NASA Contractor Report 181728

The Computational Structural Mechanics Testbed Generic Structural-Element Processor Manual

Gary Stanley and Shahram Nour-Omid

Lockhed Missiles and Space Company, Inc.
Palo Alto, California

Contract NAS1-18444

March 1990

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

Preface

The purpose of this manual is to document the usage and development of structural finite element processors based on the CSM Testbed's Generic Element Processor (GEP) template. By convention, such processors have names of the form ESi, where i is an integer.

This manual is therefore intended for both Testbed users who wish to invoke ES processors during the course of a structural analysis, and Testbed developers who wish to construct new element processors (or modify existing ones).

The contents of this manual were compiled by Gary M. Stanley of Lockheed Palo Alto Research Laboratory, who is also the principal author. Contributors include:

Lockheed Palo Alto Research Laboratory

Bryan J. Hurlbut
Shahram Nour-Omid
Charles R. Rankin
Gary M. Stanley
Lyle W. Swenson, Jr.
David S. Kang (currently at Charles Stark Draper Laboratory Inc.)

NASA Langley Research Center

Norman F. Knight, Jr.

Analytical Services and Materials, Inc.

Mohammad A. Aminpour

Lockheed Engineering and Science Company

Christine G. Lotts
Susan L. McCleary

Update Log	Date
Initial Draft	January 1989
Revised Draft	May 1989

Table of Contents

1. INTRODUCTION

CHAPTER OUTLINE

1.1 What is the Generic Element Processor (GEP)?

The Computational Structural Mechanics (CSM) Testbed software system is a framework for structural and numerical methods research (see ref. 1). The CSM Testbed utilizes a high-level command language (ref. 2) and data manager (ref. 3) and allows the coupling of independent FORTRAN processors together such that specific structural analysis functions may be performed. Various analysis processors and specific pre-processing, solution, post-processing, and utility procedures are being developed by independent researchers (*e.g.*, see ref. 4). These processors and procedures utilize the data manager for archiving and retrieving data to and from data libraries which may be interrogated by the user (see ref. 5). The development, implementation, and assessment of finite element technology for structural analysis is performed using the concept of a generic element processor.

The Generic Element Processor (GEP) is the standard (*i.e.*, recommended) software mechanism for accessing and implementing structural finite elements in the CSM Testbed. Actually, the GEP is not really a "processor" at all, but rather a generic *template* for implementing a multitude of structural-element (ES*) processors, each of which may be viewed as an independent module in the CSM Testbed. What each of these independent ES*i* processors (*e.g.*, ES1, ES2, ES3, ...) have in common is a standard software driver, or "shell", which ensures that they all "speak" the same *command language* and create the same *data structures*, regardless of how different they are on the inside. It is by virtue of the generic *shell*, that element users may access all ES*i* processors in precisely the same manner, and that element developers may implement new elements using precisely the same interface routines, regardless of their different internal characteristics. In summary, the GEP is *an extendible and easy-to-use vehicle for integrated element research, development, and application within the CSM Testbed.*

Note that there is no general rule regarding how many different element types may be implemented within a specific ES*i* processor. For example, some element processors may

* ES is used rather than SE as an abbreviation for Structural Element, so that all element processor names will begin with the letter E. Eventually, the Testbed will include other element processor types besides structural; for example fluid elements (EF), thermal elements (ET) and control — or constraint — elements (EC). Just think of the second letter as a subscript (*e.g.*, E_s, E_f, E_t, ...).

Note that there is no general rule regarding how many different element types may be implemented within a specific ES*i* processor. For example, some element processors may contain only a single structural element type, while others may contain a family of elements of a particular class (*e.g.*, 4-, 9-, and 16-node shell elements). Still other element processors may contain an entire library of finite elements, including various beam, shell, and solid elements that may (or may not) be based on a common formulation. The diversity of an ES*i* processor depends only on the diversity of the developer.

1.2 What are the key features of the GEP?

In addition to providing a common interface for element users and developers, the generic ES processor shell performs the following additional features for the developer:

- It performs all system (*i.e.*, architectural) functions required for compatibility with the CSM Testbed.
- It handles most (if not all) of the global database transactions required by element developers (see Chapter 6).
- It can automatically perform all operations needed for large-rotation (small strain) geometrically nonlinear analysis, using a built-in *corotational* algorithm (see Chapter 4).
- It will provide a common generic interface to built-in *constitutive utilities*, which will facilitate nonlinear material modeling (see Chapter 5).
- It performs all transformations from the element intrinsic coordinate system to computational (nodal freedom) coordinate systems.
- It has an option to suppress automatically extraneous global freedoms based on element-type participation (see Sections 2.2 and 2.5).
- It accommodates the implementation of non-standard elements, which are treated as "black boxes" which must perform most of the above operations themselves (as described in Section 3.4).

and, for the user, it also facilitates the writing of analysis procedures by providing

- A high-level procedure interface (called procedure ES), which automatically invokes all pertinent ES processors with a single generic call (as described in Section 2.3).

In summary, the Generic ES processor performs many of the standard overhead functions that would otherwise (unnecessarily) burden the element developer and, at the same time, provides a convenient interface for the user.

1.3 What are some of the limitations of the GEP?

The GEP is designed to accommodate "conventional" element types more fully than "unconventional" types. This means that it may be harder to implement some elements than others; however, there is usually a way to implement just about any element. For example, most of the built-in functions performed by the ES processor shell (listed above) only work for *standard* elements, which include a variety of 1-D, 2-D and 3-D element types (described in Section 3.4). Elements that do not fit within the standard mold, are referred to as non-standard, or "wild" elements. Wild elements may be implemented either as "black boxes" within an ES processor; or by building a custom-made ES processor shell. While this latter option involves considerably more work, it is a viable option, and as long as the special-purpose ES processor shell conforms to the standard command language and output datasets used by standard ES processors, the result will look the same to the user (*e.g.*, at the procedure level), and hence will be compatible with the CSM Testbed.

1.4 How is the GEP organized?

The organization (software design) of the GEP for structural-element (ES) processors is illustrated in Figure 1.1, and the internal design of the Generic ES processor shell is described in Chapter 7. The design features a standard outer software "shell" which processes user commands (such as FORM STIFFNESS and FORM FORCE), handles all input/output from/to the global database (through calls to the GAL data manager), and performs the additional CSM-oriented functions described in Section 1.2. Inside this shell is the personalized "kernel" supplied by the element developer, which may be written in just about any style or granularity, provided that it is in FORTRAN. Finally, a *standard* set of kernel routines must be completed by the developer to connect the shell and the developer's kernel for each of the many intrinsic element functions.

The *standard kernel routines* transform data structures gathered from the database by the ES shell into a form accessible to the element developer's kernel routines, and later transform results produced by the kernel into standard data structures which are output by the ES shell to the global database.

Note the close correspondence between ES processor commands and standard kernel routines. For example, commands such as FORM STIFFNESS/MATL and FORM FORCE/INT are translated by the ES shell into calls to kernel routines, ES0KM and ES0FI, respectively. Such correspondences are summarized in Table 3.2.

1.5 How does the user perform structural analysis using the GEP?

All structural element processors based on the GEP template which typically have names beginning with ES should look alike to the user. The user may invoke one or more of these processors either *directly* — by running each ES processor, one at a time, and issuing the proper commands; or *indirectly* — by using the high-level ES procedure interface, which automatically invokes all of the required ES processors for a given problem. The only exception is for *element definition*, in which case each ES processor *must* be run individually (either directly or indirectly using the ES procedure interface). Also, ES processors must be used *in conjunction* with Testbed processor ELD, which defines element connectivity and initializes various global datasets.

Information on the theory and use of specific element-types within individual ES processors is provided by the corresponding element developer. The user may find specific documentation for various ES*i* processors in the CSM Testbed User's Manual (ref. 4). The user interface to the GEP is described in Chapter 2 of this manual.

1.6 How does the developer add new structural elements using the GEP?

To add a new element processor to the CSM Testbed, using the generic ES processor shell, the developer should follow the "recipe" described in Section 3.5. Briefly, the procedure consists of (i) completing the set of standard kernel routines, which for structural elements include such functions as: element definition, stiffness formation, force formation, strain computation, mass formation, and various transformations; (ii) linking these new routines with the ES shell to form a new ES processor (usually a separate executable at first); and (iii) modifying model-definition procedures to refer to the new ES processor for testing and application.

The main responsibility of the element developer is to supply standard kernel routines. *Templates* for these routines, which contain standard (but general) argument lists and formal declaration statements, are provided in source-code form, so that the developer only needs to fill-in the interior of each functional routine (see Section 3.2 for a summary). The interior of these kernel routines should do whatever is necessary to connect the standard argument variables to the developer's personal (kernel) subroutine argument lists — unless the developer wishes to perform all computations directly within the kernel routines.

Once the developer has finished the standard kernel routines and underlying kernel utilities, creation of a new ES processor simply involves linking the developer-supplied code to a pre-existing *object library* corresponding to the generic ES shell. This process enables each ES processor shell to be developed independently, in an analogous manner to the development of the CSM Testbed *architecture*. The developer interface is described in Chapter 3 of this manual.

1.7 How does the GEP differ from what was previously in the Testbed?

The initial version of the CSM Testbed (referred to as NICE/SPAR) processed elements in an entirely different manner than the present approach. In the initial version, each element function (*e.g.*, stiffness, force, mass, ...) corresponded to a different element processor (*e.g.*, EKS, GSF, M, ...), and each of these element processors embedded *all elements* implemented in the Testbed. This software architecture not only made it very difficult to add new elements (especially more than one at a time), but because of the orientation of these older processors toward *linear* analysis, it presented serious obstacles for upgrading the Testbed to *nonlinear* analysis.

In the present approach, all of the older element processors, with the exception of processors ELD and K, have been replaced by the Generic Structural-Element (ES) Processor series (ES1, ES2, ES3, ...), each of which performs *all element functions* for a given element type (or types), and each of which is usually constructed by an individual element developer.

The exceptions mentioned above, processors ELD and K, have been temporarily retained for expediency. Processor ELD is still used to define *element connectivity*, and processor K is still used as an *element matrix assembler*. Even though elements associated with ES processors are viewed as "experimental elements" by processors ELD and K, the restriction of the original experimental-element facility on the number of different ES processors that may be employed simultaneously in the same model has been removed.

Another difference is that processor K is used *strictly* as an element assembler in conjunction with ES processors; *i.e.*, it is not used to perform the various element transformations and expansions that it performed on earlier Testbed elements. Instead, such operations are all performed within the generic ES processor shell — before assembly. Finally, the generic ES processor utilizes a *data structure similar to the one formerly employed by "experimental elements"*.

1.8 What is the purpose of this document?

The present document is intended to serve as a multi-purpose reference manual for the Generic Element Processor (GEP), with special emphasis on structural-element (ES) processors that are constructed using the GEP template. Thus, it contains chapters on: the User Interface (Chapter 2), which provides instructions and examples on how to employ ES processors to perform structural analysis within the Testbed; the Developer Interface (Chapter 3), which provides instructions and examples on how to add new elements (as ES processors) to the Testbed; the Corotational Interface (Chapter 4), which describes the theory, implementation and usage of the built-in corotational approach to geometric nonlinearity; the Constitutive Interface (Chapter 5), which describes the various user and developer options for performing material constitutive calculations; the Database Interface (Chapter 6), which describes most of the global datasets employed by ES processors for both input and output; and the Architecture Interface (Chapter 7), which describes the internal software design of the ES processor shell, to enable its maintenance, evolution and customizing for advanced or special-purpose features.

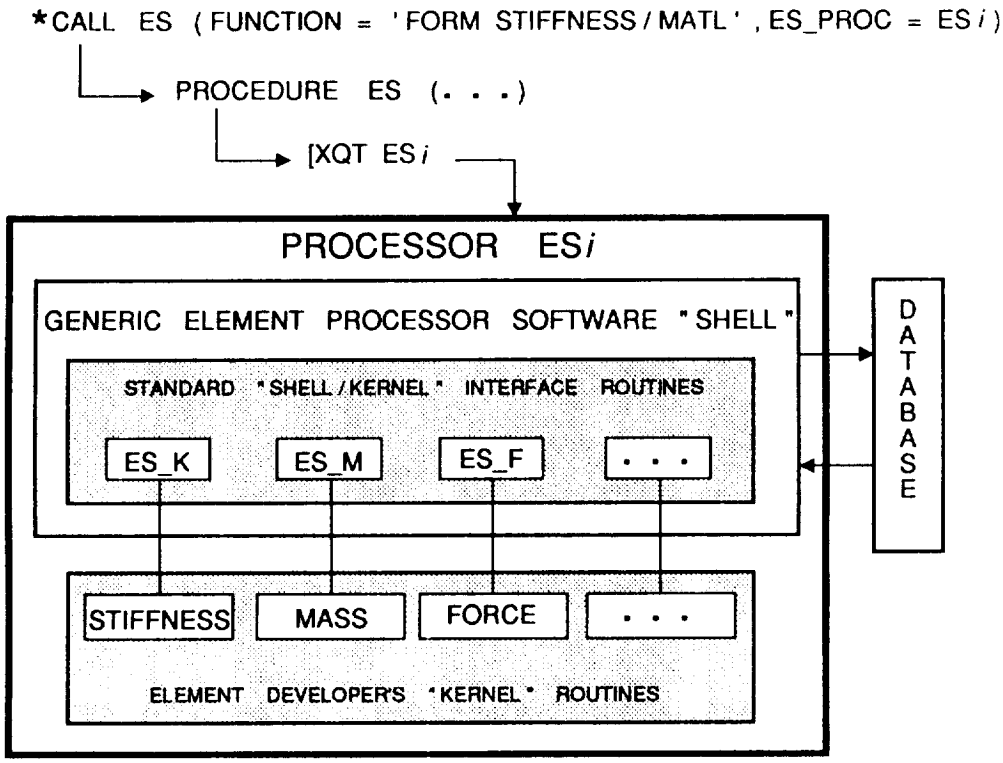


Figure 1.1 Generic Element Processor Design.

THIS PAGE LEFT BLANK INTENTIONALLY.

2. USER INTERFACE

CHAPTER OUTLINE

Section	*Title*	*Description*
2.1	**Overview**	Introduces the user to the Generic Element processor concept and the structural element (ES) processor template in particular. Briefly summarizes how ES processors are used in the Testbed to perform structural analysis.
2.2	**ES Processor Commands**	Summarizes and describes the usage of standard commands used by all ES processors. Includes a list of relevant macrosymbols and input/output datasets for each command. Standard element definition conventions (*e.g.*, node numbers) are also described here.
2.3	**Procedure Interface to ES Processors**	Describes a high-level procedure to access ES processors. Enables the user to write analysis procedures that automatically invoke all ES processors associated with a given problem, with a single procedure call.
2.4	**Glossary of ES Processor Macrosymbols**	Detailed definitions of all standard macrosymbols used by ES processors — and, equivalently, arguments used by the generic ES procedure.
2.5	**ES Processor/Procedure Usage Examples**	Examples for using ES processors, and/or the generic ES procedure, for pre-processing and post-processing, linear analysis, and nonlinear analysis.

