error of each <b>TRIANG5</b> element	i r
<b>FEERRORS_QUAD1</b>	Finite element error of each <b>QUAD1</b> element	i r
<b>FEERRORS_QUAD3</b>	Finite element error of each <b>QUAD3</b> element	i r
<b>FEERRORS_TETRAH1</b>	Finite element error of each <b>TETRAH1</b> element	i r
<b>FEERRORS_BLOCK1</b>	Finite element error of each <b>BLOCK1</b> element	i r

Table 2.7: The finite element error file keywords and keyword data

An example finite element error file is shown for Figure 2.2

```
# An example finite element error file

FEERRORS_TRIANG1
1 0.0021
2 0.0032
3 0.0123
4 0.0001
```

## 2.6 The Finite Element Stress File

The element stresses are defined in the stress file (**.ste**). The number of stress points for each element must be defined in the **.mdf** file with the corresponding keyword, like **NSTRESS\_POINTS\_LINK1**, etc. For each stress point a line is defined consisting of 6 components, e.g. normal stresses in the x, y and z directions, the three shear stresses. Other interpretation is possible as well, e.g. the first three components store the principal stresses while the other three components store the angles of the principal directions. Any of the components can be ignored, e.g. in a plane problem only the first three components will be used, as the two normal stresses and a shear stress, or in the case of truss elements only one component is used.

**IMPORTANT** In the case of **TRIANG2** and **QUAD3** elements the stresses are defined at the stress points per layer per element, therefore it is a requirement that the material file is available for these elements. For all other elements the existence of the material file is not necessary to load/write stresses.

Table 2.8 shows the possible keywords.

Keyword	Keyword Data	Data Type
<b>STRESSES_LINK1</b>	6 available components of stresses	i i 6*r
<b>STRESSES_LINK2</b>	6 available components of stresses	i i 6*r
<b>STRESSES_LINK3</b>	6 available components of stresses	i i 6*r
<b>STRESSES_LINK4</b>	6 available components of stresses	i i 6*r
<b>STRESSES_LINK5</b>	6 available components of stresses	i i 6*r
<b>STRESSES_TRIANG1</b>	6 available components of stresses	i i 6*r
<b>STRESSES_TRIANG2</b>	6 available components of stresses	i i i 6*r
<b>STRESSES_TRIANG3</b>	6 available components of stresses	i i 6*r
<b>STRESSES_TRIANG4</b>	6 available components of stresses	i i 6*r
<b>STRESSES_TRIANG5</b>	6 available components of stresses	i i 6*r
<b>STRESSES_QUAD1</b>	6 available components of stresses	i i 6*r
<b>STRESSES_QUAD3</b>	6 available components of stresses	i i i 6*r
<b>STRESSES_TETRAH1</b>	6 available components of stresses	i i 6*r
<b>STRESSES_BLOCK1</b>	6 available components of stresses	i i 6*r

Table 2.8: The stress file keywords and keyword data

An example finite element stress file is shown for Figure 2.2



```

# An example stress file

# Triangl stresses
# element index
#   Stress point index
#       Ssigma x      Ssigma y      Ssigma z      Tau xy      Tau yz      Tau zx
#-----
STRESSES_TRIANG1
  1  1      1.000e+00    2.000e+00    3.000e+00    4.000e+00    5.000e+00    6.000e+00
  1  2      7.000e+00    8.000e+00    9.000e+00   10.000e+00   11.000e+00   12.000e+00
  2  1     13.000e+00   14.000e+00   15.000e+00   16.000e+00   17.000e+00   18.000e+00
  2  2     19.000e+00   20.000e+00   21.000e+00   22.000e+00   23.000e+00   24.000e+00
  3  1     25.000e+00   26.000e+00   27.000e+00   28.000e+00   29.000e+00   30.000e+00
  3  2     31.000e+00   32.000e+00   33.000e+00   34.000e+00   35.000e+00   36.000e+00
  4  1     37.000e+00   38.000e+00   39.000e+00   40.000e+00   41.000e+00   42.000e+00
  4  2     43.000e+00   44.000e+00   45.000e+00   46.000e+00   47.000e+00   48.000e+00

```

## 2.7 The Element Geometric Definition File

The element geometric definition file **.gmf** defines the boundary of the finite element problem domain using Non-Uniform Rational B-Splines (NURBS). It also defines how the idealisation is restrained or constrained.

For further information please see, the “**MESHGEN USER MANUAL**”.

# Chapter 3

## The MESH Structure

The E-Lib standard stores mesh data in a C structure called **MESH**. This results in very simple E-Lib function declarations and enables items to be added to the mesh description without the need for altering pre-existing programs which use the library. Table 3.1, 3.2 and 3.3 shows the definition of the **MESH** structure.

```
typedef struct
{
    char *Title;                /* title */

    int NMeshPoints;            /* number of mesh-points */
    int NNodes;                 /* number of finite element nodes */
    int NElemsLink1;            /* number of LINK1 elements */
    int NElemsLink2;            /* number of LINK2 elements */
    int NElemsLink3;            /* number of LINK3 elements */
    int NElemsLink4;            /* number of LINK4 elements */
    int NElemsLink5;            /* number of LINK5 elements */
    int NElemsTriang1;           /* number of TRIANG1 elements */
    int NElemsTriang2;           /* number of TRIANG2 elements */
    int NElemsTriang3;           /* number of TRIANG3 elements */
    int NElemsTriang4;           /* number of TRIANG4 elements */
    int NElemsTriang5;           /* number of TRIANG5 elements */
    int NElemsQuad1;            /* number of QUAD1 elements */
    int NElemsQuad3;            /* number of QUAD3 elements */
    int NElemsTetra1;           /* number of TETRAH1 elements */
    int NElemsBlock1;           /* number of BLOCK1 elements */
    int TotalNElems;            /* total number of elements */
    int NBCNodes;               /* number of boundary condition nodes */
    int NLoadedNodes;           /* number of loaded nodes */
    int NMats;                  /* number of materials */
    int NMdfMats;               /* number of .mdf declared materials */
    int NCompMatsType1;         /* number of type 1 composite materials */
    int NTimeSteps;             /* number of time-steps */
    int NInternalTimeSteps;     /* number of internal time-steps */
    int NExternalTimeSteps;     /* number of external time-steps */
    int FirstNetIndex[E_NELEM_TYPES]; /* first net index of each element type */
}
```

Table 3.1: (a) The **MESH** structure

```

double TimeStep;          /* time-step */
double DampingFactor;     /* viscous damping factor */
double Beta;              /* Newmark's beta integration constant */
double Gamma;             /* Newmark's gamma integration constant */

int NStressPointsLink1;   /* number of stress points in LINK1 */
int NStressPointsLink2;   /* number of stress points in LINK2 */
int NStressPointsLink3;   /* number of stress points in LINK3 */
int NStressPointsLink4;   /* number of stress points in LINK4 */
int NStressPointsLink5;   /* number of stress points in LINK5 */
int NStressPointsTriang1; /* number of stress points in TRIANG1 */
int NStressPointsTriang2; /* number of stress points in TRIANG2 */
int NStressPointsTriang3; /* number of stress points in TRIANG3 */
int NStressPointsTriang4; /* number of stress points in TRIANG4 */
int NStressPointsTriang5; /* number of stress points in TRIANG5 */
int NStressPointsQuad1;   /* number of stress points in QUAD1 */
int NStressPointsQuad3;   /* number of stress points in QUAD3 */
int NStressPointsTetra1; /* number of stress points in TETRAH1 */
int NStressPointsBlock1;  /* number of stress points in BLOCK1 */

double **MeshPointCoords; /* mesh-point coordinates */

int **NodesLink1;         /* LINK1 node indices */
int **NodesLink2;         /* LINK2 node indices */
int **NodesLink3;         /* LINK3 node indices */
int **NodesLink4;         /* LINK4 node indices */
int **NodesLink5;         /* LINK5 node indices */
int **NodesTriang1;       /* TRIANG1 node indices */
int **NodesTriang2;       /* TRIANG2 node indices */
int **NodesTriang3;       /* TRIANG3 node indices */
int **NodesTriang4;       /* TRIANG4 node indices */
int **NodesTriang5;       /* TRIANG5 node indices */
int **NodesQuad1;         /* QUAD1 node indices */
int **NodesQuad3;         /* QUAD3 node indices */
int **NodesTetra1;        /* TETRAH1 node indices */
int **NodesBlock1;        /* BLOCK1 node indices */

int TotalNodalDOF;        /* total nodal degrees of freedom */
int *NodalDOF;            /* nodal degrees of freedom */

int *BCNodes;             /* boundary condition nodes */
int **BCTypes;            /* boundary condition types */
double **BCDispls;        /* initial nodal displacements */
double **BCSpringConsts;  /* spring constants */

int *LoadedNodes;         /* loaded nodes */
double **Loads;           /* loads */

```

Table 3.2: (b) The **MESH** structure

```

void **Mats; /* materials */
MAT **MatsLink1; /* LINK1 materials */
MAT **MatsLink2; /* LINK2 materials */
MAT **MatsLink3; /* LINK3 materials */
MAT **MatsLink4; /* LINK4 materials */
MAT **MatsLink5; /* LINK5 materials */
MAT **MatsTriang1; /* TRIANG1 materials */
COMPMAT1 **CompMatsTriang2; /* TRIANG2 composite materials */
MAT **MatsTriang3; /* TRIANG3 materials */
MAT **MatsTriang4; /* TRIANG4 materials */
MAT **MatsTriang5; /* TRIANG5 materials */
MAT **MatsQuad1; /* QUAD1 materials */
COMPMAT1 **CompMatsQuad3; /* QUAD3 composite materials */
MAT **MatsTetra1; /* TETRAH1 materials */
MAT **MatsBlock1; /* BLOCK1 materials */

int *NodeTypes; /* remeshing node types */
int *NNurbsCurves; /* number of NURBS curves per node */
double **NurbsCurveParams; /* NURBS curve parameters */
NURBS_CURVE ***NurbsCurves; /* node NURBS curves */

int *GNIs; /* global node indices */
int *GEIs; /* global element indices */

int *GEIsLink1; /* LINK1 global element indices */
int *GEIsLink2; /* LINK2 global element indices */
int *GEIsLink3; /* LINK3 global element indices */
int *GEIsLink4; /* LINK4 global element indices */
int *GEIsLink5; /* LINK5 global element indices */
int *GEIsTriang1; /* TRIANG1 global element indices */
int *GEIsTriang2; /* TRIANG2 global element indices */
int *GEIsTriang3; /* TRIANG3 global element indices */
int *GEIsTriang4; /* TRIANG4 global element indices */
int *GEIsTriang5; /* TRIANG5 global element indices */
int *GEIsQuad1; /* QUAD1 global element indices */
int *GEIsQuad3; /* QUAD3 global element indices */
int *GEIsTetra1; /* TETRAH1 global element indices */
int *GEIsBlock1; /* BLOCK1 global element indices */

GEOM Geom; /* geometric model */
MAT *Props; /* material properties */
COMPMAT1 *CompPropsType1; /* type 1 composite material properties */
DECOMP Decomp; /* decomposition data */
MESHPARAM MeshParam; /* mesh parameters */
FEERROR FEError; /* finite element errors */
STRESSES Stresses; /* finite element stresses */

int KeywordSet[E_NKEYWORDS]; /* keyword Booleans */
int SubStructSet[E_NSUBSTRUCTS]; /* sub-structure Booleans */
}
MESH;

```

Table 3.3: (c) The **MESH** structure

### 3.1 The MESH Structure

The first member of the **MESH** structure is the character pointer **Title** which points to the mesh title.

The nodes in a mesh are identified by two separate sets of indices. The first set runs from 1 to the number of nodes in the mesh, **NNodes**. However, when a mesh is decomposed into two or more subdomains, the node indices in the resulting submeshes will differ from those of the corresponding nodes in the original mesh. By assigning an integer to each node which remains constant *throughout all decompositions*, the second type of indexing, known as global node indexing, enables a track to be kept on which nodes lie in which subdomain. The global index of a node is its index as defined in the original mesh. Global indices are stored in the member vector **GNI**s.

When numbering the nodes in a mesh, the first **NMeshPoints** nodes (1, ..., **NMeshPoints**) must be element vertices. The indices for each element are stored in the order indicated in Figure 2.1 and according to the element type. For example,

```
mesh->NodesTriang1[i - 1][j - 1]
```

is the node index of the  $j$ -th vertex of the  $i$ -th **TRIANG1** element, where  $i = 1, \dots, \mathbf{NElemsTriang1}$  and  $j = 1, \dots, \mathbf{E\_NNODES\_TRIANG1}$ . The values of the integers **E\_NNODES\_TRIANG1** etc. are defined in `<e_lib.h>`.