2.1 Overview

The Generic Structural-Element (or ES) Processor provides a *template* with which many individual finite-element processors may be developed and coexist as independent modules in the CSM Testbed. The generic processor template for *structural* elements is referred to as ES, and all element processors built with this template have names that begin with ES (*e.g.*, ES1, ES2, ...). Each of these ES processors performs *all* operations for all of the elements implemented within the processor — including element definition, stiffness, force, mass (etc.) generation, and various pre-processing and post-processing functions.*

Since ES processors are typically built by different developers, a wide variety of user-interface characteristics may be expected. However, because of the generic template employed, all ES processors share the same command language and global datasets. This commonality means that a user has to learn only one convention to invoke any ES processor.

The main difference between ES processors will be in the specific elements implemented within them. Some ES processors may have only one element type inside. Others may have a family of elements of a certain class (*e.g.*, 4-, 9-, and 16-node shell elements). Still others may embed an entire library of structural elements, containing various members of each class (*e.g.*, beam, shell, solid). Each of these specific element types is given a corresponding name within each ES processor, so that the combination of ES processor name and element type is unique. Thus, to employ a particular ES processor correctly, the user will have to consult specific documentation on the individual elements contained within that processor. Such documentation is provided in the CSM Testbed User's Manual (ref. 4).

In the following sections, the features of ES processors are described in detail. The description includes ES processor commands, macrosymbols, and the datasets required or produced by these commands (Section 2.2); the high-level *procedure* interface, which makes it possible to write analysis procedures that invoke a single, generic procedure (called ES)

* Exception: Element connectivity for all element processors is currently performed by Testbed processor ELD, using the <ES_EXPE_CMD>.

to generate element arrays automatically for all ES processors required in a given problem (Section 2.3); a glossary of ES macrosymbols (Section 2.4), which gives more detailed definitions for the macrosymbols and procedure arguments referred to throughout the chapter; and some explicit examples of how to use ES processors directly or indirectly using the ES procedure interface (Section 2.5).

2.2 ES Processor Commands

All structural element (ES) processors based on the generic element processor template share the same processor commands. This feature makes it easier for the user and enables the construction of a generic command procedure to handle any combination of ES processors (see Section 2.3). Most of the commands recognized by ES processors fall into three categories: DEFINE, FORM, and POST, which roughly correspond to pre-processing, computation, and post-processing phases of analysis. The DEFINE commands are used to prepare or reformat model definition datasets, such as element connectivity (currently defined using processor ELD), freedom activity, loads, etc. The FORM commands are used to form element computational data, such as stiffness matrices, force vectors, etc. The POST commands are used to post-process element-oriented results, for example, interpolation to obtain internal node displacements, or extrapolation of stresses from element integration points to nodes. Additionally, there is an INITIALIZE command which must be invoked once, prior to invoking any of the FORM commands. A summary of ES processor commands and their respective functions is provided in Table 2.1.

In addition to these commands, ES processors can be controlled with a number of built-in *macrosymbols*, all of which begin with ES_. The macrosymbols typically set logical switches and/or control parameters, and allow reassignment of database names from their default values.

The remainder of this section describes each of these commands in detail, including syntax and a summary of relevant input/output macrosymbols and input/output datasets. For detailed definitions of the macrosymbols and datasets, the reader is referred to Section 2.4 (Glossary of Macrosymbols) and Chapter 6 (Database Interface), respectively.

Table 2.1 Summary of Generic ES Processor Commands

Command		*Description*
INITIALIZE		Initialize element data for FORM
DEFINE	ELEMENTS	Define element parameters
	FREEDOMS	Perform automatic DOF suppression
	LOADS	Define element (distributed) loads
FORM	STIFFNESS [/MATL/GEOM/LOAD/TANG]	Form element stiffness
	FORCE [/INT/EXT/RES/DYN]	Form element force
	MASS [/CONS/DIAG]	Form element mass
	STRAIN	Form element strains
	STRESS	Form element stresses
POST	STRAIN	Postprocess strains
	STRESS	Postprocess stresses
	DISPLACEMENTS	Postprocess displacements

2.2.1 DEFINE Commands

The DEFINE family of ES processor commands is used for various pre-processing functions. Included here are the following special cases:

- DEFINE ELEMENTS
- DEFINE FREEDOMS
- DEFINE LOADS

The DEFINE ELEMENTS command is used in conjunction with processor ELD to set up database tables and macrosymbols summarizing the key attributes of all element (ES) processors and specific element types to be employed in a given model. The DEFINE FREEDOMS command is used to facilitate automatic element degree of freedom suppression using the generic ES procedure (see Section 2.3). The DEFINE LOADS command is used to store distributed element load descriptions in the database, which are later used to generate consistent nodal loads. Point forces, applied motions, and boundary conditions are defined at the global level using processors TAB and AUS.

2.2.1.1 The DEFINE ELEMENTS Command

The DEFINE ELEMENTS command is used during pre-processing to indicate that a specific element type, within a given ES processor, is going to be included in the model. This command must currently be used *in conjunction with, and generally prior to, processor ELD* — which defines element nodal and property-table connectivity, and sets up various element datasets.

The main function of the DEFINE ELEMENTS command is to prepare a dataset called ES.SUMMARY (see Chapter 6), which contains a list of all active ES processors and associated element types for a given problem. This command enables the generic ES *procedure* (Section 2.3) to process all pertinent ES processors automatically during the solution phase. A secondary function of this command is to define various intrinsic element parameters, such as the number of element nodes, integration points, and store them in the ES.SUMMARY dataset and also in global macrosymbols that are accessible to the user at the procedure level. The database records are useful as a summary of element attributes and implementation status, which the user may check at any stage of the analysis.

The standard node-numbering sequences for a variety of standard 1-D, 2-D and 3-D element topologies is shown in Figure 2.1. These conventions should be used when defining element connectivity using Testbed processor ELD. Note that while ES processor *developers* may have their own internal conventions for numbering element nodes, ES processor users should always use the standard convention. Developers are asked to define a node re-sequencing array (NODES) to establish the relationship between external and internal node numbers, so that all elements with common topologies can be defined in a consistent manner by the user or an automated pre-processor.

2.2.1.1.1 Syntax

DEFINE ELEMENTS

2.2.1.1.2 Input Macrosymbols

ES_NAME	Element-type name
ES_PARS	Array of element research parameters
ES_SUM_DS	Name of element summary dataset. (Default: ES.SUMMARY)

2.2.1.1.3 Output Macrosymbols

ES_C	Degree of inter-element continuity (*e.g.*, 0=C^0, 1=C^1)
ES_CNS	Element constitutive processing type
ES_DIM	Element intrinsic dimensionality (1=beam, 2=plate or shell, 3=solid)
ES_ES*i*	Status indicators for various element functions (see Section 2.4)
ES_NDOF	Number of element freedoms per node
ES_NEN	Number of element nodes
ES_NIP	Number of element integration points
ES_NEE	Number of element equations (*e.g.*, product of <ES_NEN> *and* <ES_NDOF>)
ES_NORO	Normal rotation (drilling) freedom tolerance
ES_NSTR	Number of element stress components per integration point
ES_OPT	Element option number
ES_SHAP	Element shape (idTRIA or idQUAD)
ES_STOR	Number of entries stored in Segment 6 of dataset <ES_NAME>.EFIL.* (see Chapter 6).

2.2.1.1.4 Input Datasets

None.

2.2.1.1.5 Output Datasets

<ES_SUM_DS>	Structural element summary dataset (see Chapter 6). Contains nominal record groups for each element parameter listed under "Output Macrosymbols".

2.2.1.2 The DEFINE FREEDOMS Command

The DEFINE FREEDOMS command can be used to generate a table of potentially active degrees-of-freedom (dof) for elements that have been previously defined by an ES processor. This table is output to the database as a single dataset, ES.DOFS, which may be "accumulated" for an entire model by invoking the DEFINE FREEDOMS command in a series of ES processor runs — one for each processor/element-type combination associated with the model. The resulting dataset will contain a table of potentially active degrees of freedom reflecting all elements in the model that may be merged with the actual constraint datasets (*e.g.*, CON..*icon*). This merging process is performed using the MERGE_DOF command of processor VEC to achieve automatic suppression of degrees of freedoms superfluous to the element (*e.g.*, eliminate the drilling freedoms). Alternatively, a more convenient way of performing this whole sequence of operations is by calling the generic ES *procedure* with the argument COMMAND = 'DEFINE FREEDOMS'.

2.2.1.2.1 Syntax

DEFINE FREEDOMS

2.2.1.2.2 Input Macrosymbols

ES_DOF_DS	Name of element freedom dataset in which table of potentially active nodal freedoms will be stored and merged with other elements. (Default: ES.DOFS)
ES_NAME	Element-type name
ES_PARS	Array of element research parameters
ES_TGC_DS	Name of nodal transformations dataset. (Default: QJJT.BTAB.2.19)
ES_XYZ_DS	Name of nodal coordinates dataset. (Default: JLOC.BTAB.2.5)

2.2.1.2.3 Output Macrosymbols

None.

2.2.1.2.4 Input Datasets

DEF.<ES_NAME>	Element definition (connectivity) dataset.
DIR.<ES_NAME>	Element directory dataset.
<ES_TGC_DS>	Nodal transformations dataset. (Default: QJJT.BTAB.2.19)
<ES_XYZ_DS>	Nodal coordinates dataset. (Default: JLOC.BTAB.2.5)

2.2.1.2.5 Output Datasets

<ES_DOF_DS>	Table of potentially active freedoms for current set of elements, or for *accumulated* set if other ES processors/elements have been employed for this purpose earlier. (Default: ES.DOFS)

2.2.1.3 The DEFINE LOADS Command

The DEFINE LOADS command can be used to define *element loads* and store them in the database for subsequent recovery during the analysis. By element loads, we refer to distributed forces (*e.g.*, line loads, pressures, body forces) which require element processing to convert them into *consistent nodal forces*. The purpose of the DEFINE LOADS command is simply to store the primitive element load distributions in the database. Consistent nodal forces can then be computed subsequently using the FORM FORCE/EXT command, discussed in Section 2.2.3.2.

2.2.1.3.1 Syntax

```
DEFINE LOADS /Type   [/LIVE]   [/SYSTEM = System]
        [GROUP = grp1, grp2, grpinc]
                [ELEMENT = elt1, elt2, eltinc]
                        [Boundary = bnd1, bnd2, bndinc]
                                [NODE = nod1, nod2, nodinc]
                                        LOAD = load_values
END DEFINE
```

2.2.1.3.2 Load "Type" Qualifiers

LINE — Line loads are defined as forces (and/or moments) per unit length. They may be applied to 1-D (*e.g.*, beam) elements or along edges of 2-D and 3-D elements.

PRESSURE — Pressure loads are defined as forces per unit area that are directed normal to an element's surface. Positive pressure values are assumed to point along the "outward" normal to the element surface (see Figure 2.1b)

SURFACE — Surface loads are defined as general traction vectors, (*i.e.*, force or moment per unit element surface area). They may be applied to 2-D elements, and to selected surfaces of 3-D elements.

BODY — Body loads are defined as forces per unit mass, and may be applied to 1-D, 2-D and 3-D elements. A typical example of a body load is gravity, where the gravitational constant, g, is the magnitude, the direction is fixed (*e.g.*, towards earth), and both are constant for all nodes and elements in the structure.

2.2.1.3.3 Load "System" Qualifiers

GLOBAL — Indicates that the components of the load vector, specified using the LOAD phrase, are expressed in the global-Cartesian coordinate system.

NODAL — Indicates that the components of the load vector, specified using the LOAD phrase, are expressed in the nodal-Cartesian (*i.e.*, computational) coordinate system at each node. This system is the same as that specified by processor TAB's ALTREF command, and stored in the QJJT.BTAB dataset.

ELEMENT — Indicates that the components of the load vector, specified using the LOAD phrase, are expressed in the element-Cartesian coordinate system. This system is the same as the element corotational (or E) frame, shown in Figure 2.1a for various element types.

2.2.1.3.4 LIVE Load Qualifier

The LIVE load qualifier is used to designate element loads that are to be displacement dependent. Currently, the only type of live load implemented is the live pressure load (DEFINE LOAD /PRESSURE /LIVE), which denotes a pressure load that remains normal to the element surface during deformation. A common example of this type of loading is *hydrostatic* pressure loading of a submerged shell structure.

2.2.1.3.5 Load GROUP Specification

GROUP = grp1, grp2, grpinc

The GROUP specification is an optional phrase used to set a range or subset of element groups to be loaded within the current element type. The range specification is such that "grp1" is the first group in the range, "grp2" is the last group in the range, and "grpinc" is the increment used to count from "grp1" to "grp2". The value of "grpinc" defaults to 1; the value of "grp2" defaults to "grp1". *The default specification is from 1 to the total number of element groups.*

2.2.1.3.6 Load ELEMENT Specification

ELEMENT = elt1, elt2, eltinc

The ELEMENT specification is an optional phrase used to set a range or subset of elements *within each element group* to be loaded. The range specification is such that "elt1" is the first element in the range, "elt2" is the last element in the range, and "eltinc" is the increment used to count from "elt1" to "elt2". The value of "eltinc" defaults to 1; the value of "elt2" defaults to "elt1", and will automatically be reset to the total number of elements in each specified element group if "elt2" is too large. *The default specification is from 1 to the total number of elements within each element group specified by the GROUP phrase.*

2.2.1.3.7 Load "Boundary" Specification

Boundary = bnd1, bnd2, bndinc

The "Boundary" specification is an optional phrase used to set a range of element boundaries to be loaded — within the given range of GROUPs and ELEMENTs. The "Boundary" name depends on the load "Type". The parameter, "bnd1", is the first boundary in the range, "bnd2" is the last boundary in the range, and "bndinc" is the increment used to count from "bnd1" to "bnd2". Legitimate "Boundary" names are:

LINE	Specifies range of element lines to be loaded (2-D/3-D elements only)
SURFACE	Specifies range of element surfaces to be loaded (3-D elements only)

The default specification is from 1 to the total number of boundaries, of type "Boundary" within the specified range of elements. Note that the "Boundary" specification is irrelevant for BODY forces, for LINE loads on 1-D elements, or for SURFACE loads on 2-D elements. (see Figure 2.1b.)

2.2.1.3.8 Load NODE Specification

NODE = nod1, nod2, nodinc

The NODE specification is an optional phrase used to set a range of element boundary nodes to be loaded — within the given range of GROUPs, ELEMENTs and "Boundary"s. The integer "nod1" is the first boundary node in the range, "nod2" is the last boundary node in the range, and "nodinc" is the increment used to count from "nod1" to "nod2". The value of "nodinc" defaults to 1. The value of "nod2" defaults to "nod1" and is automatically reset to the maximum of "nod2" and the total number of boundary nodes per element boundary. *The default specification is from 1 to the total number of boundary nodes for each of the boundaries specified by the "Boundary" phrase, and within the specified range of elements.* (see Figure 2.1b.)

2.2.1.3.9 LOAD Specification

LOAD = load_values

The LOAD specification is used to define the values of the distributed load "vector" for the range of nodes, boundaries, elements and groups specified by the NODE, Boundary, ELEMENT and GROUP phrases, respectively. For all load types except PRESSURE, the number of components in "load_values" is equal to the number of degrees of freedom per element node. For PRESSURE loads, "load_values" is just the one scalar value — considered positive in the direction of the outward normal to the element's loaded surface (see individual element processor conventions in Chapter 5 of the Testbed User's Manual, ref. 4).