**EXAMPLE** (For functions used here but not discussed see the relevant reference pages.) Lets read in the following mesh definition file (**test.mdf**)

```
# An example mesh definition file

TITLE "An example mesh"
NMESHPOINTS      6
NNODES           6
NELEMENTS_TRIANG1 4

MESHPOINT_COORDINATES
  1  0.000  0.000  0.000
  2  3.000  0.000  0.000
  3  6.000  0.000  0.000
  4  0.000  3.000  0.000
  5  3.000  3.000  0.000
  6  6.000  3.000  0.000

NODES_TRIANG1
  1  1  2  4
  2  2  5  4
  3  2  3  5
  4  3  6  5
```

and the corresponding subdomain file (**test.dom**)

```
# An example domain decomposition file
NSUBDOMAINS 2

SUBDOMAINS_TRIANG1
1 1
2 1
3 2
4 2
```

The code:	The result:
<pre>e_GetMeshData(&amp;mesh, "test"); e_GetDecompData(mesh, "test");  for(i=0; i&lt;mesh-&gt;NElemsTriang1; i++)     printf("\n%d. element %d %d %d", i+1,         mesh-&gt;NodesTriang1[i][0],         mesh-&gt;NodesTriang1[i][1],         mesh-&gt;NodesTriang1[i][2]);  e_CreateSubmesh(mesh,                 &amp;Submesh,                 "2 subdomain", 2)  for(i=0; i&lt;Submesh-&gt;NElemsTriang1; i++)     printf("\n%d. element %d %d %d",         i+1,         Submesh-&gt;NodesTriang1[i][0],         Submesh-&gt;NodesTriang1[i][1],         Submesh-&gt;NodesTriang1[i][2]);  for(i=0; i&lt;Submesh-&gt;NNodes; i++)     printf("\nlocal %d -&gt; global %d",         i+1, Submesh-&gt;GNIs[i]);</pre>	<pre>1. element 1 2 4 2. element 2 5 4 3. element 2 3 5 4. element 3 6 5  1. element 1 2 3 2. element 2 4 3  local 1 -&gt; global 2 local 2 -&gt; global 3 local 3 -&gt; global 5 local 4 -&gt; global 6</pre>

The mesh-point coordinates are stored in the matrix **MeshPointCoords**. Integer macros (**E\_XDIR**, **E\_YDIR** and **E\_ZDIR**) are defined for the x-,y- and z-coordinates. Their use is not necessary, but can make the program more readable.

**EXAMPLE** To access the mesh-point coordinates, the following structure can be used for example:

```
for(i=0; i<mesh->NMeshPoints; i++)
{
    mesh->MeshPointCoords[i][E_XDIR] = 0.00;
    mesh->MeshPointCoords[i][E_YDIR] = 0.00;
    mesh->MeshPointCoords[i][E_ZDIR] = 0.00;
}
```

There also exists more than one way to label the elements:

- In *type indexing* the elements are grouped according to their element type and indexed from 1 to the number of elements of that type. The **MESH** members **NElemsTriang1** etc. store the number of elements of each type in the mesh.
- In *net indexing* the groups of elements are placed ‘end-to-end’ and indexed from 1 to the total number of elements in the mesh, **TotalNElems**. The order of the elements within each group is the same as for type indexing, or as listed in Table 2.2. The member vector **FirstNetIndex** stores the net indices of the first element of each element type, starting from 1. If a certain type of element does not exist in the mesh then its **FirstNetIndex** value is zero. For example, if an element of type **QUAD1** has type index  $k$  where  $k = 0, \dots, \text{NElemsQuad1} - 1$ , then its net index will be

```
mesh->FirstNetIndex[E_ELEM_TYPE_QUAD1] + k - 1
```

where **E\_ELEM\_TYPE\_QUAD1** is a macro integer defined in the E-Lib header file `<e_lib.h>`. A similar macro integer is defined for each element type.

- The *global index* of an element is defined as its net index in the original mesh. These indices are stored in the vector **GEIs** using net indexing. The members **GEIsTriang1** etc. point to the first global element index of the corresponding element type.

**EXAMPLE** A simple code for the *type indexing*

```
for(i = 0; i < mesh->NElemsTriang1; i++)
    printf("\n%d. element %d %d %d",
           i+1,
           mesh->NodesTriang1[i][0],
           mesh->NodesTriang1[i][1],
           mesh->NodesTriang1[i][2]);
```

The following code has been cut from the **e-lib** source code to show the setup of **GEIs** structure and to show the use of the *net indexing*

```
for(i = 0; i < mesh->TotalNElems; i++)
    mesh->GEIs[i] = i+1;

mesh->GEIsLink1 =
    mesh->GEIs + mesh->FirstNetIndex[E_ELEM_TYPE_LINK1] - 1;
mesh->GEIsLink2 =
    mesh->GEIs + mesh->FirstNetIndex[E_ELEM_TYPE_LINK2] - 1;
[...]
```

**IMPORTANT** **TotalNElems** component of the **MESH** structure should always reflect the total number of elements in the mesh.



Some programmers like to control the time-stepping in their programs by dividing the time interval into a number of separate iterations. The variables **NInternalTimeSteps** and **NExternalTimeSteps** have been defined for this purpose, where **NExternalTimeSteps** sets of **NInternalTimeSteps** time-steps are to be performed. Each time-step is of length **TimeStep**. Alternatively, the total number of time-steps can be given by variable **NTimeSteps**.

The member **DampingFactor** is the viscous damping factor and **Beta** and **Gamma** the Newmark's  $\beta$  and  $\gamma$  integration constants. **NodalDOF** is a vector which stores the number of the degrees of freedom of each node and **TotalNodalDOF** is the total number of nodal degrees of freedom of the mesh.

**NOTE** If the **MESH** structure is created by the user the **NodalDOF** structure should be setup for compatibility. Currently no **e-lib** function relies on the existence of it.

The members **NStressPointsLink1**, etc. determine the number points where stresses should be stored. At each point six stress components can be stored. If this value is not set to a number greater than zero than the stress substructure is not accessible.

The indices of the nodes for which boundary conditions are imposed are stored in the vector **BCNodes** which is of length **NBCNodes**. For each degree of freedom,  $l$ , of the  $k$ -th boundary condition node, ( $k = 1, \dots, \text{NBCNodes}$ )

```
mesh->BCType[k - 1][1 - 1]
```

equals **E\_BC\_NUM\_FREE** if no conditions apply, **E\_BC\_NUM\_FIXED** if the node is to be initially displaced by an amount

```
mesh->BCDispls[k - 1][1 - 1]
```

to a new fixed position, and equals **E\_BC\_NUM\_SPRING** if elastic forces with the spring-constant

```
mesh->BCSpringConsts[k - 1][1 - 1]$
```

are to be modelled.

**EXAMPLE** A simple example for the handling of boundary conditions

```
Mass = e_alloc_i_vect(mesh->NNodes);

for (i = 0; i < mesh->NBCNodes; i++)
{
    n = mesh->BCNodes[i]-1;
    for (k = 0; k < E_NDIMS; k++)
    {
        if (mesh->BCTypes[i][k] == E_BC_NUM_FIXED)
            Mass[n][k] = BIG_MASS;
    }
}
```

In a similar manner to the boundary conditions,

```
mesh->Loads[k - 1][l - 1]
```

is the load applied to the  $k$ -th loaded node for the  $l$ -th degree of freedom,  $k = 1, \dots, \mathbf{NLoadedNodes}$  and the  $k$ -th loaded node has index  $\mathbf{LoadedNodes}[k - 1]$ .

The mesh definition file does not store the material properties – these are held in a separate materials file (**.mat**). Materials in the mesh definition file are referred to by name. The **MESH** member **NMdfMats** being the total number of distinct material names in the **.mdf** file and **NCompMatsType1** the total number of distinct composite materials of type 1. The member **NMats** is the total number of materials in the mesh including those contained in any composite materials. Within a program, the material properties and the properties of type 1 composite materials are stored in **MAT** and **COMPMAT1** structures respectively (see Sections 3.2 and 3.3). The member variable **Mats** has been defined to access these properties. **Mats** is a vector of void pointers where each pointer points to the appropriate **MAT** or **COMPMAT1** structure of a particular element using *net indexing*. The components of the pointer vectors **MatsTriang1**, **CompMatsTriang2**, etc. point to the material and composite material properties of each element using *element type indexing* (see the description of structure **GEIs**).

**EXAMPLE** To change the density of all **TRIANG1** element in the mesh the following code can be used

```
for(i = 0; i < mesh->NElemsTriang1; i++)
    mesh->MatsTriang1[i].Density = 100.0;
```

The **MESH** structure contains further structure types which store additional mesh data – namely decomposition data, element mesh parameters, stresses, finite element errors and geometric data. Each structure, or sub-structure as they will be called, has a particular data file associated with it and corresponding input-output functions in the **e-Lib** library. The **MESH** Boolean array **SubStructSet** keeps track of which sub-structures contain data. If, for example, stresses have been read from a **.ste** file, then

```
mesh->SubStructSet[E_SS_NUM_STRESS]
```

will be **TRUE**. The macro integer **E\_SS\_NUM\_STRESS** is defined in **<e\_lib.h>** along with similarly defined macros for each of the other sub-structures. Strictly speaking, the **SubStructSet** Booleans relate to data file types rather than to the **MESH** sub-structures. Hence

```
mesh->SubStructSet[E_SS_NUM_MATPROPS]
```

refers to whether material data and/or composite material data have been read from a materials file. These data are held in the same **.mat** file even though within a program they are stored separately in **MAT** and **COMPMAT1** structures.

As stated in Chapter 2, the mesh data files are based on a keyword format. The components of the character array **KeywordSet** act as Booleans for storing which keywords have been declared in the mesh definition file. For example,

```
mesh->KeywordSet[E_KW_NUM_NODES_TETRAH1]
```

is **TRUE** if the mesh contains elements of type **TETRAH1**. The macro integer **E\_KW\_NUM\_NODES\_TETRAH1** and macro Boolean values **TRUE** and **FALSE** are defined in `<e_lib.h>`. Similar macros are defined for all the other mesh definition file keywords.

**IMPORTANT** The **KeywordSet** Booleans are needed by several **E-Lib** functions, like the output function **PutMeshData()** which creates a mesh definition file and **CreateSubMesh()** which uses decomposition data to create a sub-mesh from an original, decomposed mesh (see Chapter 4).

Table 3.4 and 3.5 shows the dimension of vectors and matrices defined in the **MESH** structure where **E\_NELEM\_TYPES** is the number of element types and **E\_DOF\_MAX** is the maximum possible nodal degree of freedom. Table 3.6 lists whether a matrix is fragmented (`_frag_`) or contiguous (`_cont_`) (see the dynamic matrix allocation functions in Appendix 1). If a matrix is not listed in Table 3.6 it is always allocated as a continuous matrix.

The following sections describe the **MESH** sub-structures in more detail.

Member	Dimensions
FirstNetIndex	E_NELEM_TYPES
MeshPointCoords	NMeshPoints $\times$ 3
NodesLink1	NElemsLink1 $\times$ E_NNODES_LINK1
NodesLink2	NElemsLink2 $\times$ E_NNODES_LINK2
NodesLink3	NElemsLink3 $\times$ E_NNODES_LINK3
NodesLink4	NElemsLink4 $\times$ E_NNODES_LINK4
NodesLink5	NElemsLink5 $\times$ E_NNODES_LINK5
NodesTriang1	NElemsTriang1 $\times$ E_NNODES_TRIANG1
NodesTriang2	NElemsTriang2 $\times$ E_NNODES_TRIANG2
NodesTriang3	NElemsTriang3 $\times$ E_NNODES_TRIANG3
NodesTriang4	NElemsTriang4 $\times$ E_NNODES_TRIANG4
NodesTriang5	NElemsTriang5 $\times$ E_NNODES_TRIANG5
NodesQuad1	NElemsQuad1 $\times$ E_NNODES_QUAD1
NodesQuad3	NElemsQuad3 $\times$ E_NNODES_QUAD3
NodesTetra1	NElemsTetra1 $\times$ E_NNODES_TETRAH1
NodesBlock1	NElemsBlock1 $\times$ E_NNODES_BLOCK1
NodalDOF	NNodes
BCNodes	NBCNodes
BCTypes	NBCNodes $\times$ E_DOF_MAX
BCDispls	NBCNodes $\times$ E_DOF_MAX
BCSpringConsts	NBCNodes $\times$ E_DOF_MAX
LoadedNodes	NLoadedNodes
Loads	NLoadedNodes $\times$ E_DOF_MAX
Mats	TotalNElems
MatsLink1	NElemsLink1
MatsLink2	NElemsLink2
MatsLink3	NElemsLink3
MatsLink4	NElemsLink4
MatsLink5	NElemsLink5
MatsTriang1	NElemsTriang1
CompMatsTriang2	NElemsTriang2
MatsTriang3	NElemsTriang3
MatsTriang4	NElemsTriang4
MatsTriang5	NElemsTriang5
MatsQuad1	NElemsQuad1
CompMatsQuad3	NElemsQuad3
MatsTetra1	NElemsTetra1
MatsBlock1	NElemsBlock1