2.2.1.3.10 Input Macrosymbols

Name	Description
ES_NAME	Element-type name
ES_LOAD_SET	Load set number

2.2.1.3.11 Output Macrosymbols

None.

2.2.1.3.12 Input Datasets

Name	Contents
GD.<ES_NAME>.*	Element group definition dataset.

2.2.1.3.13 Output Datasets

Name	Contents
LOADS.<ES_NAME>.<ES_LOAD_SET>	Element loads dataset for load set <ES_LOAD_SET>. The record groups in this dataset depend on the types of element loads defined. They may include LINE_LDS, PRES_LDS, SURF_LDS and BODY_LDS. Live-load record groups appear with the prefix LIV; for example, LIV_PRES_LDS.

2.2.1.3.14 Examples

Example 1. Constant Pressure Loads on 2-D Elements

```
*def ES_NAME = EX47
 DEFINE LOADS /PRESSURE
        LOAD = 1.0
 END DEFINE
```

In the above example a pressure magnitude of 1.0 is applied to all nodes of all 2-D elements of type EX47.

Example 2. Constant Line Loads on 1-D Elements

```
*def ES_NAME = E210
 DEFINE LOADS /LINE
        LOAD = 1.0, 1.0, 1.0
 END DEFINE
```

In the above example, a line load, with all three global components equal to 1.0, is applied to all nodes of all 1-D elements of type E210.

Example 3. Constant Line Loads on Selected 2-D Elements

```
*def ES_NAME = EX97
 DEFINE LOADS /LINE /SYSTEM=NODAL
        ELEMENT = 10, 100, 10
              LINE = 2
                    LOAD = 1.0
 END DEFINE
```

In the above example, a line load, of magnitude 1.0 and in the direction of the third nodal (computational) basis vector, is applied to all nodes along edge 2 of EX97 (9-noded shell) elements 10, 20, 30, 40, 50, 60, 70, 80, 90, 10.

Example 4. Constant Pressure Loads on Selected 3-D Elements

```
*def ES_NAME = SD20
 DEFINE LOADS /PRESSURE
      ELEMENT = 10, 1000, 10
          SURFACE = 2
              LOAD = 1.0
 END DEFINE
```

In the above example, a pressure load, of magnitude 1.0 is applied to all nodes on surface 2 of SD20 (20-noded solid) elements 10, 20, 30, ..., 1000.

Example 5. Piecewise-Constant "Live" Pressure Loads on 2-D Elements

```
*def ES_NAME = E410
 DEFINE LOADS /PRESSURE /LIVE
      GROUP = 1
          LOAD = 1.0
      GROUP = 2
          LOAD = 2.0
 END DEFINE
```

In the above example, a live pressure load of magnitude 1.0 is applied to all elements in GROUP 1 of element type E410 (4-noded shell), and twice that amount is applied to all elements in GROUP 2 of element type E410.

2.2.1.3.15 Element Load Boundary/Node Ordering Conventions

1-D Elements:

Only line and body loads can be applied to 1-D elements. In either case, the node ordering corresponds to that used for specification of element connectivity (see Figure 2.1a).

2-D Elements:

For surface and body loading of 2-D elements, the node ordering corresponds to that used for specification of element connectivity (see Figure 2.1a). For line loading of 2-D elements, the element lines (edges) are ordered as shown in Figure 2.1b, and the node order within each line is the same as the element connectivity for 1-D elements (vertex nodes first as in Figure 2.1a).

3-D Elements:

For body loading of 3-D elements, the node ordering corresponds to that used for specification of element connectivity (see Figure 2.1a). For line loading of 3-D elements, the element lines (edges) are ordered as shown in Figure 2.1b, and the node order within each line is the same as the element connectivity for 1-D elements (Figure 2.1a). For surface loading of 3-D elements, the element surfaces are ordered as shown in Figure 2.1b, and the node order within each surface is the same as the element connectivity for 2-D elements (Figure 2.1a).

2.2.2 The INITIALIZE Command

The INITIALIZE command must be used *once* to initialize computational data and to precompute various element (and constitutive) quantities that remain constant throughout the analysis. Included in the latter category are element-dependent data whose length are given by the macrosymbol, ES_STOR, generated by the DEFINE ELEMENTS command. All initialization data are presently stored in the database, within the <ES_NAME>.EFIL dataset for a specific element type (see Chapter 6). *Note that the INITIALIZE command is a prerequisite for using any of the FORM commands.*

2.2.2.1 Syntax

INITIALIZE

2.2.2.2 Input Macrosymbols

ES_NAME	Element-type name (used to designate input/output datasets)
ES_PARS	Array of element research parameters (optional)
ES_CORO	Element corotational switch. (Default: <true>; irrelevant for linear analysis)
ES_NL_GEOM	Element geometric nonlinearity switch. (Default: <false>)
ES_NL_LOAD	Element load nonlinearity switch. (Default: <false>)
ES_NL_MATL	Element material nonlinearity switch. (Default: <false>)
ES_TGC_DS	Name of computational-to-global nodal transformation dataset. (Default: QJJT.BTAB.2.19)
ES_PRP_DS	Name of element section property dataset. (Default: PROP.BTAB.*)

2.2.2.3 Output Macrosymbols

None.

2.2.2.4 Input Datasets

DIR.<ES_NAME>.*	Directory dataset for elements of type <ES_NAME>
DEF.<ES_NAME>.*	Definition dataset for elements of type <ES_NAME>
<ES_NAME>.EFIL.*	Computational dataset for elements of type <ES_NAME>
<ES_PRP_DS>	Section property dataset. (Default: PROPS.BTAB.2.$n2$, where $n2$ is defined in processor ELD's EXPE command)
<ES_TGC_DS>	Computational-to-global nodal transformation dataset. (Default: QJJT.BTAB.2.19)

2.2.2.5 Output Datasets

<ES_NAME>.EFIL.*	Computational dataset for elements of type <ES_NAME>. Will include initialized element storage array in Segment 6 of each element record (see Chapter 6).

2.2.3 FORM Commands

The FORM family of ES processor commands is used to form all of the essential element arrays required during the computational phases of analysis. Included here are the following special cases:

- FORM STIFFNESS
- FORM FORCE
- FORM MASS
- FORM STRAIN
- FORM STRESS

The FORM STIFFNESS command forms element stiffness matrices (material, geometric, load and/or tangent), and saves them in the database in the so-called EFIL dataset, for subsequent assembly. The FORM FORCE command forms element force vectors (internal, external, residual and/or dynamic), and *assembles* them directly into a preexisting system force-vector dataset. The FORM MASS command forms element mass matrices (consistent or lumped), and either stores them in the EFIL dataset, if they are consistent (full) matrices, or assembles them directly into a system mass ("vector") dataset, if they are lumped (diagonal). The FORM STRAIN (or STRESS) command computes element strains (or stresses) — continuum or resultant, depending on the element class — and deposits them in separate STRN (or STRS) datasets. The FORM STRAIN and FORM STRESS commands are typically used for post-processing; they are not required during the solution phase, as stresses and strains are automatically computed by the FORM STIFFNESS and/or FORM FORCE commands, as needed.

2.2.3.1 The FORM STIFFNESS Command

The FORM STIFFNESS command is used to form element stiffness matrices for all elements associated with the current ES processor and element-type name (macrosymbol ES_NAME).

2.2.3.1.1 Syntax

FORM STIFFNESS [/Qualifier]

2.2.3.1.2 Qualifiers

Valid qualifiers are given below with the default value of the qualifier underlined.

MATERIAL — Indicates element *material* stiffness matrices are to be formed. The material stiffness matrix is defined as that part of the tangent (total) stiffness matrix which depends explicitly on material properties, obtained by linearization of the material constitutive relation. For linear analysis, it is equivalent to the tangent stiffness.

GEOMETRIC — Indicates element *geometric* stiffness matrices are to be formed. The geometric stiffness matrix is defined as that part of the tangent (total) stiffness matrix which depends explicitly on stresses. It is obtained by linearization of the strain-displacement relations, and is often called the "initial stress" stiffness matrix. It is needed for both buckling eigenvalue analysis and for nonlinear analysis.

LOAD — Indicates element *load* stiffness matrices are to be formed. The load stiffness matrix is defined as that part of the tangent (total) stiffness matrix emanating from displacement-dependent loads (*i.e.*, external forces that explicitly depend on displacements). It is typically absent except in cases involving loading by fluid pressure or other follower forces, and in those cases may be needed for either eigenvalue or nonlinear analysis.

TANGENT — Indicates element *tangent* stiffness matrices are to be formed. The tangent (total) stiffness matrix is defined as the derivative of the element residual force vector with respect to the element displacement vector. By definition, it includes all pertinent effects (material, geometric, and load stiffnesses) and should be used in conjunction with any form of nonlinear analysis.

2.2.3.1.3 Input Macrosymbols

ES_NAME — Element-type name (used to designate input/output datasets)

ES_PARS — Array of element research parameters (optional)

ES_CORO — Element corotational switch. (Default: <true>; irrelevant for linear analysis)

ES_PROJ — Element rigid-body projection switch. (Default: <false>)

ES_NL_GEOM — Element geometric nonlinearity switch. (Default: <false>)

ES_NL_LOAD — Element load nonlinearity switch. If <true> and the FORM STIFFNESS command qualifier is either /TANGENT, or /GEOMETRIC, then the element *load stiffness* matrix will be added to the tangent, or geometric, stiffness matrix, respectively. (If the command qualifier is /LOAD, the load stiffness matrix will be formed independent of the value of <ES_NL_LOAD>.) In either case, distributed *live* load values will be input from the element loads dataset, LOADS.<ES_NAME>.<ES_LOAD_SET> (*i.e.*, records with the prefix LIV), scaled by <ES_LOAD_FACTOR>, and used to form the element load stiffness matrices. Presently, only load stiffness matrices corresponding to live *pressure* loads are implemented, and only a small subset of element processors have this capability (see Chapter 5 of the Testbed User's Manuals, ref. 4, to see which elements have load-stiffness matrices). (Default: <false>)

ES_NL_MATL — Element material nonlinearity switch. (Default: <false>)

ES_DIS_DS Name of system displacement vector dataset. (Default: STAT.DISP.1.1)

ES_ROT_DS Name of system rotation pseudo-vector dataset. (Default: STAT.ROTA.1.1)

ES_TGC_DS Name of computational-to-global nodal transformation dataset. (Default: QJJT.BTAB.2.19).

ES_PRESTRESS Prestress flag. (Default: <false>)

ES_S*i*[1:*ng*] Constant prestress values for component *i* of element groups 1–*ng*, where *ng* is the total number of element groups within element type <ES_NAME>. (Note: The current prestress mechanism is preliminary. A more general implementation scheme which employs the *database* is in preparation.)

ES_PRP_DS Name of element section property dataset. (Default: PROP.BTAB.*)

ES_LOAD_SET Load set number. Indicates dataset in which external distributed loads are to be found. The external loads dataset — if it exists — is called LOADS.<ES_NAME>.<ES_LOAD_SET>, and is created using the DEFINE LOADS command. (Default: 1)

ES_LOAD_FACTOR

Load factor to be applied to all external distributed loads beforce converting to consistent nodal forces. (Default: 1.0)

2.2.3.1.4 Output Macrosymbols

None.

2.2.3.1.5 Input Datasets

DIR.<ES_NAME>.*	Directory dataset for elements of type <ES_NAME>
DEF.<ES_NAME>.*	Definition dataset for elements of type <ES_NAME>
<ES_NAME>.EFIL.*	Computational dataset for elements of type <ES_NAME>
<ES_PRP_DS>	Section property dataset. (Default: PROPS.BTAB.2.$n2$, where $n2$ is defined in processor ELD's EXPE command)
<ES_DIS_DS>	System displacement vector dataset. (Default: STAT.DISP.1.1)
<ES_ROT_DS>	System rotation pseudo-vector dataset. (Default: STAT.ROTA.1.1)
<ES_TGC_DS>	Computational-to-global nodal transformation dataset. (Default: QJJT.BTAB.2.19)
LOADS.<ES_NAME>.<ES_LOAD>	Element loads dataset created using the DEFINE LOADS command. Relevant only for /LOAD stiffness, or for /TANGENT and /GEOMETRIC stiffness if <ES_NL_LOAD> is true.

2.2.3.1.6 Output Datasets

<ES_NAME>.EFIL.*	Computational dataset for elements of type <ES_NAME>. Will include updated element stiffness matrices and, in nonlinear analysis, updated corotational geometric data.

2.2.3.2 The FORM FORCE Command

The FORM FORCE command is used to form element force vectors (internal and/or external) for all elements of a given type (as specified by macrosymbol ES_NAME) within a given ES processor. Currently, element force vectors are immediately assembled into a system force vector; *i.e.*, there is no allocation for element force vectors in the database.

2.2.3.2.1 Syntax

FORM FORCE [/Qualifier]

2.2.3.2.2 Qualifiers

Valid qualifiers are given below with the default value of the qualifier underlined.

INTERNAL — Indicates element *internal* force vectors are to be formed and assembled. The internal force vector is defined as the set of element nodal forces which depends explicitly on the element internal stress distribution (and or initial strains/temperatures). In the case of a conservative system, this vector emanates from the first variation of the element strain energy.

EXTERNAL — Indicates element *external* force vectors are to be formed and assembled. The external force vector is defined as the set of consistent element nodal forces corresponding to the distributed loads specified using the DEFINE LOADS command.

RESIDUAL — Indicates element *residual* force vectors are to be formed and assembled. The residual force vector is defined as the difference between the external force vector and the internal force vector (*i.e.*, $\mathbf{f}^{res} = \mathbf{f}^{ext} - \mathbf{f}^{int}$).

DYNAMIC — Indicates element *dynamic* (or inertial) force vectors are to be formed and assembled. These forces are due to the product of the element mass matrix and the element acceleration vector (in linear analysis) and correspond to the inertial forces in d'Alembert's principle.

2.2.3.2.3 Input Macrosymbols

ES_NAME	Element-type name (used to designate input/output datasets)
ES_PARS	Array of element research parameters (optional)
ES_CORO	Element corotational switch. (Default: <true>; irrelevant for linear analysis)
ES_PROJ	Element rigid-body projection switch. (Default: <false>)
ES_NL_GEOM	Element geometric nonlinearity switch. (Default: <false>)
ES_NL_LOAD	Element load nonlinearity switch. If <true>, only live (displacement-dependent) loads will be processed from the element loads dataset, LOADS.<ES_NAME>.<ES_LOAD_SET>. This means that only records in this dataset beginning with the prefix LIV will be input (*e.g.*, LIV_PRES_LDS). Presently, only live pressure loads are implemented (see DEFINE LOADS command). If <false>, only *non-live* loads (*i.e.*, records in the element load dataset that do not have the prefix LIV) will be processed. (Default: <false>)
ES_NL_MATL	Element material nonlinearity switch. (Default: <false>)
ES_DIS_DS	Name of system displacement vector dataset. (Default: STAT.DISP.1.1)
ES_FRC_DS	Name of system (assembled) force dataset. (Default: INT.FORC.1.1)
ES_ROT_DS	Name of system rotation pseudo-vector dataset. (Default: STAT.ROTA.1.1)
ES_TGC_DS	Name of computational-to-global nodal transformation dataset. (Default: QJJT.BTAB.2.19)
ES_PRESTRESS	Prestress flag. (Default: <false>)

ES_Si[1:*ng*] Constant prestress values for component *i* for element groups 1–*ng*, where *ng* is the total number of element groups within element type <ES_NAME>. (Note: The current prestress mechanism is preliminary. A more general implementation scheme which employs the *database* is in preparation.)

ES_LOAD_SET Load set number. Indicates dataset in which external distributed loads are to be found. The external loads dataset — if it exists — is called LOADS.<ES_NAME>.<ES_LOAD_SET>, and is created using the DEFINE LOADS command. (Default: 1)

ES_LOAD_FACTOR

Load factor to be applied to all external distributed loads beforce converting to consistent nodal forces. (Default: 1.0)

2.2.3.2.4 Output Macrosymbols

None.

2.2.3.2.5 Input Datasets

DIR.<ES_NAME>.*	Directory dataset for elements of type <ES_NAME>
DEF.<ES_NAME>.*	Definition dataset for elements of type <ES_NAME>
<ES_NAME>.EFIL.*	Computational dataset for elements of type <ES_NAME>
<ES_PRP_DS>	Section property dataset. (Default: PROPS.BTAB.2.$n2$, where $n2$ is defined in processor ELD's EXPE command)
<ES_DIS_DS>	System displacement vector dataset. (Default: STAT.DISP.1.1)
<ES ROT_DS>	System rotation pseudo-vector dataset. (Default: STAT.ROTA.1.1)
<ES_TGC_DS>	Computational-to-global nodal transformation dataset. (Default: QJJT.BTAB.2.19)

LOADS.<ES_NAME>.<ES_LOAD_>

Element loads dataset created using the DEFINE LOADS command. Not relevant for /INTERNAL loads.

2.2.3.2.6 Output Datasets

<ES NAME>.EFIL.*	Computational dataset for elements of type <ES_NAME> Will include updated element corotational data if geometrically nonlinear analysis.
<ES_FRC_DS>	System force dataset: contains assembled element internal, external, residual, or dynamic forces, depending upon the command qualifier used.

2.2.3.3 The FORM MASS Command

The FORM MASS command is used to form element consistent mass matrices, or to form and assemble element diagonal (lumped) mass matrices, for all elements associated with the current ES processor and element-type as specified by macrosymbol ES_NAME.

2.2.3.3.1 Syntax

FORM MASS [/Qualifier]

2.2.3.3.2 Qualifiers

Valid qualifiers are given below with the default value of the qualifier underlined.

CONSISTENT Indicates element *consistent* mass matrices are to be formed. These mass matrices are output to the <ES_NAME>.EFIL dataset, and may be assembled into a system mass matrix using processor K.

DIAGONAL Indicates element *diagonal* (*i.e.*, lumped) mass matrices are to be formed and assembled into a system diagonal mass "vector".

2.2.3.3.3 Input Macrosymbols

ES_NAME Element-type name (used to designate input/output datasets)

ES_PARS Array of element research parameters (optional)

ES_CORO Element corotational switch. (Default: <true>; irrelevant for linear analysis)

ES_NL_GEOM Element geometric nonlinearity switch. (Default: <false>)

ES_MASS_DIAG First word of dataset name for diagonal mass matrix. (Default: DEM)

ES_MASS_DS First two words of dataset name for diagonal mass matrix. (Default: <ES_MASS_DIAG>.DIAG)

ES_DIS_DS Name of system displacement vector dataset. (Default: STAT.DISP)

ES_ROT_DS	Name of system rotation pseudo-vector dataset. (Default: STAT.ROTA.1.1)
ES_TGC_DS	Name of computational-to-global nodal transformation dataset. (Default: QJJT.BTAB.2.19)
ES_PRP_DS	Name of element section property dataset. (Default: PROP.BTAB.*)

2.2.3.3.4 Output Macrosymbols

None.

2.2.3.3.5 Input Datasets

DIR.<ES_NAME>.*	Directory dataset for elements of type <ES_NAME>
DEF.<ES_NAME>.*	Definition dataset for elements of type <ES_NAME>
<ES_NAME>.EFIL.*	Computational dataset for elements of type <ES_NAME>
<ES_PRP_DS>	Section property dataset. (Default: PROPS.BTAB.2.$n2$, where $n2$ is defined in processor ELD's EXPE command)
<ES_MASS_DS>	System vector into which the element diagonal mass matrices will be assembled if the /DIAG command qualifier was used. This dataset is updated by the ES processor; *i.e.*, it is used as both input and output and must be initialized by the user *before* the FORM MASS/DIAG command is issued. (Default: <ES_MASS_DIAG>.DIAG)
<ES_DIS_DS>	System displacement vector dataset; relevant only for geometrically nonlinear analysis. (Default: STAT.DISP.1.1)
<ES_ROT_DS>	System rotation pseudo-vector dataset; relevant only for geometrically nonlinear analysis. (Default: STAT.ROTA.1.1)

<ES_TGC_DS>	Computational-to-global nodal transformation dataset. (Default: QJJT.BTAB.2.19)

2.2.3.3.6 Output Datasets

<ES_NAME>.EFIL.*	Computational dataset for elements of type <ES_NAME>. Will include element consistent mass matrix if the /CONS command qualifier was used; otherwise, this dataset is used only as input.
<ES_MASS_DS>	System vector into which the element diagonal mass matrices will be assembled if the /DIAG command qualifier was used; otherwise, this dataset is irrelevant. (Default: <ES_MASS_DIAG>.DIAG)

2.2.3.4 The FORM STRAIN Command

The FORM STRAIN command is used to compute strains for all elements of a given element type as specified by macrosymbol ES_NAME within a given ES processor. It is primarily a "post-processing" command; that is, strains do not have to be explicitly formed for analysis purposes. They are automatically formed as needed during the processing of other commands, such as FORM FORCE and FORM STIFFNESS.

Element strains may be computed at element integration points, at element centroids (by averaging if the centroid is not an integration point, or at element nodes by extrapolation from integration points. Moreover, they may be expressed in either the element stress coordinate system, or along directions corresponding to a selected permutation of the material coordinate axes. The selected strain quantities are output to the database in dataset STRN.<ES_NAME>.*, using a different record group for each of the strain options described above. Note that the element strains in this dataset correspond to *resultant* quantities for beam, plate and shell elements.

2.2.3.4.1 Syntax

FORM STRAIN

2.2.3.4.2 Input Macrosymbols

ES_NAME	Element-type name (used to designate input/output datasets)
ES_PARS	Array of element research parameters (optional)
ES_CORO	Element corotational switch. (Default: <true>; irrelevant for linear analysis)
ES_PROJ	Element rigid-body projection switch. (Default: <false>; irrelevant for nonlinear analysis)
ES_NL_GEOM	Element geometric nonlinearity switch. (Default: <false>)
ES_NL_LOAD	Element load nonlinearity switch. (Default: <false>)

ES_NL_MATL Element material nonlinearity switch. (Default: <false>)

ES_DIS_DS Name of system displacement vector dataset. (Default: STAT.DISP.1.1)

ES_ROT_DS Name of system rotation pseudo-vector dataset. (Default: STAT.ROTA.1.1)

ES_TGC_DS Name of computational-to-global nodal transformation dataset. (Default: QJJT.BTAB.2.19)

ES_PRP_DS Name of element section property dataset. (Default: PROP.BTAB.*)

ES_STRAIN_DS Last part (subscript numbers) of element strain dataset name. The entire dataset name is: STRN.<ES_NAME>.<ES_STRAIN_DS>. For linear problems, <ES_STRAIN_DS> corresponds to <load_set>.<constrain_set> and for nonlinear problems it corresponds to the load step number. (Default: STRN.<ES_NAME>.0.0)

ES_STR_DIR Element stress/strain directions; element, material or global; same as SREF parameter in processor ELD; see Section 2.4 Glossary. (Default: 0 ⇒ element stress basis)

ES_STR_LOC Element stress/strain evaluation points: CENTROIDS, INTEG_PTS or NODES. (Default: CENTROIDS)

2.2.3.4.3 Output Macrosymbols

None.

2.2.3.4.4 Input Datasets

DIR.<ES_NAME>.*	Directory dataset for elements of type <ES_NAME>
DEF.<ES_NAME>.*	Definition dataset for elements of type <ES_NAME>
<ES_NAME>.EFIL.*	Computational dataset for elements of type <ES_NAME>
<ES_PRP_DS>	Section property dataset. (Default: PROPS.BTAB.2.$n2$, where $n2$ is defined in processor ELD's EXPE command)
<ES_DIS_DS>	System displacement vector dataset. (Default: STAT.DISP.1.1)
<ES_ROT_DS>	System rotation pseudo-vector dataset. (Default: STAT.ROTA.1.1)
<ES_TGC_DS>	Computational-to-global nodal transformation dataset. (Default: QJJT.BTAB.2.19)

2.2.3.4.5 Output Datasets

STRN.<ES_NAME>.<ES_STRAIN_DS>

Element strain dataset. One of the following record groups is created, depending on input macrosymbols ES_STR_LOC and ES_STR_DIR:

CENTROIDS_S*dir*.1:*nel*
INTEG_PTS_S*dir*.1:*nel*
NODES_S*dir*.1:*nel*

where *dir* indicates the strain-output direction option, and is either 0 (element strain coordinate system) or the index of the material axis (1, 2, or 3) represents the x strain-output direction. The y and z output axes are determined by cyclic permutation over the remaining material axes.

2.2.3.5 The FORM STRESS Command

The FORM STRESS command is used to compute stresses for all elements of a given element type (specified by macrosymbol <ES_NAME>) within a given ES processor. It is primarily a "post-processing" command; that is, stresses do not have to be explicitly formed for analysis purposes. They are automatically formed as needed during the processing of other commands, such as FORM FORCE and FORM STIFFNESS.

Element stresses can be computed at element integration points, element centroids by averaging if the centroid is not an integration point, or element nodes by extrapolation from integration points. Moreover, they may be expressed in either the element stress coordinate system, or along directions corresponding to a selected permutation of the material coordinate axes. The selected stress quantities are output to the database in dataset STRS.<ES_NAME>.*, using a different record group for each of the stress options described above. Note that the element stresses in this dataset correspond to *resultant* quantities for beam, plate, and shell elements.

2.2.3.5.1 Syntax

FORM STRESS

2.2.3.5.2 Input Macrosymbols

ES_NAME	Element-type name (used to designate input/output datasets)
ES_PARS	Array of element research parameters (optional)
ES_CORO	Element corotational switch. (Default: <true>; irrelevant for linear analysis)
ES_PROJ	Element rigid-body projection switch. (Default: <false>; irrelevant for nonlinear analysis)
ES_NL_GEOM	Element geometric nonlinearity switch. (Default: <false>)
ES_NL_LOAD	Element load nonlinearity switch. (Default: <false>)

ES_NL_MATL	Element material nonlinearity switch. (Default: <false>)
ES_DIS_DS	Name of system displacement vector dataset. (Default: STAT.DISP.1.1)
ES_ROT_DS	Name of system rotation pseudo-vector dataset. (Default: STAT.ROTA.1.1)
ES_TGC_DS	Name of computational-to-global nodal transformation dataset. (Default: QJJT.BTAB.2.19)
ES_PRP_DS	Name of element section property dataset. (Default: PROP.BTAB.*)
ES_STRESS_DS	Last part (subscript numbers) of element stress dataset name. The entire dataset name is: STRS.<ES_NAME>.<ES_STRESS_DS>. For linear problems, <ES_STRESS_DS> corresponds to <load_set>.<constrain_set> and for nonlinear problems it corresponds to the load step number. (Default: STRS.<ES_NAME>.0.0)
ES_STR_DIR	Element stress/strain directions; element or material; same as SREF parameter in processor ELD; see Section 2.4 Glossary. (Default: 0 ⇒ element stress basis)
ES_STR_LOC	Element stress/strain evaluation points: CENTROIDS, INTEG_PTS or NODES. (Default: CENTROIDS)

2.2.3.5.3 Output Macrosymbols

None.

2.2.3.5.4 Input Datasets

DIR.<ES_NAME>.*	Directory dataset for elements of type <ES_NAME>
DEF.<ES_NAME>.*	Definition dataset for elements of type <ES_NAME>
<ES_NAME>.EFIL.*	Computational dataset for elements of type <ES_NAME>
<ES_PRP_DS>	Section property dataset. (Default: PROPS.BTAB.2.$n2$, where $n2$ is defined in rocessor ELD's EXPE command)
<ES_DIS_DS>	System displacement vector dataset. (Default: STAT.DISP.1.1)
<ES_ROT_DS>	System rotation pseudo-vector dataset. (Default: STAT.ROTA.1.1)
<ES_TGC_DS>	Computational-to-global nodal transformation dataset (Default: QJJT.BTAB.2.19)

2.2.3.5.5 Output Datasets

STRS.<ES_NAME>.<ES_STRESS_DS>

Element stress dataset. One of the following record groups is created, depending on input macrosymbols ES_STR_LOC and ES_STR_DIR:

CENTROIDS_S*dir*.1:*nel*
INTEG_PTS_S*dir*.1:*nel*
NODES_S*dir*.1:*nel*

where *dir* indicates the stress-output direction option, and is either 0 (element stress coordinate system) or the index of the material axis (1, 2, or 3) represents the x stress-output direction. The y and z output axes are determined by cyclic permutation over the remaining material axes.

2.2.4 POST Commands

The POST family of ES processor commands is used for various element post-processing functions. Included here are the following special cases:

- POST DISPLACEMENT
- POST STRAIN
- POST STRESS

The POST DISPLACEMENT command may be used to interpolate computed nodal displacements to selected interior points, such as element integration points or element geometric nodes, which are used by elements that employ a higher-order geometric description than that reflected by their *active* nodes (*i.e.*, super-parametric elements). This latter option fills in the empty slots in the system displacement vector corresponding to the geometric nodes, so that the displacement configuration can then be graphically post-processed.

The POST STRAIN and POST STRESS commands enable the user to "move" existing element strain or stress data from one set of locations to another within an element, *e.g.*, from integration points to nodes. They also enable the user to transform existing strain or stress data from one coordinate basis to another; for example, from element local to global cartesian axes. Thus, these two commands provide an expedient alternative to repeated use of the FORM STRAIN and FORM STRESS commands, allowing the user to interpolate, extrapolate or transform stress/strain data without having to recompute all data.