Table 3.4: (a) The **MESH** structure vector and matrix dimensions

Member	Dimensions
NodeTypes	NMeshPoints
NNurbsCurves	NMeshPoints
NurbsCurveParams	NMeshPoints $\times$ NNurbsCurves[k]
NurbsCurves	NMeshPoints $\times$ NNurbsCurves[k] $\times$ Geom $\rightarrow$ NNurbsCurves
GNI	NNodes
GEIs	TotalNElems
GEIsLink1	NElemsLink1
GEIsLink2	NElemsLink2
GEIsLink3	NElemsLink3
GEIsLink4	NElemsLink4
GEIsLink5	NElemsLink5
GEIsTriang1	NElemsTriang1
GEIsTriang2	NElemsTriang2
GEIsTriang3	NElemsTriang3
GEIsTriang4	NElemsTriang4
GEIsTriang5	NElemsTriang5
GEIsQuad1	NElemsQuad1
GEIsQuad3	NElemsQuad3
GEIsTetrah1	NElemsTetrah1
GEIsBlock1	NElemsBlock1
Props	NMats
CompPropsType1	NCompMatsType1
KeywordSet	E_NKEYWORDS
SubStructSet	E_NSUBSTRUCTS

Table 3.5: (b) The **MESH** structure vector and matrix dimensions

Member	Matrix Type
MeshPointCoords	.cont. (.frag. for <b>__E_MSWIN__</b> )
NodesLink1	.cont. (.frag. for <b>__E_MSWIN__</b> )
NodesLink2	.cont. (.frag. for <b>__E_MSWIN__</b> )
NodesLink3	.cont. (.frag. for <b>__E_MSWIN__</b> )
NodesLink4	.cont. (.frag. for <b>__E_MSWIN__</b> )
NodesLink5	.cont. (.frag. for <b>__E_MSWIN__</b> )
NodesTriang1	.cont. (.frag. for <b>__E_MSWIN__</b> )
NodesTriang2	.cont. (.frag. for <b>__E_MSWIN__</b> )
NodesTriang3	.cont. (.frag. for <b>__E_MSWIN__</b> )
NodesTriang4	.cont. (.frag. for <b>__E_MSWIN__</b> )
NodesTriang5	.cont. (.frag. for <b>__E_MSWIN__</b> )
NodesQuad1	.cont. (.frag. for <b>__E_MSWIN__</b> )
NodesQuad3	.cont. (.frag. for <b>__E_MSWIN__</b> )
NodesTetrah1	.cont. (.frag. for <b>__E_MSWIN__</b> )
NodesBlock1	.cont. (.frag. for <b>__E_MSWIN__</b> )
BCTypes	.frag.
BCDispls	.frag.
BCSpringConsts	.frag.
Loads	.frag.

Table 3.6: The **MESH** structure matrix types

## 3.2 The MAT Structure

The **MAT** structure stores the material property data (Figure 3.1). **Density** is the material density and **YMod** the isotropic Young's modulus. **YMod\_x**, **YMod\_y** and **YMod\_z** are the x-, y- and z-direction Young's moduli. Although not a material property, the planar element thickness is stored in the member **Thickness**.

```
typedef struct
{
    char *Name;                /* material name */

    int MatType;               /* index for a user defined material model */

    double Density;            /* density */
    double YMod;               /* isotropic Young's modulus */
    double YMod_x;             /* x-direction Young's modulus */
    double YMod_y;             /* y-direction Young's modulus */
    double YMod_z;             /* z-direction Young's modulus */
    double PoissonsRatio;      /* Poisson's ratio */
    double Thickness;          /* thickness */

    int iParam1;
        .
        .
        .
    int iParam6;

    double dParam1;
        .
        .
        .
    double dParam20;

    int KeywordSet[E_NMAT_PROP_KEYWORDS];
}
MAT;
```

Figure 3.1: The **MAT** structure

Users can define their own material properties using the integer variables **iParam1**, ..., **iParam6** and double precision variables **dParam1**, ..., **dParam20**. The definition of each variable should be clearly documented. The member **Mat-Type** holds an index of a user defined material model.

The members of the vector **KeywordSet** act as Booleans indicating which material properties have been set. For example, if **YMod** has been defined for the *i* material then

```
mesh->Props[i].KeywordSet[E_MKW_NUM_YMOD]
```

will be **TRUE**.

### 3.3 The COMPMAT1 Structure

This structure stores type 1 composite material properties (Figure 3.2). These composites are composed of **NLayers** layers of thicknesses **Thicknesses**[ $k - 1$ ],  $k = 1, \dots, \mathbf{NLayers}$ . **LayerMats** is a vector of **MAT** pointers which point to the **MAT** structure of each layer.

**EXAMPLE** For example, the  $k$ -th layer of the  $i$ -th **TRIANG2** element in the **MESH** structure **mesh** will have the density

```
mesh->MatsTriang2[i - 1]->LayerMats[k - 1].Density
```

```
typedef struct
{
    char *Name;                /* composite material name */
    int NLayers;               /* number of layers */
    MAT **LayerMats;           /* layer materials */
    double *Thicknesses;       /* layer thicknesses */
}
COMPMAT1;
```

Figure 3.2: The **COMPMAT1** structure

Table 3.7 shows the dimension of vectors and matrices defined in the **COMPMAT1** structure

Member	Dimensions
<b>LayerMats</b>	<b>NLayers</b>
<b>Thicknesses</b>	<b>NLayers</b>

Table 3.7: The **COMPMAT1** structure vector dimensions



### 3.4 The DECOMP Structure

The **NSubDoms** member of the **DECOMP** structure (Figure 3.3) stores the number of subdomains created during domain decomposition. The subdomain indices of the elements are stored in the vector **SubDoms** in the order of the net element indices. The pointers **SubDomsTriang1** etc. point to the subdomain index of the first element of each element type.

**EXAMPLE** To sum up the elements in subdomain 2 the following code can be used

```
counter = 0;

for(i = 0; i < mesh->TotalNElems; i++)
    if(mesh->Decomp.SubDoms[i] == 2)
        counter++;
```

or to count the **TRIANG1** elements in subdomain 2 the code will be

```
counter = 0;

for(i = 0; i < mesh->NElemsTriang1; i++)
    if(mesh->Decomp.SubDomsTriang1[i] == 2)
        counter++;
```

```
typedef struct
{
    int NSubDoms;                /* number of subdomains */
    int *SubDoms;                /* subdomain indices */
    int *SubDomsLink1;           /* LINK1 subdomain indices */
    int *SubDomsLink2;           /* LINK2 subdomain indices */
    int *SubDomsLink3;           /* LINK3 subdomain indices */
    int *SubDomsLink4;           /* LINK4 subdomain indices */
    int *SubDomsLink5;           /* LINK5 subdomain indices */
    int *SubDomsTriang1;         /* TRIANG1 subdomain indices */
    int *SubDomsTriang2;         /* TRIANG2 subdomain indices */
    int *SubDomsTriang3;         /* TRIANG3 subdomain indices */
    int *SubDomsTriang4;         /* TRIANG4 subdomain indices */
    int *SubDomsTriang5;         /* TRIANG5 subdomain indices */
    int *SubDomsQuad1;           /* QUAD1 subdomain indices */
    int *SubDomsQuad3;           /* QUAD3 subdomain indices */
    int *SubDomsTetrah1;         /* TETRAH1 subdomain indices */
    int *SubDomsBlock1;          /* BLOCK1 subdomain indices */
}
DECOMP;
```

Figure 3.3: The **DECOMP** structure

Member	Dimensions
SubDoms	TotalNElems
SubDomsLink1	NElemsLink1
SubDomsLink2	NElemsLink2
SubDomsLink3	NElemsLink3
SubDomsLink4	NElemsLink4
SubDomsLink5	NElemsLink5
SubDomsTriang1	NElemsTriang1
SubDomsTriang2	NElemsTriang2
SubDomsTriang3	NElemsTriang3
SubDomsTriang4	NElemsTriang4
SubDomsTriang5	NElemsTriang5
SubDomsQuad1	NElemsQuad1
SubDomsQuad3	NElemsQuad3
SubDomsTetrah1	NElemsTetrah1
SubDomsBlock1	NElemsBlock1

Table 3.8: The **DECOMP** structure vector dimensions

### 3.5 The MESHPARAM Structure

The structure **MESHPARAM** (Figure 3.4) stores the element mesh parameters in the vector **MeshParams** using net element indexing. **MeshParamsTriang1** etc. point to the element mesh parameter of the first element of each element type. It is possible to store the nodal mesh parameters in the vector **NodalMeshParams**. The variables **NodalMeshParamsSet**, **ElemMeshParamsSet** indicate whether either nodal or element mesh parameters or both are set. Table 3.9 shows the dimensions of the vectors.

For an example how to use the structure see Section 3.4.

```
typedef struct
{
    double *NodalMeshParams;          /* nodal mesh parameters */
    double *ElemMeshParams;           /* element mesh parameters */
    double *ElemMeshParamsLink1;      /* LINK1 element mesh parameters */
    double *ElemMeshParamsLink2;      /* LINK2 element mesh parameters */
    double *ElemMeshParamsLink3;      /* LINK3 element mesh parameters */
    double *ElemMeshParamsLink4;      /* LINK4 element mesh parameters */
    double *ElemMeshParamsLink5;      /* LINK5 element mesh parameters */
    double *ElemMeshParamsTriang1;     /* TRIANG1 element mesh parameters */
    double *ElemMeshParamsTriang2;     /* TRIANG2 element mesh parameters */
    double *ElemMeshParamsTriang3;     /* TRIANG3 element mesh parameters */
    double *ElemMeshParamsTriang4;     /* TRIANG4 element mesh parameters */
    double *ElemMeshParamsTriang5;     /* TRIANG5 element mesh parameters */
    double *ElemMeshParamsQuad1;       /* QUAD1 element mesh parameters */
    double *ElemMeshParamsQuad3;       /* QUAD3 element mesh parameters */
    double *ElemMeshParamsTetrah1;     /* TETRAH1 element mesh parameters */
    double *ElemMeshParamsBlock1;      /* BLOCK1 element mesh parameters */

    int NodalMeshParamsSet;            /* nodal mesh parameters set? */
    int ElemMeshParamsSet;             /* element mesh parameters set? */
}
MESHPARAM;
```

Figure 3.4: The **MESHPARAM** structure

Member	Dimensions
NodalMeshParams	NMeshPoints
ElemMeshParams	TotalNElems
ElemMeshParamsLink1	NElemsLink1
ElemMeshParamsLink2	NElemsLink2
ElemMeshParamsLink3	NElemsLink3
ElemMeshParamsLink4	NElemsLink4
ElemMeshParamsLink5	NElemsLink5
ElemMeshParamsTriang1	NElemsTriang1
ElemMeshParamsTriang2	NElemsTriang2
ElemMeshParamsTriang3	NElemsTriang3
ElemMeshParamsTriang4	NElemsTriang4
ElemMeshParamsTriang5	NElemsTriang5
ElemMeshParamsQuad1	NElemsQuad1
ElemMeshParamsQuad3	NElemsQuad3
ElemMeshParamsTetrah1	NElemsTetrah1
ElemMeshParamsBlock1	NElemsBlock1

Table 3.9: The **MESHPARAM** structure vector dimensions

## 3.6 The FEERROR Structure

The structure **FEERROR** (Figure 3.5) stores the finite element errors in the vector **FEErrors** using net element indexing. **FEErrorsTriang1** etc. point to the finite element errors of the first element of each element type. Table 3.10 shows the dimensions of the vectors.

For an example how to use the structure see Section 3.4.

```
typedef struct
{
    double *FEErrors;                /* finite element errors */
    double *FEErrorsLink1;           /* LINK1 finite element errors */
    double *FEErrorsLink2;           /* LINK2 finite element errors */
    double *FEErrorsLink3;           /* LINK3 finite element errors */
    double *FEErrorsLink4;           /* LINK4 finite element errors */
    double *FEErrorsLink5;           /* LINK5 finite element errors */
    double *FEErrorsTriang1;          /* TRIANG1 finite element errors */
    double *FEErrorsTriang2;          /* TRIANG2 finite element errors */
    double *FEErrorsTriang3;          /* TRIANG3 finite element errors */
    double *FEErrorsTriang4;          /* TRIANG4 finite element errors */
    double *FEErrorsTriang5;          /* TRIANG5 finite element errors */
    double *FEErrorsQuad1;           /* QUAD1 finite element errors */
    double *FEErrorsQuad3;           /* QUAD3 finite element errors */
    double *FEErrorsTetrah1;          /* TETRAH1 finite element errors */
    double *FEErrorsBlock1;          /* BLOCK1 finite element errors */
}
FEERROR;
```

Figure 3.5: The **FEERROR** structure

Member	Dimensions
<b>FErrors</b>	<b>TotalNElems</b>
<b>FErrorsLink1</b>	<b>NElemsLink1</b>
<b>FErrorsLink2</b>	<b>NElemsLink2</b>
<b>FErrorsLink3</b>	<b>NElemsLink3</b>
<b>FErrorsLink4</b>	<b>NElemsLink4</b>
<b>FErrorsLink5</b>	<b>NElemsLink5</b>
<b>FErrorsTriang1</b>	<b>NElemsTriang1</b>
<b>FErrorsTriang2</b>	<b>NElemsTriang2</b>
<b>FErrorsTriang3</b>	<b>NElemsTriang3</b>
<b>FErrorsTriang4</b>	<b>NElemsTriang4</b>
<b>FErrorsTriang5</b>	<b>NElemsTriang5</b>
<b>FErrorsQuad1</b>	<b>NElemsQuad1</b>
<b>FErrorsQuad3</b>	<b>NElemsQuad3</b>
<b>FErrorsTetrah1</b>	<b>NElemsTetrah1</b>
<b>FErrorsBlock1</b>	<b>NElemsBlock1</b>

Table 3.10: The **FEERROR** structure vector dimensions

### 3.7 The STRESSES Structure

**IMPORTANT** Stress data can only exist if the variable **NStressPointsLink1**, etc. in the **MESH** structure has been defined. In the case of **TRIANG2** and **QUAD3** elements material information should be available as well in the **COMPAT1** structure. Obviously, the appropriate keywords should be set, as well.