CURRENTLY NOT IMPLEMENTED

Table 2.2 CORRESPONDENCE BETWEEN ANALYSIS TYPE AND ES PROCESSOR COMMANDS	
Analysis Type	*Processor Commands*
Linear Statics	**FORM STIFFNESS/MATERIAL** **FORM FORCE/EXTERNAL**
Linear Dynamics	**FORM STIFFNESS/MATERIAL** **FORM MASS/CONSISTENT** **FORM FORCE/EXTERNAL**
Linear Buckling	**FORM STIFFNESS/MATERIAL** **FORM STIFFNESS/GEOMETRIC** **FORM FORCE/EXTERNAL**
Linear Vibration	**FORM STIFFNESS/MATERIAL** **FORM MASS/{CONS | DIAG}** **FORM STIFFNESS/GEOMETRIC** **FORM FORCE/EXTERNAL**
Nonlinear Statics	**FORM STIFFNESS/TANGENT** **FORM FORCE/INTERNAL** **FORM FORCE/EXTERNAL** **FORM FORCE/RESIDUAL**
Nonlinear Dynamics	**FORM STIFFNESS/TANGENT** **FORM MASS/{CONS | DIAG}** **FORM FORCE/INTERNAL** **FORM FORCE/EXTERNAL** **FORM FORCE/RESIDUAL**

Table 2.2 CORRESPONDENCE BETWEEN ANALYSIS TYPE AND ES PROCESSOR COMMANDS, concluded.	
All Pre-Processing	DEFINE ELEMENTS DEFINE FREEDOMS INITIALIZE
All Post-Processing	FORM { STRAIN | STRESS } POST { DISP | STRA | STRE }

2.3 Procedure Interface to ES Processors

In most cases, it is more convenient to use a command-language procedure to invoke ES processors, rather than invoking them directly (*i.e.*, with the [xqt or *run directives). This is especially true in problems that involve more than one ES processor (or element type), in which case a procedure can perform the necessary looping automatically. For such problems, use of a procedure interface can reduce the chance of introducing input errors, and can greatly simplify the task of writing generic *analysis* procedures by simplifying the references to element (ES) processors. A generic procedure has been developed to meet this need. It is called ES, and it may be viewed as a high-level interface for performing element functions with a general *set* of ES processors. Procedure ES can be invoked with the calling sequence given in Table 2.3-1 where each of the arguments in the calling sequence, except for FUNCTION and NUM_CON_DS, correspond to ES processor *macrosymbols* of the same name which were introduced in Section 2.2 (ES processor commands), and are defined in Section 2.4 (Glossary of ES Processor Macrosymbols).

The FUNCTION argument defines the processor function and may be set to any valid ES processor command. For example,

FUNCTION = 'FORM STIFFNESS/MATERIAL'

would perform element stiffness formation for all element (ES) processors, and element types, currently defined in the model. The user should consult the appropriate subsection in Section 2.2, to determine which macrosymbols (*i.e.*, procedure arguments) are relevant for a particular FUNCTION (*i.e.*, command) and which datasets are input/output as a result of the FUNCTION.

Table 2.3-1 Procedure ES Calling Sequence

Call	*Argument Name*		*Default Value*		*Comment*
*call ES (	FUNCTION	=	none	; --	. Command
	ES_PROC	=	none	; --	. Processor name
	ES_NAME	=	none	; --	. Element-type name
	ES_PARS	=	0.0	; --	. Research parameters
	ES_CORO	=	1	; --	. Corotational option
	ES_FIKX	=	none	; --	.
	ES_NL_GEOM	=	0	; --	. Geom. nonlinearity
	ES_NL_MATL	=	0	; --	. Matl. nonlinearity
	ES_NL_LOAD	=	0	; --	. Load nonlinearity
	ES_DIS_DS	=	STAT.DISP.1.1	; --	. Displacement dataset
	ES_DOF_DS	=	ES.DOFS	; --	. Active-freedom dataset
	ES_ECC_DS	=	WALL.PROP	; --	. Section. Eccentricity
	ES_FRC_DS	=	INT.FORC.1.1	; --	. Force-vector dataset
	ES_ROT_DS	=	STAT.ROTA.1.1	; --	. Rotation-vector dataset
	ES_SUM_DS	=	ES.SUMMARY	; --	. ES summary dataset
	ES_MASS_DIAG	=	DEM	; --	. Diagonal Mass Matrix
	ES_STRAIN_DS	=	' '	; --	. Elt. strain dataset index
	ES_STRESS_DS	=	' '	; --	. Elt. stress dataset index
	ES_STR_LOC	=	'CENTROIDS'	; --	. Stress/strain locations
	ES_STR_DIR	=	0	; --	. Stress/strain directions
	NUM_CON_DS	=	1	; --	. No. of constraint datasets
	ES_LOAD_SET	=	1	; --	. Load set number
	ES_LOAD_FACTOR	=	1.0	; --	. Load factor
	ES_COUNT	=	0	; --	. Element processor count
	LDI	=	1	; --	. Library number
)					.

ES procedure arguments ES_PROC (processor name), ES_NAME (element-type name), and ES_PARS (research parameters), are required *only for*:

FUNCTION = 'DEFINE ELEMENTS'

in which case, only one element-processor and element-type at a time are processed by each call to procedure ES. Thus, the DEFINE ELEMENTS calls must be made one at a time, until all pertinent element processors/types have been defined. Thereafter, for all other kinds of FUNCTION calls (*e.g.*, FUNCTION = 'FORM STIFFNESS'), procedure ES will process that FUNCTION (*i.e.*, command) for *all* element processors/types previously registered using the FUNCTION = 'DEFINE ELEMENTS' calls.

ES procedure argument NUM_CON_DS is pertinent *only for*:

FUNCTION = 'DEFINE FREEDOMS'

in which case, the argument indicates how many constraint (CON) datasets should be processed for *automatic degree of freedom suppression*, which removes irrelevant freedoms from the model, based on element type participation. The ES procedure assumes that consecutive CON datasets exist from CON..1 through CON..[NUM_CON_DS].

2.4 Glossary of ES Processor Macrosymbols

Formal definitions for each of the *macrosymbols* that may be used in conjunction with ES processors are given in Table 2.4-1. These definitions are given for both *input* macrosymbols, with which the user may set or reset various parameters associated with particular commands, and *output* macrosymbols, which are defined automatically as a result of the DEFINE ELEMENTS command (see Section 2.2).

The definitions given here also apply to the *procedure arguments* appearing in the generic ES procedure interface described in Section 2.3. These arguments intentionally have the same names as the ES processor macrosymbols.

> NOTE: Macrosymbols that are *output* by ES processors, or otherwise defined using the ES procedure, are underlined in the following glossary. These parameters must not be set by the user; they are made available to the user in order to facilitate procedure writing. For example, macrosymbol ES_EXPE_CMD is defined by procedure ES in response to a call with FUNCTION='DEFINE ELEMENTS'. This macrosymbol may then be expanded directly as the EXPE command for processor ELD. Most of these output parameters are also available to the user in the dataset ES.SUMMARY, where they are stored as nominal records of the same name.

Table 2.4-1 ES PROCESSOR MACROSYMBOL GLOSSARY

Macrosymbol	*Type*	*Definition*
ES_C	I	Continuity of interelement "displacement" field, *e.g.*, 0 ⇒ C^0 (displacement continuity only) 1 ⇒ C^1 (displacement and slope continuity)
ES_CLAS	A	Element class. Currently valid classes: BEAM, SHELL, SOLID, WILD (See Section 3.4 for examples.)
ES_CNS	I	Constitutive interface option (see Chapter 5). 0 ⇒elements use the standard constitutive interface for stress and tangent-modulus calculations; 1 ⇒elements compute their own stresses, but use standard constitutive interface for tangent-modulus calculations; ≥2 ⇒elements compute their own stresses and tangent-moduli; standard constitutive interface is not used.
ES_CORO	I	Corotation switch; employed by ES processor for automatic treatment of geometric nonlinearity due to large rotations. Relevant only if problem is geometrically nonlinear. (See Chapter 4 for an explanation of these options.) 0 ⇒ Off: corotational operations will be skipped; 1 ⇒ Low-Order Option: basic corotational transformations will be employed to enable large rotations; 2 ⇒ High-Order Option: a more accurate (and expensive) treatment of large rotations and consistent linearization than option 1 will be employed.
ES_COUNT	I	Element processor count. Relevant only for DEFINE ELEMENTS command. 0 ⇒ First element processor to be defined; create ES.SUMMARY dataset. > 0 ⇒ Not first element processor to be defined; use existing ES.SUMMARY dataset and Increment element processor number by one.
ES_DIM	I	Number of intrinsic element spatial dimensions, *e.g.*, 1 if ES_CLAS = BEAM 2 if ES_CLAS = SHELL 2 or 3 if ES_CLAS = SOLID

Table 2.4-1 ES PROCESSOR MACROSYMBOL GLOSSARY (continued)

Macrosymbol	*Type*	*Definition*
ES_DIS_DS	A	Name of system displacement-vector dataset (SYSVEC format). Relevant for most FORM commands and some POST commands. (Default: STAT.DISP.1.1)
ES_DOF_DS	A	Name of element freedom-table dataset (SYSVEC format). Relevant only for DEFINE FREEDOMS command. When using procedure ES, this dataset will automatically be created, initialized, and updated cumulatively with contributions from all pertinent ES processors. (Default: ES.DOFS)
ES_ECC_DS	A	First two words of dataset name that contains section property data including reference surface eccentricities. (Default: WALL.PROP).
ES_EXPE_CMD	A	Complete EXPE command line appropriate for defining elements with processor ELD. This macrosymbol is constructed only in response to a call to procedure ES with FUNCTION='DEFINE ELEMENTS'.
ES_FRC_DS	A	Name of system force-vector dataset (SYSVEC format), where element distributed forces (internal and/or external) are to be assembled. Relevant only for FORM FORCE command, in which case this vector must be created and initialized before issuing the command (or calling procedure ES). Contributions from all pertinent ES processors will be assembled into this vector if procedure ES is called with argument FUNCTION = 'FORM FORCE ...'. (Default: INT.FORC.1.1)
ES_LOAD_FACTOR	F	Load factor. (Default: 1.0)
ES_LOAD_SET	I	Load set number. (Default: 1)

Table 2.4-1 ES PROCESSOR MACROSYMBOL GLOSSARY (continued)

Macrosymbol	*Type*	*Definition*
ES_MASS_DIAG	A	First word of the dataset name for a diagonal mass matrix. (Default: DEM).
ES_MASS_DS	A	First two words of the dataset name for a diagonal mass matrix. (Default: <ES_MASS_DIAG>.DIAG).
ES_NAME	A	Name of element type, within current ES processor, to be processed by subsequent commands. (For example EX97 would be a valid element-type name within processor ES1.)
ES_NDOF	I	Number of freedoms per element node. Currently valid options (2, 3 or 6), *e.g.*,
		2 for 2-D solid elements (u, v)
		3 for 3-D solid elements (u, v, w)
		6 for beam, plate or shell elements $(u, v, w, \theta_x, \theta_y, \theta_z)$
ES_NEE	I	Number of element equations; = ES_NEN × ES_NDOF.
ES_NEN	I	Number of element nodes.
ES_NIP	I	Number of element integration points; *i.e.*, points at which stresses (continuum or resultants, depending on element type) are stored.
ES_NL_GEOM	I	Geometric nonlinearity switch; 0 ⇒ Off: problem is geometrically linear (small displacements/rotations);
		1 ⇒ Low-Order Option: problem is geometrically nonlinear, but elements should use linear strain-displacement relations. Meaningful only if ES_CORO > 0, so that large rotations can be handled automatically by the corotational algorithm;

Table 2.4-1 ES PROCESSOR MACROSYMBOL GLOSSARY (continued)

Macrosymbol	*Type*	*Definition*
		2 ⇒ High-Order Option: problem is geometrically nonlinear and elements should use *nonlinear* element strain-displacement relations. May be used in conjunction with ES_CORO > 0 to obtain higher-order accuracy for beam and shell elements that employ moderate-rotation strain-displacement relations;
		3 ⇒ same as 2 plus finite strains are expected.
ES_NL_LOAD	I	Load nonlinearity switch; 1 ⇒ process displacement-dependent element loads only; 0 ⇒ process displacement-independent element loads only.
ES_NL_MATL	I	Material nonlinearity switch; 1 ⇒ element is materially nonlinear; 0 ⇒ element materially linear.
ES_NORO	I	Element normal-rotation parameter. Relevant only for automatic freedom suppression of plate/shell elements. Indicates minimum angle (in degrees) between shell element normal vector and any computational basis vector — at each element node — below which the corresponding rotational freedom should be suppressed if no other elements are attached;
		=0 ⇒Element has normal-rotation ("drilling") stiffness; normal rotational freedoms will automatically be suppressed.
		>0 ⇒Element does not have normal-rotation ("drilling") stiffness; it is assumed that rotational stiffness exists about any computational axis that makes an angle of at least <ES_NORO> degrees with the element normal vector at an element node.

Table 2.4-1 ES PROCESSOR MACROSYMBOL GLOSSARY (continued)

Macrosymbol	*Type*	*Definition*
ES_NPAR	I	Number of element research parameters in array ES_PARS.
ES_NSTR	I	Number of stress components per integration point. Currently valid options compatible with standard constitutive interface: 8 for ES_CLAS = SHELL and ES_C = 0; 6 for ES_CLAS = SOLID and ES_DIM = 3; 6 for ES_CLAS = SHELL and ES_C = 1; 6 for ES_CLAS = BEAM and ES_C = 0; 4 for ES_CLAS = BEAM and ES_C = 1; 3 for ES_CLAS = SOLID and ES_DIM = 2.
ES_OPT	I	Element-type option number. Meaningful only to the element developer; it is the developer's numerical equivalent of ES_NAME.
ES_PARS	D	Array of element research parameters. Meaning depends on specific element type. (Consult appropriate section in CSM Testbed User's Manual, ref. 4).
ES_PROC	A	Name of element (ES) processor to be executed (*e.g.*, ES1, ES2, ...); currently relevant only as an argument for procedure ES, and only when argument FUNCTION = 'DEFINE ELEMENTS'.

Table 2.4-1 ES PROCESSOR MACROSYMBOL GLOSSARY (continued)

Macrosymbol	*Type*	*Definition*
ES_PROJ	I	Rigid-body *projection* option. Used to automatically remove (most of) the spurious energy generated by some elements during infinitesimal rigid-body motion. This is performed by operating on the element stiffness and force arrays with a projection matrix (or *projector*). The projector, and its derivative, can have a beneficial effect on element accuracy in both linear and geometrically nonlinear regimes. 0 ⇒Off: projection will be omitted; 1 ⇒Low-order Option: basic projection will be included; 2 ⇒High-order Option: basic projection plus a differential correction to the geometric stiffness will be included.
ES_ROT_DS	A	Name of system rotation pseudo-vector dataset (SYSVEC format). Relevant for most FORM commands during *geometrically nonlinear* analysis, but only if elements with rotational freedoms are involved. (Default: STAT.ROTA.1.1)
ES_SHAP	A	Shape of element surface used to define coordinate triad; currently recognized options: LINE, TRIA or QUAD.
ES_STOR	I	Number of "private" variables to be stored/retrieved for the element developer in dataset EFIL.<ES_NAME>.
ES_STR_DIR	I	Element stress/strain direction option. Indicates in which coordinate system element stresses or strains in datasets defined by <ES_STRESS_DS> or <ES_STRAIN_DS> (respectively) will be computed. Relevant only for command = 'FORM STRAIN' or 'FORM STRESS'. Valid options: 0 ⇒ element stress coordinate system 1 ⇒ material axes $\{x_m, y_m, z_m\} = \{x_g, y_g, z_g\}$ 2 ⇒ material axes $\{y_m, z_m, x_m\} = \{y_g, z_g, x_g\}$ 3 ⇒ material axes $\{z_m, x_m, y_m\} = \{z_g, x_g, y_g\}$ (Note: For isotropic materials, the first material axis is replaced by the corresponding global axis; see the SREF command under processor ELD in the Testbed User's Manual (ref. 4) for details. Default: 0)

Table 2.4-1 ES PROCESSOR MACROSYMBOL GLOSSARY (continued)

Macrosymbol	*Type*	*Definition*
ES_STR_LOC	A	Element stress/strain evaluation point option. Indicates where stresses or strains in datasets defined by <ES_STRESS_DS> or <ES_STRAIN_DS> (respectively) will be computed. Relevant only for command = 'FORM STRAIN' or 'FORM STRESS'. Currently valid options: CENTROIDS ⇒ element centroids; creates record group: CENTROIDS_<ES_STR_DIR>.1:nel NODES ⇒ element nodes; creates record group: NODES_<ES_STR_DIR>.1:nel INTEG_POINTS ⇒ element integration points; creates: INTEG_PTS_<ES_STR_DIR>.1:nel where "nel" is the number of elements in the dataset, and where macrosymbol ES_STR_DIR designates the directions of the stress/strain components, and is defined elsewhere in this Glossary. (Default: 'CENTROIDS')
ES_STRAIN_DS	A	Third part of element strain dataset name. The first name is always STRN; the second name is always the element-type name, *i.e.*, <ES_NAME>; and the third name, <ES_STRAIN_DS> must be a string of integers separated by periods. For example, if <ES_STRAIN_DS> = <step>.<iter> where <step> = 20 and <iter> = 3, and if the element-type name were EX97, then the full dataset name would be: STRN.EX97.20.3 Relevant only for command = 'FORM STRAIN'. (No default; absence means that strains will be stored (embedded) within dataset EFIL.<ES_NAME> only — *currently not implemented.*)

Table 2.4-1 ES PROCESSOR MACROSYMBOL GLOSSARY (continued)

Macrosymbol	*Type*	*Definition*
ES_STRESS_DS	A	Third part of element stress dataset name. The first name is always STRS; the second name is always the element-type name, *i.e.*, <ES_NAME>; and the third name, <ES_STRESS_DS> must be a string of integers separated by periods. For example, if <ES_STRESS_DS> = <step>.<iter>, where <step> = 20 and <iter> = 3, and if the element-type name were EX97, then the full dataset name would be: STRS.EX97.20.3 Relevant only for command = 'FORM STRESS'. (No default; absence means that stresses will be stored (embedded) within dataset EFIL.<ES_NAME> only.)
ES_SUM_DS	A	Name of ES summary dataset, which contains nominal records corresponding to most of the macrosymbol parameters appearing in this Glossary — for each ES processor/element defined in the model. Relevant for all ES commands. (Default: ES.SUMMARY)
ES_TWIS	I	Sign of twisting curvature for shell elements. +1 ⇒twist based on continuum definition of shear strain; −1 ⇒twist based on negative of continuum definition. (Note: The default convention for constitutive matrices output from processor LAU corresponds to the -1 option. Hence processor ES compensates for this reversal if ES_TWIS = +1.)
ES_TGC_DS	I	Name of nodal transformation dataset (QJJT.BTAB.*.*)
ES_XYZ_DS	I	Name of nodal coordinate dataset (JLOC.BTAB.*.*)
LDI	I	Logical device index or library number for archiving and retrieving data. (Default: 1)

2.5 ES Processor/Procedure Usage Examples

The following examples illustrate how element processors based on the generic structural-element (ES) processor template can be used to perform structural analysis with the CSM Testbed. For simplicity, separate examples of pre-processing (*i.e.*, model generation), linear analysis, nonlinear analysis, and post-processing (*i.e.*, stress recovery) are considered. The differences between employing individual ES processors directly versus accessing them indirectly using the generic ES procedure interface will be stressed, with an intended bias towards the high-level procedure interface.

2.5.1 Pre-Processing Examples

For clarity, a very simple problem will be considered and the use of the generic structural-element processor illustrated by showing all of the steps involved in generating a finite element model for this problem. Both the physical problem and the discrete model to be used is shown in Figure 2.2. The problem is a rectangular plate (10 in. by 5 in.), cantilevered on one edge, and loaded on the other edge by a concentrated lateral force. For the model, a 3 by 3 nodal grid is used, connected by a 2 by 2 mesh of 4-node plate/shell elements. The Testbed procedure for this model is given in Figure 2.3. The interpretation of each line of the procedure will now be described.

The *PROCEDURE statement in Figure 2.3 shows the arguments (parameters) for the procedure and sets default values for each of them (x-length, y-length, thickness, elastic modulus, Poisson's ratio, and precision). Thus the dimensions and properties of the plate model are parametrized, but the finite element discretization is fixed.† The next two lines ([xqt TAB and START 9) run the TAB processor and reserve space for a total of 9 nodes. Then, the JLOC command and subsequent data define the global coordinates for these 9 nodes in a rectangular, 3 by 3 grid.

† In practice, it is often the other way around: The model properties and dimensions are fixed, while the discretization is varied. We have fixed the discretization here merely to simplify the example.

for shell elements (see the CSM Testbed User's Manual, ref. 4, for details on processor LAU).

The key portion of the example begins with the *call to procedure ES with FUNCTION = 'DEFINE ELEMENTS', in which element type EX42 of processor ES1 is registered for participation in the model. This call also causes a number of element type-oriented macrosymbols, all beginning with ES_ to be defined — for example, ES_NEN (number of element nodes), ES_NIP (number of element integration points), etc. These macrosymbol values are automatically built into the character string macrosymbol ES_EXPE_CMD (by procedure ES), which serves as the EXPE command for processor ELD. Note that in the PLATE_MODEL example, <ES_EXPE_CMD> appears immediately after [XQT ELD (execute processor ELD). This sequence causes the ES_EXPE_CMD macrosymbol to expand internally into the following command line:

```
EXPE <ES_NAME> 4 <ES_OPT> <ES_NEN> <ES_NDOF> --
                                <ES_NST> 1 10<ES_DIM> <CSM_PRECISION>
```

which would eventually decode into:

```
EXPE EX42 4 2 4 6 0 1 102 2
```

The execution of processor ELD with the above EXPE command is necessary for generation of various element datasets such as DEF.EX42.* and DIR.EX42.* (see the CSM Testbed User's and Dataset Manuals, refs. 4 and 5, respectively). Note that the ES_ macrosymbols referenced in this example are all described in the Macrosymbol Glossary (Section 2.4).

Next, the NSECT command is used as a section property pointer. NSECT=1 means that the integrated constitutive matrix stored in the first column of dataset PROP.BTAB.2.101 will be employed by all elements whose nodal connectivity is defined on the following lines. The element nodal connectivity is then defined for four 4-node elements. Note that the numbering convention is counter-clockwise within each element. (see Figure 2.1)

Boundary conditions and loads are then defined using the CON command of processor TAB, and the SYSVEC command of processor AUS, respectively (see Figure 2.3). The CON command suppresses all freedoms along $x = 0$ (the built-in edge), and the SYSVEC command distributes transverse nodal forces along the other edge, which add up to a unit

CON command suppresses all freedoms along $x = 0$ (the built-in edge), and the SYSVEC command distributes transverse nodal forces along the other edge, which add up to a unit load. Note that both boundary conditions and loads have been defined with respect to nodes rather than elements.

Finally, the ES procedure is called again to perform *automatic degree of freedom suppression*, using the DEFINE FREEDOMS command. In this case, the effect will be for processor ES1 to suppress drilling rotational freedoms (*i.e.*, the sixth degree of freedom) at all nodes, since element type EX42 has no stiffness associated with these freedoms. If the plate had blade stiffeners which were also modeled with EX42 elements, all six freedoms at nodes along the plate/stiffener intersection lines would be retained, since at least one of the intersecting elements at those nodes would possess the necessary stiffness.

2.5.2 Linear Analysis Examples

A sample linear static analysis procedure, which employs the generic ES procedure (Section 2.3) to invoke the appropriate ES processors is shown in Figure 2.4. For purposes of illustration, the problem has been kept simple (notice that there are no procedure arguments), but keep in mind that many analysis procedures may involve more sophisticated features.

The procedure in Figure 2.4 can be read as follows. The [xqt E directive causes processor E to construct *.EFIL.* datasets for all participating element types. Note that, while space for these datasets is reserved in the database, meaningful data has not yet been deposited there.

The first call to procedure ES then causes initialization data to be stored in the EFIL datasets by all participating ES processors — as prescribed by previous calls to procedure ES with function equal to 'DEFINE ELEMENTS'. For example, if the pre-processing example given in Figure 2.3 had preceded the call to procedure L_STATIC, then only processor ES1, element type EX42, would be invoked. More information on the effect of the 'INITIALIZE' command may be found in Section 2.2.2.

Next, the element material (linear) stiffness matrices are formed for all elements in the model by the second call to procedure ES. That is, *call ES (function='FORM STIFFNESS/MATL'). The element matrices are deposited in Segment 5 of the EFIL dataset (see Chapter 6). Since no other arguments are employed in this call to ES, the default values are implied. Thus, for example, the problem is assumed to be linear (ES_NL_GEOM = 0), and there is no need for a displacement dataset (whose name is given by argument ES_DIS_DS) to be input by the ES processors.

Next, assembly of the element stiffness matrices into a system matrix is performed by processor K. The element matrices have already been transformed to the computational or nodal degree of freedom bases, so that the function of processor K is merely to add appropriate submatrices.

Finally, processors INV and SSOL are executed to factor and solve the assembled system of equations, respectively. The displacement solution will be stored, as indicated by the RESET command for processor SSOL, in dataset STAT.DISP.1.1.

2.5.3 Nonlinear Analysis Examples

A brief example of how to employ ES processors in nonlinear analysis procedures by included selected excerpts from an actual nonlinear static analysis procedure that will hopefully convey the essential aspects. Typically, engineering-oriented users will invoke an existing nonlinear analysis procedure rather than writing their own, so this example is intended more for researchers involved in algorithm development.

The "skeleton" of a nonlinear static analysis procedure is shown in Figure 2.5, with only those aspects involving ES processors shown. There is very little difference in the use of the ES procedure to invoke ES processors from what was employed in the *linear* static analysis procedure (compare with Figure 2.4), except that some additional arguments must be explicitly defined.

First, the usual call to 'INITIALIZE' all ES processors is present. This call enables the participating element processors to generate, and store, any data that will be used repeatedly during the analysis — rather than having to recompute it at each iteration of every analysis load step.

Next, the nested load-step and iteration loops, typical of most incremental/iterative nonlinear solution algorithms for structural analysis are encountered. Within these loops, it is necessary to compute residual force vectors (right-hand sides) at each iteration, and tangent stiffness matrices at selected load steps/iterations.

The assembled residual force vector is computed by (i) zeroing a system internal force vector (using processor VEC), (ii) calling ES to form/assemble all element contributions to the internal force vector, and (iii) subtracting the assembled internal force vector from a load-step scaled external force vector, which is assumed to have been generated elsewhere.

Notice that in the FORM FORCE/INT call to ES, arguments ES_NL_GEOM, ES_CORO, ES_DIS_DS, ES_ROT_DS and ES_FRC_DS are each explicitly defined. The reader is urged to look up these arguments up in the Macrosymbol Glossary, Section 2.4. These arguments let the ES processors know that the problem is geometrically nonlinear (both globally and at the element level). The corotational algorithm is to be employed to make the rotational motion "appear" small at the element level, but allow it to be arbitrarily large globally (see Chapter 4). The current displacement dataset is called TOT.DISP.<$step>, where

<$step> is the load-step number; the current rotation (pseudo-vector) dataset is called TOT.ROTA.<$step>; and the current system internal force vector into which the element contributions are to be assembled is called INT.FORC.<$step>.

Finally, the formation/factorization of the tangent stiffness matrix is shown in Figure 2.5, which involves (i) formation/transformation of the element tangent stiffness matrices by ES processors, which are deposited in the *.EFIL.* datasets; (ii) assembly of the element matrices into the system tangent stiffness matrix by processor K; and (iii) factorization of the assembled (system) tangent stiffness matrix by processor INV. Note that both material and geometric stiffness contributions have been superimposed at the element level.

2.5.4 Post-Processing Examples

Post-processing refers to functions which can be performed after the solution has been obtained for a linear or nonlinear static structural analysis. For example, in a nonlinear static (or transient) analysis, the user may choose not to archive the stresses and strains which were used as intermediate variables during the process of obtaining a displacement solution history. The user may then compute stresses and/or strains at selected load (or time) steps and save these in the database for perusal. The end-user phase of post-processing is of course the actual printing or display of the results (displacements, stresses, strains, etc.); however, the main interest here is in the prerequisite functions that are performed by the ES processors.

An example of a post-processing procedure is given in Figure 2.6 that employs ES processors to form, and archive both stresses and strains in the database after the mainstream analysis has already been performed. Procedure STRESS_STRAIN contains arguments to select the stress/strain locations (the default is at element centroids), component directions (the default is in the element local stress/strain coordinate system), geometric nonlinearity and corotational flags, the range of load or time steps to process, and the root name of existing displacement and rotation (for nonlinear analysis) datasets to be employed for strain computation.

Note that there is a step loop in the procedure, and that both stresses and strains for all participating elements are formed using a single call to procedure ES per load step. This is because the FORM STRESS command automatically causes strains to be formed as well

as stresses, and when both ES_STRAIN_DS and ES_STRESS_DS are explicitly defined, then both of these quantities are also written to the database. The user is advised to refer to Sections 2.3 (ES procedure) and 2.4 (Macrosymbol Glossary) for a better understanding of the calling sequence used in Figure 2.6 for procedure ES.