The **StressesLink1**, etc. members store the stress components for the elements using element type indexing. The size of the vectors is shown in Tabel 3.11. **NStressPointsLink1**, etc. define the number of stress points for the elements, while **E\_NUM\_STRESS\_COMPONENTS** macro defines that there can be max. 6 stress components at each point. Any of the stress components can be ignored by the user. By default, there are six integer macros to access the stress components,

- **E\_SIGMA\_X** for  $\sigma_x$  stress,
- **E\_SIGMA\_Y** for  $\sigma_y$  stress,
- **E\_SIGMA\_Z** for  $\sigma_z$  stress,
- **E\_TAU\_XY** for  $\tau_{xy}$  stress,
- **E\_TAU\_YZ** for  $\tau_{yz}$  stress,
- **E\_TAU\_ZX** for  $\tau_{zx}$  stress,

but any other interpretation of the components can be used. For example the first three components represent the principal stress while the next three components provides the angles of the principal directions.

**EXAMPLE** To zero all stress components for **TRIANG1** element the following code can be used

```
for(i = 0; i < mesh->NElemsTriang1; i++)
  for(j = 0; j < mesh->NStressPointsTriang1; j++)
    for(k = 0; k < E_NUM_STRESS_COMPONENTS; k++)
      mesh->Stresses.StressesTriang1[i][j][k] = 0.00;
```

```

typedef struct
{
    double ***StressesLink1;          /* LINK1 finite element stresses */
    double ***StressesLink2;          /* LINK2 finite element stresses */
    double ***StressesLink3;          /* LINK3 finite element stresses */
    double ***StressesLink4;          /* LINK4 finite element stresses */
    double ***StressesLink5;          /* LINK5 finite element stresses */

    double ***StressesTriang1;         /* TRIANG1 finite element stresses */
    double ***StressesTriang2;         /* TRIANG2 finite element stresses */
    double ***StressesTriang3;         /* TRIANG3 finite element stresses */
    double ***StressesTriang4;         /* TRIANG4 finite element stresses */
    double ***StressesTriang5;         /* TRIANG5 finite element stresses */

    double ***StressesQuad1;           /* QUAD1 finite element stresses */
    double ***StressesQuad3;           /* QUAD3 finite element stresses */
    double ***StressesTetrah1;         /* TETRAH1 finite element stresses */
    double ***StressesBlock1;          /* BLOCK1 finite element stresses */
}
STRESSES;

```

Figure 3.6: The **STRESSES** structure

Member	Dimensions
StressesLink1	$\text{NElemsLink1} \times \text{NStressPointsLink1} \times \text{E\_NUM\_STRESS\_COMPONENTS}$
StressesLink2	$\text{NElemsLink2} \times \text{NStressPointsLink2} \times \text{E\_NUM\_STRESS\_COMPONENTS}$
StressesLink3	$\text{NElemsLink3} \times \text{NStressPointsLink3} \times \text{E\_NUM\_STRESS\_COMPONENTS}$
StressesLink4	$\text{NElemsLink4} \times \text{NStressPointsLink4} \times \text{E\_NUM\_STRESS\_COMPONENTS}$
StressesLink5	$\text{NElemsLink5} \times \text{NStressPointsLink5} \times \text{E\_NUM\_STRESS\_COMPONENTS}$
StressesTriang1	$\text{NElemsTriang1} \times \text{NStressPointsTriang1} \times \text{E\_NUM\_STRESS\_COMPONENTS}$
StressesTriang2	$\text{NElemsTriang2} \times \text{NStressPointsTriang2} \times$ $\text{CompMatsTriang2}[i] \rightarrow \text{NLayers} \times \text{E\_NUM\_STRESS\_COMPONENTS}$
StressesTriang3	$\text{NElemsTriang3} \times \text{NStressPointsTriang3} \times \text{E\_NUM\_STRESS\_COMPONENTS}$
StressesTriang4	$\text{NElemsTriang4} \times \text{NStressPointsTriang4} \times \text{E\_NUM\_STRESS\_COMPONENTS}$
StressesTriang5	$\text{NElemsTriang5} \times \text{NStressPointsTriang5} \times \text{E\_NUM\_STRESS\_COMPONENTS}$
StressesQuad1	$\text{NElemsQuad1} \times \text{NStressPointsQuad1} \times \text{E\_NUM\_STRESS\_COMPONENTS}$
StressesQuad3	$\text{NElemsQuad3} \times \text{NStressPointsQuad3} \times$ $\text{CompMatsQuad3}[i] \rightarrow \text{NLayers} \times \text{E\_NUM\_STRESS\_COMPONENTS}$
StressesTetrah1	$\text{NElemsTetrah1} \times \text{NStressPointsTetrah1} \times \text{E\_NUM\_STRESS\_COMPONENTS}$
StressesBlock1	$\text{NElemsBlock1} \times \text{NStressPointsBlock1} \times \text{E\_NUM\_STRESS\_COMPONENTS}$

Table 3.11: The **STRESSES** structure vector dimensions



## **3.8 The GEOM Structure**

For further information see “**MESHGEN USER MANUAL**”.

# Chapter 4

## The E-Lib Library

The E-Lib library comprises two main types of function. The first deals with data stored in a **MESH** structure, the second is more general and can be used in any type of program.

A number of different versions of the library are needed to allow for the various platforms on which a program may run. The locations of the library files can be found in Table 4.1 which also contains the inclusion directory of the header file `<e_lib.h>`. This file must be included in all source code modules that use the E-Lib library and must be preceded by a definition statement for one of the five platform macros `__E_CLANSIC__`, `__E_MSWIN__`, `__E_PVM__`, `__E_T800RTL__` and `__E_T800SAL__`. These macros denote command line ANSI C, Windows, PVM, transputer full run-time library and transputer stand-alone library respectively.

**IMPORTANT** One and only one of these macros must be defined.

From a program user's point of view, the only effect on the E-Lib library on changing the platform is the way the functions handle error messages and program termination. In the `__E_CLANSIC__` and `__E_T800RTL__` modes, error messages from the E-Lib error message functions are sent to the standard error stream and the program is immediately terminated by a call of the E-Lib function `e_exit(1)`. The procedure is the same for `__E_PVM__` except that termination is preceded by a call of the PVM function `pvm_exit()`. This removes the calling process from the virtual machine. E-Lib functions compiled for the `__E_MSWIN__` platform do not terminate in the event of failure. Instead, a message box is displayed and the function returns a value to indicate an error has occurred. Errors in a program built for the `__E_T800SAL__` platform cause the E-Lib function to enter an infinite loop without outputting an error message, thereby preventing any further execution of the calling routine.

Unless otherwise stated, all E-Lib functions display an error message in the event of failure and for Windows programs return an error value.

**NOTE** The `e_PlatformMessage` function (which is called by the error message functions) and the memory allocation and deallocation functions are all semaphore protected for the `__E_T800RTL__` and `__E_T800SAL__` platforms. None of the remaining E-Lib functions are protected in this way and must therefore be used with care in tasks containing more than one thread.

Platform	OP	E-Lib Library
..E-CLANSIC..	UNIX	<i>E_LIB</i> /lib/e_lib_sparc.a <i>E_LIB</i> /lib/e_lib_sgi.a <i>E_LIB</i> /lib/e_lib_linux.a
..E-CLANSIC..	DOS	<i>E_LIB</i> \lib\e_lib_do.lib
..E-MSWIN..	Windows	<i>E_LIB</i> \lib\e_lib_w.lib
..E-PVM..	UNIX	<i>E_LIB</i> /lib/e_lib_pvm3-sparc.a
..E-T800RTL..	DOS	<i>E_LIB</i> \lib\e_lib_m.t8
..E-T800SAL..	DOS	<i>E_LIB</i> \lib\e_lib_w.t8
		<b>&lt;e-lib.h&gt; Include Path</b>
		UNIX <i>E_LIB</i> /include/unix
		DOS <i>E_LIB</i> \include\dos
<i>E_LIB</i> = /opt/share/e-lib		

Table 4.1: The E-Lib library network pathnames and include paths

# Appendix 1 – The E-Lib Library

This appendix contains declarations and descriptions of all the functions in the E-Lib library. The term ‘error message’ is used in a general sense to refer to the action taken if a function error occurs. The particular action will depend upon the platform on which the program is running (see Chapter 4). Unless otherwise stated, the E-Lib functions always ‘display an error message’ in the event of failure.

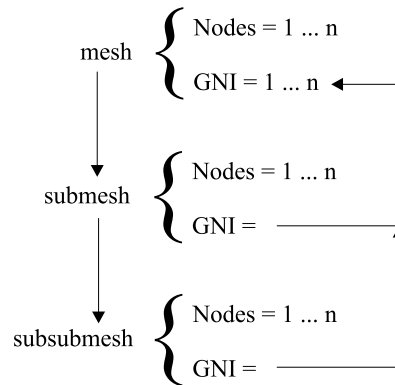
Reference should be made to Table 4.1 for the network location of the relevant E-Lib library file and the E-Lib include path.

FUNCTION	Mesh input-output functions
PLATFORM	Not <code>__E_T800SAL__</code>
DECLARATION	<pre> #include &lt;e_lib.h&gt;  int  e_GetMeshData(MESH**mesh, char *InputFile); int  e_PutMeshData(MESH *mesh, char *OutputFile); int  e_GetMaterials(MESH *mesh, char *InputFile); int  e_PutMaterials(MESH *mesh, char *OutputFile); int  e_GetDecompData(MESH *mesh, char *InputFile); int  e_PutDecompData(MESH *mesh, char *OutputFile); int  e_GetMeshParams(MESH *mesh, char *InputFile); int  e_PutMeshParams(MESH *mesh, char *OutputFile); int  e_GetStresses(MESH *mesh, char *InputFile); int  e_PutStresses(MESH *mesh, char *OutputFile); int  e_GetFEErrors(MESH *mesh, char *InputFile); int  e_PutFEErrors(MESH *mesh, char *OutputFile); </pre>
DESCRIPTION	<p>The <b>e_Get*()</b> and <b>e_Put*()</b> functions read and write mesh data to and from data files. The character pointers point to the input and output filenames which must not include the file extension. The appropriate file extension is added by the E-Lib function. The return value is 1 if the operation is successful, otherwise it is 0.</p> <p><b>e_GetMeshData()</b> creates a new <b>MESH</b> structure and reads in the main description of the mesh from a <b>.mdf</b> file. The memory for the <b>MESH</b> structure is allocated by the function. <b>e_PutMeshData()</b> writes the mesh data to a <b>.mdf</b> file.</p> <p><b>e_GetMaterials()</b> reads material and composite material properties from a <b>.mat</b> file. <b>e_PutMaterials()</b> writes the properties of to a <b>.mat</b> file.</p> <p><b>e_GetDecompData()</b> creates a new <b>DECOMP</b> structure and reads in domain decomposition data from a <b>.dom</b> file. <b>e_PutDecompData()</b> writes the domain decomposition data to a <b>.dom</b> file.</p> <p><b>e_GetMeshParams()</b> creates a new <b>MESHPARAM</b> structure and reads in the element mesh parameters from a <b>.emp</b> file. <b>e_PutMeshParams()</b> writes the element mesh parameters to a <b>.emp</b> file.</p> <p><b>e_GetStresses()</b> creates a new <b>STRESS</b> structure and reads in the stresses of each <b>TRIANG1</b> element from a <b>.ste</b> file. <b>e_PutStresses()</b> writes the stresses of the <b>TRIANG1</b> elements to a <b>.ste</b> file.</p> <p><b>e_GetFEErrors()</b> creates a new <b>FEERROR</b> structure and reads in the error value of each <b>TRIANG1</b> element from a <b>.fee</b> file. <b>e_PutFEErrors()</b> writes the errors of the <b>TRIANG1</b> elements to a <b>.fee</b> file.</p>
NOTE	<p>If a discrepancy exists between the value of the <b>MESH</b> member <b>NBCNodes</b> obtained from the mesh definition file and the actual number of boundary conditions defined in the file or between the <b>MESH</b> member <b>NLoadedNodes</b> and the actual number of loads, then <b>e_GetMeshData()</b> outputs a warning rather than an error message. In either case, the mesh member is set to the correct value.</p>
SEE ALSO	Mesh memory deallocations functions

<b>FUNCTION</b>	Mesh memory deallocation functions
<b>PLATFORM</b>	All
<b>DECLARATION</b>	<pre>#include &lt;e_lib.h&gt;  void e_FreeDecompData (MESH *mesh); void e_FreeFEErrors (MESH *mesh); void e_FreeMatProps (MESH *mesh); void e_FreeMeshData (MESH *mesh); void e_FreeMeshParams (MESH *mesh); void e_FreeStresses (MESH *mesh);</pre>
<b>DESCRIPTION</b>	<p><b>e_FreeMeshData ()</b> deallocates all memory associated with a <b>MESH</b> structure. It is safe to free a <b>MESH</b> structure only if it has first been properly initialised. A <b>MESH</b> structure is guaranteed to be properly initialised if it is created by either the <b>e_GetMeshData ()</b> or <b>e_CreateSubMesh ()</b> function.</p> <p>The remaining <b>e_Free* ()</b> functions free the memory of the corresponding sub-structure. It is safe to call these functions provided either the <b>MESH</b> structure has been properly initialised or the sub-structure was created by a <b>e_Get* ()</b> function.</p> <p>All of the above functions have no effect on a null <b>MESH</b> pointer.</p>
<b>SEE ALSO</b>	Mesh input functions

FUNCTION	Transputer send and receive functions
PLATFORM	<code>__E_T800RTL__</code> , <code>__E_T800SAL__</code>
DECLARATION	<pre> #include &lt;e_lib.h&gt;  void    e_chan_out_Buffer(int msglength,                           void *buffer, CHAN *chan); int      e_chan_in_Buffer(void *buffer, CHAN *chan);  void    e_chan_out_MeshData(MESH *mesh, CHAN *chan); void    e_chan_in_MeshData(MESH**mesh, CHAN *chan); void    e_chan_out_DecompData(MESH *mesh, CHAN *chan); void    e_chan_in_DecompData(MESH *mesh, CHAN *chan); void    e_chan_out_MeshParams(MESH *mesh, CHAN *chan); void    e_chan_in_MeshParams(MESH *mesh, CHAN *chan); void    e_chan_out_Mats(MESH *mesh, CHAN *chan); void    e_chan_in_Mats(MESH *mesh, CHAN *chan); </pre>
DESCRIPTION	<p><b>e_chan_out_Buffer()</b> sends a message of <b>msglength</b> bytes via the transputer channel pointed to by <b>chan</b>. <b>buffer</b> points to the message.</p> <p><b>e_chan_in_Buffer()</b> receives a buffered message on the channel pointed to by <b>chan</b>.</p> <p>Similarly, <b>e_chan_out_MeshData()</b> and <b>e_chan_in_MeshData()</b> send and receive <b>MESH</b> structures. <b>e_chan_in_MeshData()</b> allocates the required memory. The remaining functions respectively send and receive domain decomposition data, element mesh parameters and material properties. The sending functions extract the relevant information from the <b>MESH</b> structure pointed to by <b>mesh</b> and the receiving functions allocate memory as necessary and then write data into the appropriate mesh sub-structure.</p>

<b>FUNCTION</b>	Create a sub-mesh <b>MESH</b> structure
<b>PLATFORM</b>	All
<b>DECLARATION</b>	<pre>#include &lt;e_lib.h&gt;  int e_CreateSubMesh(MESH *mesh, MESH **SubMesh,                     char *SubMeshTitle, int SubdomainIndex);</pre>
<b>DESCRIPTION</b>	<p><b>e_CreateSubMesh()</b> uses domain decomposition data to create the <b>MESH</b> structure of a sub-mesh. The function argument <b>mesh</b> points to the original mesh and <b>*SubMesh</b> points to the <b>MESH</b> structure to be created. All relevant attributes are extracted from the original mesh to form the new structure, the memory for which is allocated by the function. The partitioning data must be contained in the <b>DECOMP</b> structure of the original mesh. <b>SubdomainIndex</b> is the subdomain index of the sub-mesh to be created and <b>SubMeshTitle</b> points to the title to be assigned to the sub-mesh.</p> <p>If a loaded node in the original mesh is shared by two or more subdomains, then each of the corresponding nodes in the created sub-meshes will experience the same full load.</p> <p><b>GNI</b> and <b>GEI</b> components of the <b>MESH</b> mesh structure are not modified during the generation of a submesh.</p>





<b>FUNCTION</b>	Mesh type functions
<b>PLATFORM</b>	All
<b>DECLARATION</b>	<pre>#include &lt;e_lib.h&gt;  int e_IsOfOneElementType(MESH *mesh); int e_PlanarElementsOnly(MESH *mesh);</pre>
<b>DESCRIPTION</b>	<p><b>e_IsOfOneElementType()</b> returns 1 if the mesh pointed to by <b>mesh</b> is composed of only one element type, otherwise it returns 0.</p> <p><b>e_PlanarElementsOnly()</b> returns 1 if the mesh pointed to by <b>mesh</b> is composed of only planar elements, otherwise it returns 0.</p>

FUNCTION	Input-output functions
PLATFORM	Not <code>__E_T800SAL__</code> except <code>e_PlatformMessage()</code>
DECLARATION	<pre>#include &lt;e_lib.h&gt;  int e_GetLine(char *line, int MaxLineLength,               int *LineNumber, FILE *fp); int e_GetNextLine(char *line, int *LineNumber,                   char *FullPathName, FILE *fp);  void e_beep(void); void e_PlatformMessage(char *message);</pre>
DESCRIPTION	<p><code>e_GetLine()</code> and <code>e_GetNextLine()</code> both read a line of text from a file into the character array <code>line</code>, where <code>fp</code> points to the input file. The functions skip over blank and comment lines. <code>LineNumber</code> points to the current line number, the value of which should be set to 0 before the initial call of either of the two functions. The line number is incremented accordingly thereafter. Each call of <code>e_GetLine()</code> and <code>e_GetNextLine()</code> reads from the start of a new line.</p> <p><code>e_GetLine()</code> reads in up to <code>MaxLineLength</code> characters into the array <code>line</code>. This does not include the terminating null. <code>line</code> is assumed to be of sufficient length to hold the string and in the event of an error, no error messages are generated. The return value is 1 if the read is successful, EOF if the end of file is encountered and 0 if an error occurs.</p> <p><code>e_GetNextLine()</code> reads up to <code>E_LINE_MAX</code> characters into the character array <code>line</code> not including the terminating null. The macro <code>E_LINE_MAX</code> is defined in the header file <code>&lt;e_lib.h&gt;</code> and <code>line</code> is assumed to be of at least <code>E_LINE_MAX + 1</code> in length. In the event of an error or the end of the file, the line number and the file pathname <code>FullPathName</code> are passed to the E-Lib error message function <code>e_FileLineError()</code> and <code>e_GetNextLine()</code> returns 0. If the read is successful, the return value is 1.</p> <p><code>e_beep()</code> outputs a beep under DOS and Windows. It should not be used in UNIX applications.</p> <p><code>e_PlatformMessage()</code> outputs a message appropriate for the platform on which the program is built. On the <code>__E_CLANSIC__</code> and <code>__E_PVM__</code> platforms, the message is sent to the standard error stream. On the <code>__E_T800SAL__</code> platform the message is also sent to the standard error stream but the printing function is semaphore protected and the output function is flushed after use (this is also semaphore protected). The <code>__E_MSWIN__</code> platform causes a message box to be displayed. <code>e_PlatformMessage()</code> for <code>__E_T800SAL__</code> simply performs a return.</p>

FUNCTION	Warning message functions
PLATFORM	All
DECLARATION	<pre>#include &lt;e_lib.h&gt;  void e_GeneralWarning(char *Warning, ...); void e_FileWarning(char *FullPathName, char *Warning); void e_LineWarning(int LineNumber, char *Warning); void e_FileLineWarning(int LineNumber, char *FullPathName,                       char *Warning);</pre> <p>Each of these functions uses the function <b>e_PlatformMessage()</b> to output a warning message.</p> <p>The output of <b>e_GeneralWarning()</b> is of the form "WARNING: <i>Warning</i>\n", where <i>Warning</i> is the string pointed to by <b>Warning</b>.</p> <p><b>e_FileWarning()</b> outputs a message of the form "WARNING: <i>Warning</i>: '<i>FullPathName</i>'\n", where <i>FullPathName</i> is the file pathname pointed to by <b>FullPathName</b>.</p> <p><b>e_LineWarning()</b> outputs a message of the form "WARNING: line <i>LineNumber</i>: <i>Warning</i>\n", where <i>LineNumber</i> is the value of the current line number, <b>LineNumber</b>.</p> <p>The form of the output of <b>e_FileLineWarning()</b> is "WARNING: line <i>LineNumber</i>: <i>Warning</i>: '<i>FullPathName</i>'\n".</p> <p>In all cases, the entire message to be displayed is assumed to be no greater than <b>E_LINE_MAX</b> characters in length, not including the terminating null. <b>E_LINE_MAX</b> is defined in the header file <b>&lt;e_lib.h&gt;</b>.</p>
SEE ALSO	<b>e_PlatformMessage()</b>

FUNCTION	Error message functions
PLATFORM	All
DECLARATION	<pre>#include &lt;e_lib.h&gt;  void e_GeneralError(char *ErrorMessage, ...); void e_FileError(char *FullPathName, char *ErrorMessage); void e_LineError(int LineNumber, char *ErrorMessage); void e_FileLineError(int LineNumber, char *FullPathName,                     char *ErrorMessage ); void e_SystemError(void);</pre> <p>Each of these functions uses the function <b>e_PlatformMessage()</b> to output an error message and, with the exception of <b>__E_MSWIN__</b> programs, then calls <b>e_exit(1)</b> to terminate the program.</p> <p>The output of <b>e_GeneralError()</b> is of the form "ERROR: <i>ErrorMessage</i>\n", where <i>ErrorMessage</i> is the string pointed to by <b>ErrorMessage</b>.</p> <p><b>e_FileError()</b> outputs a message of the form "ERROR: <i>ErrorMessage</i>: '<i>FullPathName</i>'\n", where <i>FullPathName</i> is the file pathname pointed to by <b>FullPathName</b>.</p> <p><b>e_LineError()</b> outputs a message of the form "ERROR: line <i>LineNumber</i>: <i>ErrorMessage</i>\n", where <i>LineNumber</i> is the value of the current line number <b>LineNumber</b>.</p> <p>The form of the output of <b>e_FileLineError()</b> is "ERROR: line <i>LineNumber</i>: <i>ErrorMessage</i>: '<i>FullPathName</i>'\n".</p> <p><b>e_SystemError()</b> outputs the message "System error\n".</p> <p>In all cases, the entire message to be displayed is assumed to be no greater than <b>E_LINE_MAX</b> characters in length, not including the terminating null. <b>E_LINE_MAX</b> is defined in the header file <b>&lt;e_lib.h&gt;</b>.</p>
SEE ALSO	<b>e_exit()</b> , <b>e_PlatformMessage()</b>

<b>FUNCTION</b>	Program termination
<b>PLATFORM</b>	All
<b>DECLARATION</b>	<pre>#include &lt;e_lib.h&gt;  void e_exit(int ExitStatus);</pre>
<b>DESCRIPTION</b>	<p><b>e_exit()</b> terminates <b>__E_CLANSIC__</b>, <b>__E_PVM__</b> and <b>__E_T800RTL__</b> platform programs with the exit status <b>ExitStatus</b>. For <b>__E_PVM__</b>, the PVM function <b>pvm_exit()</b> is called before termination. <b>__E_MSWIN__</b> programs are not terminated automatically. Instead a message box is displayed informing the user that the <b>exit()</b> function has been called and that the program should be terminated. <b>__E_T800SAL__</b> programs enter an infinite loop.</p>

<b>FUNCTION</b>	Vector dynamic memory allocation functions
<b>PLATFORM</b>	All
<b>DECLARATION</b>	<pre>#include &lt;e_lib.h&gt;  void    *e_calloc(unsigned int length, unsigned int NBytes);  int      *e_alloc_i_vect(unsigned int length); float    *e_alloc_f_vect(unsigned int length); double   *e_alloc_d_vect(unsigned int length); char     *e_alloc_c_vect(unsigned int length); void **e_alloc_vp_vect(unsigned int length);</pre>
<b>DESCRIPTION</b>	<p><b>e_calloc()</b> is a platform independent version of the standard dynamic memory allocation function <b>calloc()</b>. If the allocation is successful, the function returns a void pointer to a vector of length <b>length</b>, where each element is of size <b>NBytes</b>. Each component is initialised to zero. If the memory cannot be allocated, <b>e_calloc()</b> returns NULL.</p> <p><b>e_alloc_i_vect()</b>, <b>e_alloc_f_vect()</b>, <b>e_alloc_d_vect()</b> and <b>e_alloc_c_vect()</b> return pointers to integer, float, double precision and character arrays of length <b>length</b> respectively and initialises them to zero. If the allocation is not successful, the functions return NULL. <b>e_alloc_vp_vect()</b> returns a pointer to a vector of null void pointers or NULL if the allocation fails.</p>
<b>NOTE</b>	Vectors of zero length can be allocated
<b>SEE ALSO</b>	Vector deallocation function