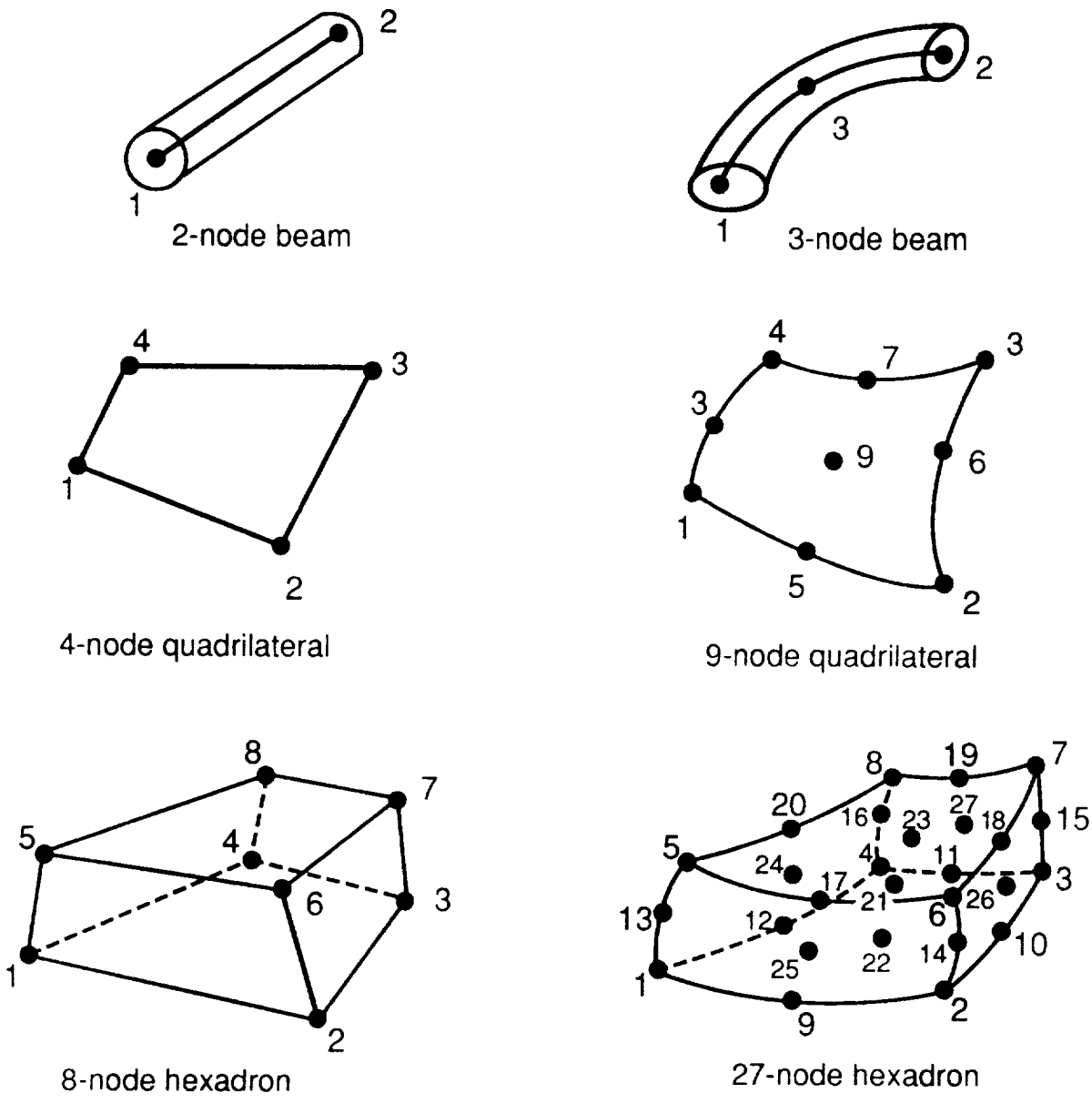
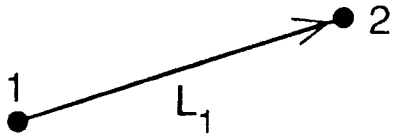
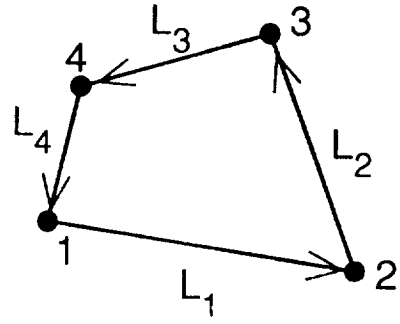


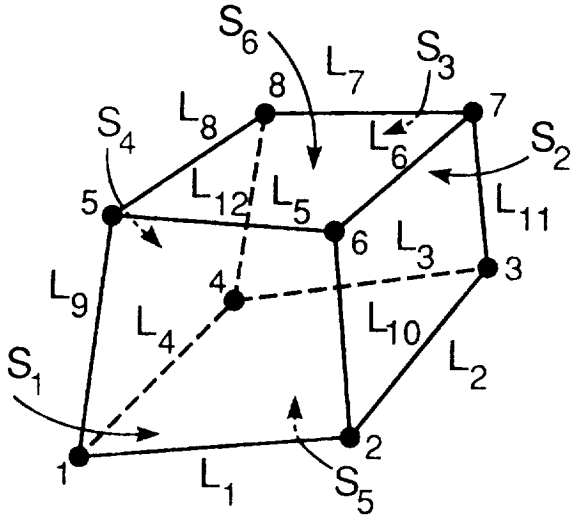
Figure 2.1a Standard Element Node Numbering Conventions.



1 D Elements/Lines (L)



2 D Elements/Surfaces (S)



3 D Elements

Figure 2.1b Standard Element Boundary Number Conventions.

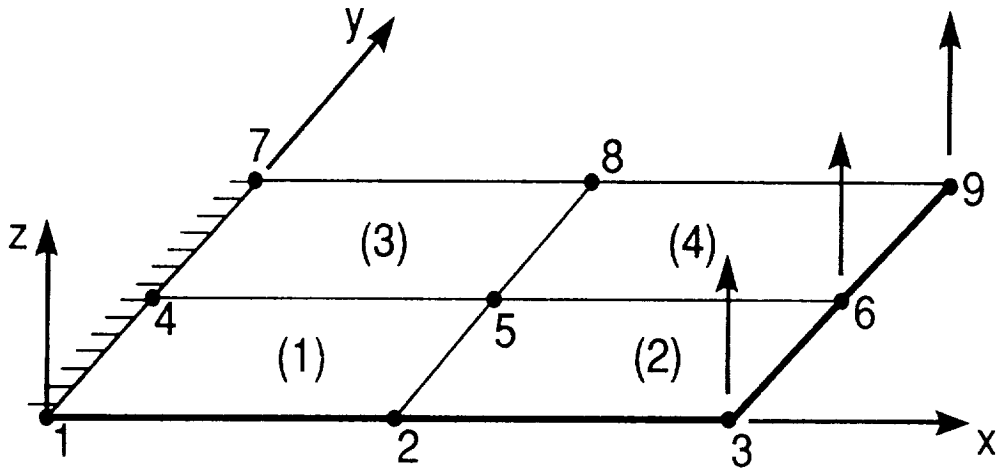


Figure 2.2 Sample Problem for ES Processor Usage.

```
*Procedure PLATE_MODEL ( Lx=10.; Ly=5.; h=.1; E=1.e7; PR=.3; prec=2 )
     [xqt TAB
      START 9
.
. GENERATE NODES  ( rectangular grid, 3 x 3 nodes )
.
      JLOC
         1    0.,  0. , 0.    [Lx],  0. , 0.  3, 1, 3
         3    0., [Ly], 0.    [Lx], [Ly], 0.
.
. TABULATE MATERIAL AND SECTION PROPERTIES
.
     [xqt AUS
         TABLE(NI=16,NJ=1): OMB DATA 1 1
         *def G = < [E] / < 2.*<1. + [PR]> > >
          I=1,2,3,4,5,6  : J=1 : [E] [PR] [E] <G> <G> <G>
         TABLE(NI=3,NJ=1,ITYPE=0): LAM OMB 1 1
          I=1,2,3        : J=1:   1 [h] 0.0
.
. RUN CONSTITUTIVE PRE-PROCESSOR
.
     [xqt LAU
.
. GENERATE ELEMENTS
.
. ------------------------------------------------------------------
    *call ES ( function='DEFINE ELEMENTS'; ES_PROC=ES1; ES_NAME=EX42 )
. ------------------------------------------------------------------
     [xqt ELD
         <ES_EXPE_CMD>
          NSECT = 1
.          elt 1         elt 2          elt 3          elt 4
          1 2 5 4   :  2 3 6 5   :    4 5 8 7    :    5 6 9 8
.
. IMPOSE BOUNDARY CONDITIONS
.
   [xqt TAB
    CON
       ZERO 1:6 : 1 : 4 : 9
.
. APPLY LOADS
.
   [xqt AUS
    SYSVEC : APPL FORC 1
   i=3 : j=3 : .25   :   i=3 : j=6 : .50    :   i=3 : j=9 : .25
.
. ------------------------------------------------------------------
    *call ES ( function = 'DEFINE FREEDOMS' )
. ------------------------------------------------------------------
*end
```

Figure 2.3 Sample Pre-Processing Procedure.

```
*procedure L_STATIC
..
.  ---------------------------
.  Initialize Element Datasets
.  ---------------------------
   [xqt E
.
.  -------------------------------------
.  Initialize Element Computational Data
.  -------------------------------------
   *call ES ( function = 'INITIALIZE' )
.
.  ---------------------------------------
.  Form Element Material Stiffness Matrices
.  ---------------------------------------
   *call ES ( function = 'FORM STIFFNESS/MATERIAL' )
.
.  --------------------------------
.  Assemble Material Stiffness Matrix
.  --------------------------------
   [xqt K
.
.  ----------------------
.  Factor Stiffness Matrix
.  ----------------------
   [xqt INV
.
.  ----------------------
.  Solve for Displacements
.  ----------------------
   [XQT SSOL
    RESET SET=1, CON=1
.
*end
```

Figure 2.4 Sample Linear Static Analysis Procedure.

```
*procedure NL_STATIC ( NUM_STEPS = 1; NUM_ITERS = 10 )
. ------------------------------------
. Initialize Element Computational Data
. ------------------------------------
  *call ES ( function = 'INITIALIZE' )
        :
  *do $step = 1, [NUM_STEPS]
     *do $iter = 1, [NUM_ITERS]
        -------------------------
        FORM RESIDUAL FORCE VECTOR
        -------------------------

        [xqt VEC
         INT.FORC <- 0.
         :
        *call ES ( function    = 'FORM FORCE/INTERNAL'; --
                   es_nl_geom =  2; es_coro = 1; --
                   es_dis_ds  = TOT.DISP.<$step>; --
                   es_rot_ds  = TOT.ROTA.<$step>; --
                   es_frc_ds  = INT.FORC.<$step> )
       [xqt VEC
        RES.FORC  <-  <load_factor> * EXT.FORC  -  INT.FORC.<$step>
         :
        ------------------------------------
        FORM/FACTOR TANGENT STIFFNESS MATRIX
        ------------------------------------
       *call ES ( function    = 'FORM STIFFNESS/TANGENT' ; --
                  es_nl_geom =  2; es_coro = 1; --
                  es_dis_ds  = TOT.DISP.<$step>; --
                  es_rot_ds  = TOT.ROTA.<$step>; --
                  es_frc_ds  = INT.FORC.<$step> )
       [xqt K
       [xqt INV
         :
     *enddo
  *enddo
```

Figure 2.5 Sample Nonlinear Static Analysis Procedure Excerpts.

```
*procedure STRESS_STRAIN ( LOCATION  = CENTROIDS ; DIRECTION = 0   ; --
                           NL_GEOM   = 0         ; CORO      = 0   ; --
                           NUM_STEPS = 1         ; STEPS     = 1:1 ; --
                           DIS_DS    = TOT.DISP  ; ROT_DS    = TOT.ROTA )
.  ----------------------
.  Loop on Solution Steps
.  ----------------------
   *def/i steps[1:[num_steps]]  =  [steps]
   *do $is = 1, [num_steps]
      *def/i step = <steps[<$is>]>
.     ------------------------------------------------
.     Invoke Element Processors to Form Stress/Strain
.     ------------------------------------------------
      *call ES ( function    = 'FORM STRESS'   ; --
                 es_nl_geom = [nl_geom]        ; --
                 es_coro    = [coro]           ; --
                 es_dis_ds  = [DIS_DS].<step>  ; --
                 es_rot_ds  = [ROT_DS].<step>  ; --
                 es_str_dir = [DIRECTION]      ; --
                 es_str_loc = [LOCATION]       ; --
                 es_strain_ds = <step>         ; --
                 es_stress_ds = <step>  )
   *enddo
*end
```

Figure 2.6 Sample Post-Processing Procedure.

3. DEVELOPER INTERFACE

CHAPTER OUTLINE

Section	*Title*	*Description*
3.1	**Overview**	Introduces the developer to basic concepts and overall approach for adding new elements (*i.e.*, ES processors) to the CSM Testbed.
3.2	**Standard ES Kernel Routines**	Summarizes, then describes calling sequences for, each of the standard element functional routines that must be supplied by the developer to complete the implementation of a new element (ES) processor.
3.3	**Glossary of Standard ES Kernel Arguments**	Defines, in detail, all arguments (and related parameters) appearing in the calling sequences described in Section 3.2. The definitions are arranged as an alphabetical table for easy reference. (Most of the arguments are shared by more than one standard kernel routine.)
3.4	**Examples of Specific Element Types**	Looks at some of the standard element types (*i.e.*, classes) recognized by the generic structural-element processor; for example, beam, shell and solid elements. Indicates how the various kernel arguments, described in Sections 3.2–3.3, should be defined for such elements. Elements that do not fall within this framework may be defined as "wild" elements, which are treated as a "black box" by the generic structural-element processor.
3.5	**Step-by-Step Installation of New ES Processors**	Gives step-by-step instructions for creating a new structural-element (ES) processor as a standard CSM Testbed module, which may be used either alone, or in conjunction with other ES processors for general structural analysis.

3.1 Overview

3.1.1 Basic Approach

To add a new structural element — or family of elements — to the CSM Testbed using the generic element processor template, a developer needs only to provide a set of standard element subroutines (currently in FORTRAN-77). The names and argument lists for these so-called *kernel* routines, are described in Sections 3.2 and 3.3. The standard kernel routines include such functions as formation of the element stiffness matrices (material and geometric), the element force vectors (internal and external) and element strains. Another important kernel routine is the basic definition routine (ES0D), in which the element developer sets key parameters that describe the new element(s) to the generic glement processor.

These standard kernel routines feature extendible argument arrays that enable a broad range of elements to be implemented. However, if *your* element does not fit within the standard framework, a number of alternative options are discussed in Section 3.1.2.

After these kernel routines have been completed, the developer can create his/her own structural element (ES) *processor* by simply executing a standard "link" procedure (see Section 3.5). The link procedure will create a new ES processor by combining the developer's code with a copy of the Generic Element processor "shell" (*i.e.*, driver). The resulting executable will look to the CSM Testbed just like any other ES processor — it will automatically employ the same command language and database entities — except that it will contain *only* the developer's new element(s). Thus, the developer can safely revise and test his/her own new ES processor (*e.g.*, interactively, in stand-alone mode) without having to worry about impacting other developers, or inadvertently corrupting the integrity of preexisting element processors.

Once a new ES processor has been created, it may be used immediately to perform analysis, using the analysis procedures that employ the generic ES *procedure* interface described in Chapter 2. *This fact is important.* The developer should not have to make any changes to existing analysis procedures — except for the selection of the element processor/type

during pre-processing (see Chapter 2) — in order to be able to apply the new elements to structural analysis problems. The developer's new ES processor immediately becomes a standard module in the CSM Testbed, and may be used in the same analysis with other ES processors.

Element developers who have never *used* the CSM Testbed before should consult Chapter 2 and the CSM Testbed User's Manual (ref. 4) for instructions on how to begin problem-solving once the element implementation phase is completed.

3.1.2 Standard versus Non-Standard ("Wild") Elements

Implementation of element types that do not clearly fit within the standard framework described here may require one of the following approaches: (i) straightforward extension of the examples presented in Section 3.4.1–3.4.3 for *standard* elements; (ii) use of the "wild" element approach for nonstandard elements; (iii) application of pressure on the ES processor shell *architect* to extend the standard framework; or, as a last resort, (iv) development of a special-purpose ES processor shell by the element developer.†

The terms *standard* and *non-standard* (or "wild") elements are defined as follows. *Standard* elements are fully recognized by the ES processor shell, so that standard operations such as:

- Coordinate transformations
- Corotational updates (for geometric nonlinearity)
- Constitutive processing (for linear and nonlinear materials)
- Automatic Freedom Suppression

may be performed *automatically* — by the shell. On the other hand, *wild* elements are treated as "black boxes" that must perform all such operations on their own — *i.e.*, at the

† While this last option is not recommended due to the obvious effort associated with replicating all of the processor overhead functions — such as command interpretation and formal database transactions — it is a viable one, and may be the only option in certain cases. In fact, as long as the developer is careful to employ the same command language as the Generic Element processor, this approach should produce special-purpose ES processors that are completely compatible with the standard ES processors, and may be used in existing analysis procedures without special modifications therein.

kernel level. Thus, the element developer is faced with the usual trade-off between novelty and conformity. If the element fits into one of the standard molds, then the developer's responsibility will be less than if the element is very exotic (*i.e.*, wild).