FUNCTION	Matrix dynamic memory allocation functions
PLATFORM	All
DECLARATION	<pre> #include &lt;e_lib.h&gt;  int    **e_alloc_frag_i_matr(unsigned int NRows,                              unsigned int NCols ); float  **e_alloc_frag_f_matr(unsigned int NRows,                              unsigned int NCols ); double **e_alloc_frag_d_matr(unsigned int NRows,                              unsigned int NCols ); char   **e_alloc_frag_c_matr(unsigned int NRows,                              unsigned int NCols ); void   **e_alloc_frag_vp_matr(unsigned int NRows,                              unsigned int NCols );  int    **e_alloc_cont_i_matr(unsigned int NRows,                              unsigned int NCols ); float  **e_alloc_cont_f_matr(unsigned int NRows,                              unsigned int NCols ); double **e_alloc_cont_d_matr(unsigned int NRows,                              unsigned int NCols ); char   **e_alloc_cont_c_matr(unsigned int NRows,                              unsigned int NCols ); void   **e_alloc_cont_vp_matr(unsigned int NRows,                              unsigned int NCols );  int    **e_init_frag_i_matr(unsigned int NRows); float  **e_init_frag_f_matr(unsigned int NRows); double **e_init_frag_d_matr(unsigned int NRows); char   **e_init_frag_c_matr(unsigned int NRows); void   **e_init_frag_vp_matr(unsigned int NRows); </pre>
DESCRIPTION	<p>The functions <code>e_alloc_frag_i_matr()</code>, <code>e_alloc_frag_f_matr()</code>, <code>e_alloc_frag_d_matr()</code> and <code>e_alloc_frag_c_matr()</code> respectively return pointers to integer, float, double precision and character matrices composed of <b>NRows</b> rows and <b>NCols</b> columns. Each row is allocated separately (i.e. the matrices are <i>fragmented</i>) and all elements are initialised to zero. <code>e_alloc_frag_vp_matr()</code> returns a pointer to a fragmented matrix of null void pointers.</p> <p><code>e_alloc_cont_i_matr()</code>, <code>e_alloc_cont_f_matr()</code>, <code>e_alloc_cont_d_matr()</code> and <code>e_alloc_cont_c_matr()</code> are the same as their fragmented counterparts except that the entire matrix lies in <i>contiguous</i> memory. Similarly, <code>e_alloc_cont_vp_matr()</code> returns a pointer to a contiguous matrix of null void pointers.</p> <p><code>e_init_frag_i_matr()</code>, <code>e_init_frag_f_matr()</code>, <code>e_init_frag_d_matr()</code>, <code>e_init_frag_c_matr()</code> and <code>e_init_frag_vp_matr()</code> initialise a fragmented matrix by returning a pointer of the appropriate type to a vector of <b>NRows</b> null pointers.</p> <p>All of the above functions return NULL if the memory cannot be allocated.</p>
NOTE	Matrices can be allocated with one or more zero dimensions provided the standard function <code>free()</code> is able to safely free a vector allocated with zero length

**SEE ALSO** | Matrix deallocation functions



FUNCTION	3-D matrix dynamic memory allocation functions
PLATFORM	All
DECLARATION	<pre> #include &lt;e_lib.h&gt;  int     ***e_alloc_frag_i_3D_matr(unsigned int NDepth,                                    unsigned int NRows,                                    unsigned int NCols ); float   ***e_alloc_frag_f_3D_matr(unsigned int NDepth,                                    unsigned int NRows,                                    unsigned int NCols ); double  ***e_alloc_frag_d_3D_matr(unsigned int NDepth,                                    unsigned int NRows,                                    unsigned int NCols ); char    ***e_alloc_frag_c_3D_matr(unsigned int NDepth,                                    unsigned int NRows,                                    unsigned int NCols ); void    ****e_alloc_frag_vp_3D_matr(unsigned int NDepth,                                      unsigned int NRows,                                      unsigned int NCols );  int     ***e_alloc_cont_i_3D_matr(unsigned int NDepth,                                    unsigned int NRows,                                    unsigned int NCols ); float   ***e_alloc_cont_f_3D_matr(unsigned int NDepth,                                    unsigned int NRows,                                    unsigned int NCols ); double  ***e_alloc_cont_d_3D_matr(unsigned int NDepth,                                    unsigned int NRows,                                    unsigned int NCols ); char    ***e_alloc_cont_c_3D_matr(unsigned int NDepth,                                    unsigned int NRows,                                    unsigned int NCols ); void    ****e_alloc_cont_vp_3D_matr(unsigned int NDepth,                                      unsigned int NRows,                                      unsigned int NCols );  int     ***e_init_frag_i_3D_matr(unsigned int NDepth); float   ***e_init_frag_f_3D_matr(unsigned int NDepth); double  ***e_init_frag_d_3D_matr(unsigned int NDepth); char    ***e_init_frag_c_3D_matr(unsigned int NDepth); void    ****e_init_frag_vp_3D_matr(unsigned int NDepth); </pre>
DESCRIPTION	<p>The functions <code>e_alloc_frag_i_3D_matr()</code>, <code>e_alloc_frag_f_3D_matr()</code>, <code>e_alloc_frag_d_3D_matr()</code>, <code>e_alloc_frag_c_3D_matr()</code> and <code>e_alloc_frag_vp_3D_matr()</code> return pointers to integer, float, double precision, character and void pointer three dimensional matrices respectively. Each matrix consists of a vector of <b>NDepth</b> pointers which point to <i>fragmented</i> matrices with <b>NRows</b> rows and <b>NCols</b> columns. All elements are initialised to zero, or NULL in the case of <code>e_alloc_frag_vp_3D_matr()</code>.</p> <p><code>e_alloc_cont_i_3D_matr()</code>, <code>e_alloc_cont_f_3D_matr()</code>, <code>e_alloc_cont_d_3D_matr()</code>, <code>e_alloc_cont_c_3D_matr()</code> and <code>e_alloc_cont_vp_3D_matr()</code> are the same as their fragmented counterparts except that the entire 3-D matrix lies in <i>contiguous</i> memory.</p> <p><code>e_init_frag_i_3D_matr()</code>, <code>e_init_frag_f_3D_matr()</code>, <code>e_init_frag_d_3D_matr()</code>, <code>e_init_frag_c_3D_matr()</code> and</p>

	<p><b>e_init_frag_vp_3D_matr()</b> initialise a 3-D fragmented matrix by returning a pointer of the appropriate type to a vector of <b>NDepth</b> null pointers.</p> <p>All of the above functions return NULL if the memory cannot be allocated.</p>
<b>NOTE</b>	3-D matrices can be allocated with one or more zero dimensions provided the standard function <b>free()</b> is able to safely free a vector allocated with zero length
<b>SEE ALSO</b>	2-D matrix allocation and 3-D matrix deallocation functions

FUNCTION	4-D matrix dynamic memory allocation functions
PLATFORM	All
DECLARATION	<pre> #include &lt;e_lib.h&gt;  int      ****e_alloc_frag_i_4D_matr(unsigned int NDepth,                                      unsigned int NWidth,                                      unsigned int NRows,                                      unsigned int NCols ); float    ****e_alloc_frag_f_4D_matr(unsigned int NDepth,                                      unsigned int NWidth,                                      unsigned int NRows,                                      unsigned int NCols ); double   ****e_alloc_frag_d_4D_matr(unsigned int NDepth,                                      unsigned int NWidth,                                      unsigned int NCols ); char     ****e_alloc_frag_c_4D_matr(unsigned int NDepth,                                      unsigned int NWidth,                                      unsigned int NRows,                                      unsigned int NCols ); void     *****e_alloc_frag_vp_4D_matr(unsigned int NDepth,  unsigned int NWidth,  unsigned int NRows,  unsigned int NCols );  int      ****e_alloc_cont_i_4D_matr(unsigned int NDepth,                                      unsigned int NWidth,                                      unsigned int NRows,                                      unsigned int NCols ); float    ****e_alloc_cont_f_4D_matr(unsigned int NDepth,                                      unsigned int NWidth,                                      unsigned int NRows,                                      unsigned int NCols ); double   ****e_alloc_cont_d_4D_matr(unsigned int NDepth,                                      unsigned int NWidth,                                      unsigned int NRows,                                      unsigned int NCols ); char     ****e_alloc_cont_c_4D_matr(unsigned int NDepth,                                      unsigned int NWidth,                                      unsigned int NRows,                                      unsigned int NCols ); void     *****e_alloc_cont_vp_4D_matr(unsigned int NDepth,  unsigned int NWidth,  unsigned int NRows,  unsigned int NCols );  int      ****e_init_frag_i_4D_matr(unsigned int NDepth); float    ****e_init_frag_f_4D_matr(unsigned int NDepth); double   ****e_init_frag_d_4D_matr(unsigned int NDepth); char     ****e_init_frag_c_4D_matr(unsigned int NDepth); void     *****e_init_frag_vp_4D_matr(unsigned int NDepth); </pre>
DESCRIPTION	<p>The functions <code>e_alloc_frag_i_4D_matr()</code>, <code>e_alloc_frag_f_4D_matr()</code>, <code>e_alloc_frag_d_4D_matr()</code>, <code>e_alloc_frag_c_4D_matr()</code> and <code>e_alloc_frag_vp_4D_matr()</code> return pointers to integer, float, double precision, character and void pointer four dimensional matrices respectively. Each matrix consists of a vector of <b>NDepth</b> pointers which point to <b>NWidth</b> by <b>NRows</b> by <b>NCols</b> three dimensional <i>fragmented</i> matrices. All elements are initialised to zero, or NULL in the case of <code>e_alloc_frag_vp_4D_matr()</code>.</p>

`e_alloc_cont_i_4D_matr()`, `e_alloc_cont_f_4D_matr()`,  
`e_alloc_cont_d_4D_matr()`, `e_alloc_cont_c_4D_matr()` and  
`e_alloc_cont_vp_4D_matr()` are the same as their fragmented counterparts  
except that the entire 4-D matrix lies in *contiguous* memory.

`e_init_frag_i_4D_matr()`, `e_init_frag_f_4D_matr()`,  
`e_init_frag_d_4D_matr()`, `e_init_frag_c_4D_matr()` and  
`e_init_frag_vp_4D_matr()` initialise a 4-D fragmented matrix by returning  
a pointer of the appropriate type to a vector of **NDepth** null pointers.

All of the above functions return NULL if the memory cannot be allocated.

**NOTE** 4-D matrices can be allocated with one or more zero dimensions provided the  
standard function **free()** is able to safely free a vector allocated with zero length

**SEE ALSO** 3-D matrix allocation and 4-D matrix deallocation functions

FUNCTION	Dynamic memory incrementation functions
PLATFORM	All
DECLARATION	<pre> #include &lt;e_lib.h&gt;  int    *e_incr_i_vect(unsigned int length, int *vector,                       unsigned int LengthIncr); float  *e_incr_f_vect(unsigned int length, float *vector,                       unsigned int LengthIncr); double *e_incr_d_vect(unsigned int length, double *vector,                       unsigned int LengthIncr); char   *e_incr_c_vect(unsigned int length, char *vector,                       unsigned int LengthIncr); void   *e_incr_vp_vect(unsigned int length, void *vector,                       unsigned int LengthIncr);  int    **e_incr_frag_i_matr(unsigned int NRows,                            int **matrix,                            unsigned int NRowsIncr); float  **e_incr_frag_f_matr(unsigned int NRows,                            float **matrix,                            unsigned int NRowsIncr); double **e_incr_frag_d_matr(unsigned int NRows,                            double **matrix,                            unsigned int NRowsIncr); char   **e_incr_frag_c_matr(unsigned int NRows,                            char **matrix,                            unsigned int NRowsIncr); void   ***e_incr_frag_vp_matr(unsigned int NRows,                               void ***matrix,                               unsigned int NRowsIncr);  int    **e_incr_cont_i_matr(unsigned int NRows,                            unsigned int NCols,                            int **matrix,                            unsigned int NRowsIncr); float  **e_incr_cont_f_matr(unsigned int NRows,                            unsigned int NCols,                            float **matrix,                            unsigned int NRowsIncr); double **e_incr_cont_d_matr(unsigned int NRows,                            unsigned int NCols,                            double **matrix,                            unsigned int NRowsIncr); char   **e_incr_cont_c_matr(unsigned int NRows,                            unsigned int NCols,                            char **matrix,                            unsigned int NRowsIncr); void   ***e_incr_cont_vp_matr(unsigned int NRows,                               unsigned int NCols,                               void ***matrix,                               unsigned int NRowsIncr); </pre>
DESCRIPTION	<p>The functions <code>e_incr_i_vect()</code>, <code>e_incr_f_vect()</code>, <code>e_incr_d_vect()</code>, <code>e_incr_c_vect()</code> and <code>e_incr_vp_vect()</code> increase the memory allocated for integer, float, double precision, character and void pointer vectors respectively. The vector <code>vector</code> initially has length <code>length</code> and this is increased to <code>length + LengthIncr</code> if <code>length</code> is a multiple of <code>LengthIncr</code>. If no additional memory is allocated then <code>vector</code> is returned unchanged. If the length is increased, the return value is a pointer to the new vector with the additional components set to</p>

zero or NULL. Should allocation fail, then NULL is returned. On entry, **vector** must be set to NULL if **length** equals zero.

The functions **e\_incr\_i\_frag\_matr()**, **e\_incr\_f\_frag\_matr()**, **e\_incr\_d\_frag\_matr()**, **e\_incr\_c\_frag\_matr()** and **e\_incr\_vp\_frag\_matr()** increase the number of rows allocated for integer, float, double precision, character and void pointer fragmented matrices respectively. The matrix **matrix** initially has **NRows** rows and this is increased to **NRows + NRowsIncr** if **NRows** is a multiple of **NRowsIncr**. If no additional memory is allocated then **matrix** is returned unchanged. If the number of rows is increased, the return value is a pointer to the new matrix with the additional row pointers set to NULL. Should allocation fail, then NULL is returned. On entry, **matrix** must be set to NULL if **NRows** equals zero.

The functions **e\_incr\_i\_cont\_matr()**, **e\_incr\_f\_cont\_matr()**, **e\_incr\_d\_cont\_matr()**, **e\_incr\_c\_cont\_matr()** and **e\_incr\_vp\_cont\_matr()** are the contiguous memory counterparts of the above fragmented matrix incrementation functions. **NCols** is the number of columns. The components of any additional rows are set to zero and, as above, if **NRows** is zero then **matrix** must equal NULL.

#### NOTE

Care should be taken when choosing values for **LengthIncr** and **NRowsIncr**. Values which are too small will lead to excessive reallocation of memory, whereas values which are too large will result in memory being allocated unnecessarily.

#### SEE ALSO

Vector and matrix dynamic memory allocation functions

#### EXAMPLE

The following example allocates just sufficient memory to store a string read from the command line and then prints it to standard output.

```
int    i = 0;
char *string = NULL;
char  c;

while ((c = getchar()) != '\n')
{
    string = e_incr_c_vect(i, string, 1);
    string[i++] = c;
}

string = e_incr_c_vect(i, string, 1);
printf("%s\n", string);
```

<b>FUNCTION</b>	Vector dynamic memory deallocation function
<b>PLATFORM</b>	All
<b>DECLARATION</b>	<pre>#include &lt;e_lib.h&gt;  void e_free(void *vector);</pre>
<b>DESCRIPTION</b>	<b>e_free()</b> frees the memory pointed to by <b>vector</b> . <b>vector</b> can be a null pointer, in which case <b>e_free()</b> simply returns.
<b>NOTE</b>	<b>e_free()</b> can be used to free a vector allocated with zero length provided the standard function <b>free()</b> is able to safely deallocate such a vector
<b>SEE ALSO</b>	Vector dynamic memory allocation functions

<b>FUNCTION</b>	Matrix dynamic memory deallocation functions
<b>PLATFORM</b>	All
<b>DECLARATION</b>	<pre>#include &lt;e_lib.h&gt;  void e_free_frag_i_matr(unsigned int NRows,    int **matrix); void e_free_frag_f_matr(unsigned int NRows,    float **matrix); void e_free_frag_d_matr(unsigned int NRows,    double **matrix); void e_free_frag_c_matr(unsigned int NRows,    char **matrix); void e_free_frag_vp_matr(unsigned int NRow,     void **matrix);  void e_free_cont_i_matr(unsigned int NRows,    int **matrix); void e_free_cont_f_matr(unsigned int NRows,    float **matrix); void e_free_cont_d_matr(unsigned int NRows,    double **matrix); void e_free_cont_c_matr(unsigned int NRows,    char **matrix); void e_free_cont_vp_matr(unsigned int NRows,    void **matrix);</pre>
<b>DESCRIPTION</b>	<p><code>e_free_frag_i_matr()</code>, <code>e_free_frag_f_matr()</code>, <code>e_free_frag_d_matr()</code>, <code>e_free_frag_c_matr()</code> and <code>e_free_frag_vp_matr()</code> free the memory allocated for the fragmented matrix <b>matrix</b> with <b>NRows</b> rows.</p> <p><code>e_free_cont_i_matr()</code>, <code>e_free_cont_f_matr()</code>, <code>e_free_cont_d_matr()</code>, <code>e_free_cont_c_matr()</code> and <code>e_free_cont_vp_matr()</code> deallocate contiguously allocated matrices.</p> <p>In all cases, no deallocation is performed if <b>matrix</b> is NULL.</p>
<b>NOTE</b>	Matrices allocated with one or more zero dimensions can be deallocated provided the standard function <b>free()</b> is able to safely free a vector allocated with zero length
<b>SEE ALSO</b>	Matrix dynamic memory allocation functions



<b>FUNCTION</b>	3-D matrix dynamic memory deallocation functions
<b>PLATFORM</b>	All
<b>DECLARATION</b>	<pre> #include &lt;e_lib.h&gt;  void e_free_frag_i_3D_matr(unsigned int NDepth,                            unsigned int NRows,                            int ***matrix3D ); void e_free_frag_f_3D_matr(unsigned int NDepth,                            unsigned int NRows,                            float ***matrix3D ); void e_free_frag_d_3D_matr(unsigned int NDepth,                            unsigned int NRows,                            double ***matrix3D ); void e_free_frag_c_3D_matr(unsigned int NDepth,                            unsigned int NRows,                            char ***matrix3D ); void e_free_frag_vp_3D_matr(unsigned int NDepth,                             unsigned int NRows,                             void ****matrix3D );  void e_free_cont_i_3D_matr(unsigned int NDepth,                            unsigned int NRows,                            int ***matrix3D ); void e_free_cont_f_3D_matr(unsigned int NDepth,                            unsigned int NRows,                            float ***matrix3D ); void e_free_cont_d_3D_matr(unsigned int NDepth,                            unsigned int NRows,                            double ***matrix3D ); void e_free_cont_c_3D_matr(unsigned int NDepth,                            unsigned int NRows,                            char ***matrix3D ); void e_free_cont_vp_3D_matr(unsigned int NDepth,                              unsigned int NRows,                              void ****matrix3D ); </pre>
<b>DESCRIPTION</b>	<p><b>e_free_frag_i_3D_matr()</b>, <b>e_free_frag_f_3D_matr()</b>, <b>e_free_frag_d_3D_matr()</b>, <b>e_free_frag_c_3D_matr()</b> and <b>e_free_frag_vp_3D_matr()</b> free the memory allocated for the fragmented three dimensional matrix pointed to by <b>matrix3D</b>. The 3-D matrix is assumed to consist of a vector of <b>NDepth</b> pointers which point to fragmented matrices with <b>NRows</b> rows.</p> <p><b>e_free_cont_i_3D_matr()</b>, <b>e_free_cont_f_3D_matr()</b>, <b>e_free_cont_d_3D_matr()</b>, <b>e_free_cont_c_3D_matr()</b> and <b>e_free_cont_vp_3D_matr()</b> deallocate contiguously allocated 3-D matrices.</p> <p>In all cases, no deallocation is performed if <b>matrix3D</b> is NULL.</p>
<b>NOTE</b>	3-D matrices allocated with one or more zero dimensions can be deallocated provided the standard function <b>free()</b> is able to safely free a vector allocated with zero length
<b>SEE ALSO</b>	3-D matrix dynamic memory allocation functions

FUNCTION	4-D matrix dynamic memory deallocation functions
PLATFORM	All
DECLARATION	<pre> #include &lt;e_lib.h&gt;  void e_free_frag_i_4D_matr(unsigned int NDepth,                            unsigned int NWidth,                            unsigned int NRows,                            int ****matrix4D ); void e_free_frag_f_4D_matr(unsigned int NDepth,                            unsigned int NWidth,                            unsigned int NRows,                            float ****matrix4D ); void e_free_frag_d_4D_matr(unsigned int NDepth,                            unsigned int NWidth,                            unsigned int NRows,                            double ****matrix4D ); void e_free_frag_c_4D_matr(unsigned int NDepth,                            unsigned int NWidth,                            unsigned int NRows,                            char ****matrix4D ); void e_free_frag_vp_4D_matr(unsigned int NDepth,                             unsigned int NWidth,                             unsigned int NRows,                             void ****matrix4D );  void e_free_cont_i_4D_matr(unsigned int NDepth,                            unsigned int NWidth,                            unsigned int NRows,                            int ****matrix4D ); void e_free_cont_f_4D_matr(unsigned int NDepth,                            unsigned int NWidth,                            unsigned int NRows,                            float ****matrix4D ); void e_free_cont_d_4D_matr(unsigned int NDepth,                            unsigned int NWidth,                            unsigned int NRows,                            double ****matrix4D ); void e_free_cont_c_4D_matr(unsigned int NDepth,                            unsigned int NWidth,                            unsigned int NRows,                            char ****matrix4D ); void e_free_cont_vp_4D_matr(unsigned int NDepth,                              unsigned int NWidth,                              unsigned int NRows,                              void ****matrix4D ); </pre>
DESCRIPTION	<p><b>e_free_frag_i_4D_matr()</b>, <b>e_free_frag_f_4D_matr()</b>, <b>e_free_frag_d_4D_matr()</b>, <b>e_free_frag_c_4D_matr()</b> and <b>e_free_frag_vp_4D_matr()</b> free the memory allocated for the fragmented four dimensional matrix pointed to by <b>matrix4D</b>. The 4-D matrix is assumed to consist of a vector of <b>NDepth</b> pointers which point to <b>NWidth</b> by <b>NRows</b> by <b>NCols</b> 3-D fragmented matrices (<b>NCols</b> is not an argument).</p> <p><b>e_free_cont_i_4D_matr()</b>, <b>e_free_cont_f_4D_matr()</b>, <b>e_free_cont_d_4D_matr()</b>, <b>e_free_cont_c_4D_matr()</b> and <b>e_free_cont_vp_4D_matr()</b> deallocate contiguously allocated 4-D matrices.</p> <p>In all cases, no deallocation is performed if <b>matrix4D</b> is NULL.</p>

<b>NOTE</b>	4-D matrices allocated with one or more zero dimensions can be deallocated provided the standard function <b>free()</b> is able to safely free a vector allocated with zero length
<b>SEE ALSO</b>	4-D matrix dynamic memory allocation functions

FUNCTION	Miscellaneous functions
PLATFORM	All
DECLARATION	<pre> #include &lt;e_lib.h&gt;  int e_GetNCutInterfaces(MESH *mesh);  int e_GetNInterSubDomNodes(MESH *mesh,                            int *NInterSubDomNodes,                            int *WeightedNInterSubDomNodes);  int e_GetInterSubDomNodes(MESH *mesh,                           int *NInterSubDomNodes,                           int **InterSubDomNodes,                           int **NNodeSubDoms,                           int ***NodeSubDoms ); void e_FreeInterSubDomNodes(int NInterSubDomNodes,                            int *InterSubDomNodes,                            int **NNodeSubDoms,                            int ***NodeSubDoms );  int e_SubDomBoundaries2PolyPolygon(MESH *mesh,                                    int **PolygonVertices,                                    int **NPolygonVertices,                                    int *NPolygons ); void e_FreeSubDomBoundaries2PolyPolygon(int *PolygonVertices,  int *NPolygonVertices); </pre>
DESCRIPTION	<p><b>e_GetNCutInterfaces ()</b> uses the decomposition data of the mesh pointed to by <b>mesh</b> to return the number of cut interfaces. It is assumed that the mesh is composed of only one element type. The return value is -1 if an error occurs.</p> <p><b>e_GetNInterSubDomNodes ()</b> calculates the number of inter-subdomain nodes in a decomposed mesh and a weighted number of inter-subdomain nodes. It is assumed that the mesh is composed of elements of only one element type. On return, <b>NInterSubDomNodes</b> points to the number of nodes shared by more than one subdomain. If a node is shared by <math>N</math> subdomains then it is given a weighting of <math>N - 1</math>. These weighted values are summed to give the value <b>*WeightedNInterSubDomNodes</b>. The return value is 1 if the function call is successful, and 0 otherwise.</p> <p><b>e_GetInterSubDomNodes ()</b> determines which finite element nodes are shared by the subdomains of a decomposed mesh. It is assumed that the mesh is composed of only one element type. On return <b>NInterSubDomNodes</b> points to the number of inter-subdomain nodes. <b>*InterSubDomNodes</b> is a vector that stores the indices of these nodes. The vector <b>*NNodeSubDoms</b> contains the number of subdomains to which each inter-subdomain node is adjacent and <b>**NodeSubDoms</b> is a matrix containing the indices of the subdomains adjacent to each inter-subdomain node. For example, suppose node <math>i</math> is the <math>k</math>-th inter-subdomain node and that it is shared by the three subdomains <math>S_1, S_2</math> and <math>S_3</math>. Then <b>(*NNodeSubDoms) [k - 1] = 3</b> and <b>(**NodeSubDoms) [k - 1] [j - 1] = <math>S_j</math>, <math>j = 1, 2, 3</math></b>. Note that <b>e_GetInterSubDomNodes ()</b> allocates the memory necessary to store <b>*InterSubDomNodes</b>, <b>*NNodeSubDoms</b> and <b>**NodeSubDoms</b>. The return value is 1 if the operation is successful, else 0.</p> <p><b>e_FreeInterSubDomNodes ()</b> frees the memory allocated by <b>e_GetInterSubDomNodes ()</b>.</p>

**e\_SubDomBoundaries2PolyPolygon()** determines the mesh-points which lie on the subdomain boundaries of a mesh in an order that can be used by the **polypolygon()** Windows graphics function. The subdomains can be fragmented. This function is useful when, say, a thick line is to be drawn around the subdomains in a Windows mesh display program. It is assumed that the mesh is composed solely of planar elements of one element type. **NPolygon** points to the number of polygons in the polypolygon and **\*NPolygonsVertices** is a vector that stores the number of vertices in each polygon. The vector **\*PolygonVertices** contains the node indices of each vertex of each polygon in the polypolygon. Note that the memory to store the vectors **\*NPolygonsVertices** and **\*PolygonVertices** is allocated by **e\_SubDomBoundaries2PolyPolygon()**. The return value is the number of polygon vertices to be plotted if the function call is successful, otherwise it is 0.

**e\_FreeSubDomBoundaries2PolyPolygon()** frees the memory allocated by **e\_SubDomBoundaries2PolyPolygon()**.

**NOTE**

**e\_GetNCutInterfaces()**, **e\_GetNInterSubDomNodes()** and **e\_SubDomBoundaries2PolyPolygon()** are computationally expensive and should be used sparingly.

FUNCTION	Mathematical functions
PLATFORM	All
DECLARATION	<pre> #include &lt;e_lib.h&gt;  int e_LUFact(int Order, double **A, int *pivot); int e_LUSolve(int Order, double **LU, double *b, int *pivot);  unsigned long e_Factorial(unsigned long Number);  double e_FactorialDP(unsigned long Number); double e_VectDotProd(unsigned int Order, double *x, double *y);  void e_VectScale(unsigned int Order, double Factor,                  double *x, double *y); void e_MatrTransp(unsigned int Order, double **A,                  double **B ); void e_MatrTranspNS(unsigned int NRows, unsigned int NCols,                    double **A, double **B); void e_MatrVectMult(unsigned int Order, double **A,                    double *x, double *y); void e_VectMatrMult(unsigned int Order, double **A,                    double *x, double *y); void e_MatrMult(unsigned int Order, double **A,                double **B, double **C ); void e_MatrMultNS(unsigned int NRowsA,                  unsigned int NColsA_NRowsB,                  unsigned int NColsB, double **A,                  double **B, double **C );  #define E_ABS(X)      (((X) &gt; 0) ? (X) : (-(X))) #define E_MIN(A,B)    (((A) &lt; (B)) ? (A) : (B)) #define E_MAX(A,B)    (((A) &gt; (B)) ? (A) : (B)) #define E_SQUARE(X)   ((X)*(X)) </pre>
DESCRIPTION	<p>The function <b>e_LUFact()</b> performs LU factorisation on the square matrix <b>A</b> of order <b>Order</b>. <b>pivot</b> is a vector which stores the new row order resulting from partial pivoting. <b>A</b> is overwritten by the factorisation matrices. If <b>pivot</b> is NULL then no partial pivoting is performed. The return value is 1 if <b>A</b> is non-singular and the operation is successful, and 0 otherwise. (See Ciarlet, P.G., <i>Introduction to Numerical Linear Algebra and Optimisation</i>, Cambridge University Press, 1989.)</p> <p><b>e_LUSolve</b> solves the matrix equation <math>Ax = b</math> where LU is the factorisation matrix of <b>A</b> obtained from <b>e_LUFact()</b>. If <b>e_LUFact()</b> was called using partial pivoting then the <b>pivot</b> obtained from <b>e_LUFact()</b> must be supplied to <b>e_LUSolve()</b>. If there was no partial pivoting then <b>pivot</b> must be set to NULL. The vector <b>b</b> is overwritten by the solution <math>x</math>. The return value is 1 if the equation is solved successfully, otherwise it is 0.</p> <p><b>e_Factorial()</b> and <b>e_FactorialDP()</b> return the factorial of the integer <b>Number</b>. The maximum permitted value of <b>Number</b> for <b>e_Factorial()</b> is 12.</p> <p><b>e_VectDotProd()</b> returns the scalar product of the two vectors <b>x</b> and <b>y</b> of order <b>Order</b> and <b>e_VectScale()</b> multiplies each of the components of <b>x</b> by <b>Factor</b> to give the vector <b>y</b>. For both of these functions, <b>x</b> and <b>y</b> can point to the same address.</p>

**e\_MatrTransp()** transposes the square matrix **A** of order **Order** to give the square matrix **B**. **e\_MatrTranspNS()** transposes the **NRows** by **NCols** matrix **A** to give the **NCols** by **NRows** matrix **B**.

**e\_MatrVectMult()** premultiplies **x** by **A** to give the vector **y** (i.e.  $Ax = y$ ) and **e\_VectMatrMult()** postmultiplies **x** by **A** to give **y** (i.e.  $x^T A = y^T$ ).

**e\_MatrMult()** multiplies the square matrices **A** and **B** to give the matrix **C** (i.e.  $AB = C$ ) whereas **e\_MatrMultNS()** performs the same calculation but for **A** with dimensions **NRowsA** by **NColsA\_NRowsB** and **B** with dimensions **NColsA\_NRowsB** by **NColsB**. Hence, at the minimum, **C** must be **NRowsA** by **NColsB**.

The macro function **E\_ABS()** returns the absolute value of its argument **X**. **E\_MIN()** and **E\_MAX()** return the minimum and maximum of **A** and **B** respectively and **E\_SQUARE()** returns the square of **X**.

## **Appendix 2 – An Example Data File Format Conversion Program**

This appendix lists the source code of a program which converts mesh data from the format previously used by BHV Topping to the E-Lib format. The program illustrates the use of the E-Lib library for a command line ANSI C program.



```

/* To convert a BHV Topping mesh file (PROJECTNAME.dat) to an E-Lib mesh
   definition file (PROJECTNAME.mdf).
/* NOTE: Only elements of types TRIANG1 and QUAD1 are converted.
*/

#define __E_CLANSIC__

#include <e_lib.h>
#include <stdio.h>
#include <string.h>

int main()
{
    int i;
    int LineNumber = 0;
    int NLinks;

    char ProjectName[E_LINE_MAX + 1];
    char InputFileName[E_LINE_MAX + 5];
    char InputLine[E_LINE_MAX + 1];
    char *pNewLineChar;

    FILE *fp;
    MESH mesh;

    /* Read in the project name and open the input file */

    printf("Enter the project name: ");
    scanf("%s", ProjectName);
    strcpy(InputFileName, ProjectName);
    strcat(InputFileName, ".dat");

    if ((fp = fopen(InputFileName, "r")) == NULL)
        e_FileError(InputFileName, "cannot open input file");

    /* Initialise the MESH structure */

    for (i = 0; i < E_NKEYWORDS; i++)
        mesh.KeywordSet[i] = FALSE;

    for (i = 0; i < E_NSUBSTRUCTS; i++)
        mesh.SubStructSet[i] = FALSE;

    /* Read in and set the title */

    mesh.Title = e_alloc_c_vect(E_LINE_MAX + 1);
    e_GetNextLine(InputLine, &LineNumber, InputFileName, fp);

    /* convert the '\n' to '\0' */
    if ((pNewLineChar = strchr(InputLine, '\n')) != NULL)
        *pNewLineChar = '\0';

    strcpy(mesh.Title, InputLine);
    mesh.KeywordSet[E_KW_NUM_TITLE] = TRUE;

    /* Skip over unwanted data */
    e_GetNextLine(InputLine, &LineNumber, InputFileName, fp);

    /* Read in the number of mesh-points and the number of TRIANG1 elements */

    e_GetNextLine(InputLine, &LineNumber, InputFileName, fp);

```

```

        if (sscanf(InputLine,"%d %d %d %d %d",
                    &mesh.NMeshPoints,&mesh.NElemsTriangl,&NLinks) != 3)
        {
            e_LineError(LineNumber,"cannot read data");
        }

        if (mesh.NMeshPoints > 0)
            mesh.KeywordSet[E_KW_NUM_NMESHPOINTS] = TRUE;
        else
            e_LineError(LineNumber,"ill-defined number of mesh-points");

        mesh.NNodes = mesh.NMeshPoints;
        mesh.KeywordSet[E_KW_NUM_NNODES] = TRUE;

        if (mesh.NElemsTriangl > 0)
            mesh.KeywordSet[E_KW_NUM_NELEMS_TRIANG1] = TRUE;
        else if (mesh.NElemsTriangl < 0)
            e_LineError(LineNumber,"ill-defined number of TRIANG1 elements");

/* Read in the number of QUAD1 elements */

        e_GetNextLine(InputLine,&LineNumber,InputFileName,fp);

        if (sscanf(InputLine,"%d",&mesh.NElemsQuad1) != 1)
            e_LineError(LineNumber,"cannot read data");

        if (mesh.NElemsQuad1 > 0)
            mesh.KeywordSet[E_KW_NUM_NELEMS_QUAD1] = TRUE;
        else if (mesh.NElemsQuad1 < 0)
            e_LineError(LineNumber,"ill-defined number of QUAD1 elements");

/* Read in the mesh-point coordinates */

        mesh.MeshPointCoords = e_alloc_cont_d_matr(mesh.NMeshPoints,3);
        for (i = 0; i < mesh.NMeshPoints; i++)
        {
            e_GetNextLine(InputLine,&LineNumber,InputFileName,fp);

            if (sscanf(InputLine,"%d %lf %lf %lf",
                        &mesh.MeshPointCoords[i][0],
                        &mesh.MeshPointCoords[i][1],
                        &mesh.MeshPointCoords[i][2]) != 3)
            {
                e_LineError(LineNumber,"cannot read data");
            }
        }

        mesh.KeywordSet[E_KW_NUM_MESHPOINT_COORDS] = TRUE;

/* Read in the node indices of the TRIANG1 elements */

        if (mesh.KeywordSet[E_KW_NUM_NELEMS_TRIANG1])
        {
            mesh.NodesTriangl =
                e_alloc_cont_i_matr(mesh.NElemsTriangl,3);

            for(i = 0; i < mesh.NElemsTriangl; i++)
            {
                e_GetNextLine(InputLine,&LineNumber,InputFileName,fp);

                if (sscanf(InputLine,"%d %d %d %d",
                            &mesh.NodesTriangl[i][0],
                            &mesh.NodesTriangl[i][1],
                            &mesh.NodesTriangl[i][2]) != 3)
                {
                    e_LineError(LineNumber,"cannot read data");
                }
            }
        }

```

```

        mesh.KeywordSet[E_KW_NUM_NODES_TRIANG1] = TRUE;
    }

    /* Skip over unwanted data */

    for (i = 0; i < NLinks; i++)
        e_GetNextLine(InputLine, &LineNumber, InputFileName, fp);

    /* Read in the node indices of the QUAD1 elements */

    if (mesh.KeywordSet[E_KW_NUM_NELEMS_QUAD1])
    {
        mesh.NodesQuad1
            = e_alloc_cont_i_matr(mesh.NElemsQuad1, 4);

        for (i = 0; i < mesh.NElemsQuad1; i++)
        {
            e_GetNextLine(InputLine, &LineNumber, InputFileName, fp);

            if (sscanf(InputLine, "%*d %d %d %d %d",
                        &mesh.NodesQuad1[i][0],
                        &mesh.NodesQuad1[i][1],
                        &mesh.NodesQuad1[i][2],
                        &mesh.NodesQuad1[i][3]) != 4)
            {
                e_LineError(LineNumber, "cannot read data");
            }
        }

        mesh.KeywordSet[E_KW_NUM_NODES_QUAD1] = TRUE;
    }

    /* Create the E-Lib mesh definition file */

    e_PutMeshData(&mesh, ProjectName);

    /* Close the input file and return */

    fclose(fp);
    return 0;
}

```