3.1.3 Special Features; Geometric/Material Nonlinearity

The reader may be entertaining all kinds of questions right now concerning how the generic element processor will accommodate all of the complexities of his/her new element. One of the most frequent questions posed by prospective element developers deals with the treatment of nonlinearity; especially geometric and material nonlinearity. The answer is that developers are free to handle such complexities on their own; however, some intrinsic capabilities have been built in to the Generic Element processor shell (driver) to alleviate the developer's burden. In particular, a *corotational* option is available for beam, shell and (thin) solid elements, which will either automatically upgrade a linear element to a geometrically nonlinear element, or extend the range of applicability of a moderate-rotation nonlinear element to handle arbitrarily large rotations.

A by-product of the built-in corotational methodology for geometric nonlinearity is its *linear* counterpart — the rigid-body *projection* operator. This latter option can be used by element developers (in some cases) to correct the intrinsic element rigid-body errors that plague some elements (*e.g.*, non-isoparametric, curved elements) in both linear and nonlinear analysis.

Similar interfaces are being planned to automate material nonlinearity — and thus decouple the element developer's work from the constitutive modeler — but at present the element developer is on his/her own when it comes to nonlinear materials. (Note: There is a standard, built-in interface for linear constitutive behavior that the element developer can exploit to get started.) More information on such special features is presented in Chapters 4 and 5.

3.1.4 Outline of this Chapter

The following sections contain all of the information necessary for an element developer to add a new element (or family of elements) to the CSM Testbed — as an independent ES processor. First in Section 3.2, we give a summary of the standard kernel routines that need to be supplied by the element developer, followed by subsections containing the detailed calling sequences (*i.e.*, input/output arguments) for each of these standard routines. However, since many of the argument variables are shared by a number of kernel-routine entry points, the full definitions for these variables are described in the glossary given in Section 3.3. This approach avoids needless duplication of certain explanations, and provides both a quick-reference manual for the experienced ES processor developer, and a more in-depth document for newcomers.

In Section 3.4, guidelines are given for implementation of specific classes of elements, for example, beam, shell and solid continuum elements. Also, some preliminary provisions for more exotic structural elements are discussed in the subsection on nonstandard elements.

Finally, in Section 3.5, specific step-by-step installation instructions are given for adding new element (ES) processors to the CSM Testbed, including the minor variations associated with different computer operating systems.

3.2 Standard ES Kernel Routines

The *standard*, developer-supplied kernel subroutines employed by the generic element (ES) processor shell are described in the following sections. A summary of the individual entry point names and their respective functions is given in Table 3.1. Then in Sections 3.2.1 through 3.2.15, the calling sequence for each of the entry points is given, with a brief description of the various input/output arguments that appear in the call. Readers should refer to Section 3.3 (Glossary of Standard ES Kernel Arguments) for a detailed explanation of each argument, and of the parameters used to dimension and index them.

Table 3.1 STANDARD KERNEL ENTRY POINTS

Entry Point	*Description*
ES0D	Element DEFINITION routine. Defines basic element parameters as a function of element name.
ES0E	Element STRAIN routine. Computes element strains (linear or non-linear) as a function of nodal displacements.
ES0FB	Element BODY FORCE routine. Computes consistent element external force vector based on distributed body loads.
ES0FI	Element INTERNAL FORCE routine. Computes element internal force vector as a function of element stresses and, optionally, nodal displacements. (NOTE: Automatically accounts for forces due to specified displacements and thermal loads using the input stresses.)
ES0FL	Element LINE FORCE routine. Computes consistent element external force vector based on distributed line loads.
ES0FP	Element PRESSURE FORCE routine. Computes consistent element external force vector based on distributed pressure loads.
ES0FS	Element SURFACE FORCE routine. Computes consistent element external force vector based on distributed surface loads.
ES0I	Element INITIALIZATION routine. Precomputes special element data that is to be stored in the database for subsequent retrieval and use by other element routines.

continued ...

Table 3.1 STANDARD KERNEL ENTRY POINTS (concluded)

Entry Point	*Description*
ES0KG	Element GEOMETRIC STIFFNESS routine. Computes element geometric stiffness matrix (linear and/or nonlinear) as a function of element stresses and (optionally) nodal displacements.
ES0KL	Element LOAD STIFFNESS routine. Computes element load stiffness matrix as a function of element *live* loads and displacements.
ES0KM	Element MATERIAL STIFFNESS routine. Computes element material stiffness matrix (linear and/or nonlinear) as a function of element constitutive matrices and (optionally) nodal displacements.
ES0MC	Element CONSISTENT MASS routine. Computes element consistent mass matrix as a function of element inertial properties and nodal coordinates.
ES0MD	Element DIAGONAL MASS routine. Computes element diagonal (lumped) mass matrix as a function of element inertial properties and nodal coordinates.
ES0N	Element NORMAL vector routine. Computes unit normal vectors at element nodes for plate/shell elements. Used primarily to affect automatic degree of freedom suppression.
ES0S	Element STRESS routine. Computes element stresses as a function of nodal displacements and constitutive matrices. (Required only for assumed stress elements).
ES0T	Element TRANSFORMATION routine. Computes orthogonal transformations at element integration points from the element's local (stress) system to the Cartesian (corotational) system in which the element's nodal coordinates are expressed.

Remark 3.1 For additional insight, Table 3.2 shows the correspondences between the ES processor commands discussed in Chapter 2 and the developer-supplied standard kernel routine entry points described in this chapter. This information can be useful to the developer for determining the most important entry points to implement — based on the analysis requirements of prospective users — and for prioritizing their implementation in case there isn't time to implement them all at once.

Table 3.2 COMMAND/SUBROUTINE CORRESPONDENCE

ES Processor Commands versus Kernel Routines

Command	*Routines*	*Qualifications*
DEFINE ELEMENTS	**ES0D**	
DEFINE FREEDOMS	**ES0D**	
	ES0N	only if DEFS(pdNORO) > 0
INITIALIZE	**ES0D**	
	ES0I	
FORM STIFFNESS/MATL	**ES0D**	
	ES0KM	
	ES0E	only if materially nonlinear
	ES0S	only if materially nonlinear and DEFS(pdCNS) > 0
	ES0T	only if nonisotropic material
FORM STIFFNESS/GEOM	**ES0D**	
	ES0KG	
	ES0KL	only if <ES_NL_LOAD> is true
	ES0E	
	ES0S	only if DEFS(pdCNS) > 0
	ES0T	only if nonisotropic material
FORM FORCE/INTERNAL	**ES0D**	
	ES0FI	
	ES0E	
	ES0S	only if DEFS(pdCNS) > 0
	ES0T	only if nonisotropic material

continued ...

Table 3.2 COMMAND/SUBROUTINE CORRESPONDENCE (concluded)

ES Processor Commands versus Kernel Routines

Command	*Routines*	*Qualifications*
FORM FORCE/EXTERNAL	**ES0D**	
	ES0FB	only if body loads present
	ES0FL	only if line present
	ES0FS	only if general surface loads present
	ES0FP	only if pressure loads present
FORM STRAIN	**ES0D**	
	ES0E	
	ES0T	only if nonisotropic material or rotated axes
FORM STRESS	**ES0D**	
	ES0E	
	ES0S	only if DEFS(pdCNS) > 0
	ES0T	only if nonisotropic material or rotated axes
FORM MASS/CONS	**ES0D**	
	ES0MC	
FORM MASS/DIAG	**ES0D**	
	ES0MD	
POST DISPLACEMENT	**ES0D**	
	ES0PD	

Remark 3.2 For the complete picture, the developer should refer to Table 3.2 in conjunction with Table 2.2, which shows the correspondences between *analysis type* and ES processor *commands*. Composition of Tables 2.2 and 3.2 thus indicates the relationship between analysis type and kernel routine entry points, *i.e.*, which entry points the developer must supply to enable the user to perform a given type of structural analysis.

3.2.1 Subroutine ES0D: Element Definition

Entry point ES0D defines element attributes for all element types implemented within a single ES processor. This enables the ES processor shell to make logical and dimensional decisions about how to call other standard kernel routines, as well as how to access the database.

Calling Sequence

call ES0D (eltnam, eltnum, ctls, defs, dofs, nodes, pars, status)

Input Arguments

ELTNAM Element-type name. developer may have as many different element types implemented within an individual ES processor as desired. The element-type name currently corresponds to the name used to define an element with the EXPE command of processor ELD.

ELTNUM Element sequence number. Useful only for diagnostic purposes.

CTLS() List of element control parameters. Typically not relevant for this subroutine; primary use is in formation routines such as ES0KM, etc.

PARS() Array of element-developer's research parameters. Useful for defining variations on basic element types before making them standard options specified by ELTNAM. These parameters are currently input to ES processors through the command macrosymbol ES_PARS.

Output Arguments

DEFS() List of intrinsic element definition parameters for the element type specified by ELTNAM. All entries in this array must be defined by the developer (see Section 3.3 Glossary). In particular, note that DEFS(pdOPT) — the element option number — will become the developer's numerical replacement for ELTNAM in calls to all other standard kernel routines.

DOFS() Table of potentially active degrees of freedom at element nodes for the element type specified by ELTNAM.

NODES() Element node-sequence mapping (for element type specified by ELTNAM).

STATUS Subroutine return status ($\geq 0 \Rightarrow$OK).

Relevant Processor Commands

All ES Processor commands

Remark 3.3 Subroutine ES0D should be the first standard kernel routine written by the element developer. It is a prerequisite for the automatic operation of the other entry points, and should make implementation of the other routines more straightforward, since a basic understanding of the element intrinsic parameters (*e.g.*, the DEFS array) will have been acquired by the developer.

Remark 3.4 Currently, the ES0D routine is *only called once for a given element type*. Thus, the values of the output arguments must be representative of the entire type (*i.e.*, ELTNAM or DEFS(pdOPT)) rather than something that varies from element to element. We plan to remove this limitation in future implementations of the ES shell.

3.2.2 Subroutine ES0E: Element Strains

Entry point ES0E should compute element strains at all "integration points" within an element. It is not required for elements that form their own stresses, directly, *e.g.*, elements based on an *assumed stress* formulation (see argument DEFS(pdCNS) in Section 3.3 for details). The element strains may be either continuum quantities or resultant (section-averaged) quantities, depending on the element type (see Section 3.4 for examples). The strains are used either as input to constitutive routines (see Chapter 5) or simply output to the database for post-processing.

Calling Sequence

call ES0E (eltnum, ctls, defs, dofs, nodes, pars, x, d, store, e, status)

Input Arguments

ELTNUM Element sequence number. Useful only for diagnostic purposes.

CTLS() List of element control parameters. Of particular importance is entry CTLS(pcNLG), which specifies the level of geometric nonlinearity to employ for strain computation.

DEFS() List of element definition parameters. The individual parameters which are relevant here will depend on the developer. Of particular importance, however, is DEFS(pdOPT), which is the element-type number (the numerical equivalent of ELTNAM in subroutine ES0D).

DOFS() List of potentially active element nodal degrees of freedom.

NODES() Element nodal resequencing map. This mapping will ordinarily not be necessary in this subroutine as all nodal arrays will already have been reordered into the developer's internal sequence upon entry.

PARS() Array of element-developer's research parameters. Useful for defining variations on basic element types before making them standard options specified by ELTNAM. These parameters are currently input to ES processors through the command macrosymbol ES_PARS.

X() Element nodal coordinate vectors; in initial configuration, element basis and developer's node sequence.

D() Element nodal displacement vectors; in element basis and developer's node sequence. If CTLS(pcCORO) > 0, then displacements are relative to corotational element frame. Otherwise, they are relative to initial element frame.

STORE() Element private storage array (defined using subroutine ESOI).

Output Arguments

E() Strains (continuum or resultant, depending on element class) at element integration points, expressed in element developer's stress bases. Linear or nonlinear strain-displacement relations may be required, as specified by CTLS(pcNLG). (See Sections 3.2 and 3.4 for details and examples.)

STATUS Subroutine return status ($\geq 0 \Rightarrow$OK).

Relevant Processor Commands

FORM STRAIN

FORM STRESS

FORM FORCE/INTERNAL

FORM STIFFNESS/GEOMETRIC

FORM STIFFNESS/MATERIAL (if materially nonlinear)

FORM STIFFNESS/TANGENT

Remark 3.5 It is not always necessary for the developer to provide *nonlinear* strain-displacement relations, especially if the built-in corotational option can be employed using the macrosymbol ES_CORO. See Chapter 4 for details.

3.2.3 Subroutine ES0FI: Element Internal Force Vector

Entry point ES0FI should compute the element internal force vector. This function is important in both linear and nonlinear analysis. In linear analysis, the element internal force vectors may be assembled to construct reaction forces, and/or to generate external forces due to specified displacements, initial strains or initial stresses. In nonlinear analysis, the element internal force vectors are assembled to compute the out-of-balance (residual) force vector for the system, which is essential for most nonlinear iteration algorithms. Inputs to this routine include stresses and displacements. It is strongly recommended that the element internal force vector, which is output, be computed as an explicit function of stresses (if possible), so that initial stress and strain effects are appropriately accounted for. The element displacement vector is provided only for those elements that employ explicit geometric nonlinearity in the event that CTLS(pcNLG) > 1 (which is not absolutely necessary if the automatic corotational option has been invoked by the ES processor user; see Chapters 2 and 4).

For assumed-displacement type elements, the internal-force vector can usually be expressed in the form:

$$\mathbf{f}^{int} = \int_{Vol} \mathbf{B}^T \boldsymbol{\sigma} \, dV$$

where $\mathbf{B}$ is the incremental strain-displacement interpolation matrix, and $\boldsymbol{\sigma}$ is the generalized stress tensor (corresponding to subroutine argument S). The volume integral is typically approximated by numerical integration, with $\boldsymbol{\sigma}$ evaluated only at integration points. For beam or shell elements, the integrals may become one-dimensional (curve) or two-dimensional (surface) domains, respectively, with the $\boldsymbol{\sigma}$ tensor (*i.e.*, the S array) representing stress resultants.

Remark 3.6 The internal force vector is necessary for both linear and nonlinear analysis. For nonlinear analysis, it is obviously required as a means for satisfying equilibrium (or the equations of motion in the case of dynamics). But even in linear analysis, internal forces are a convenient way of computing "loads" due to specified displacements, strains, stresses and/or temperatures, which are then transferred to the

right-hand side of the equations and act as external forces. In order to perform this function, internal forces must be computed as a function of *stresses*, rather than by simply multiplying the element stiffness matrix times the given displacement vector. By using the stresses, which are in input argument S, and already incorporate the effect of specified displacements, strains, etc., the internal force routine will then automatically include these contributions upon output.

Calling Sequence

call ESOFI (eltnum, ctls, defs, dofs, nodes, pars, x, d, s, store, fi, status)

Input Arguments

ELTNUM Element sequence number. Useful only for diagnostic purposes.

CTLS() List of element control parameters. Of particular importance is entry CTLS(pcNLG), which specifies the level of geometric nonlinearity to employ for internal-force computation.

DEFS() List of element definition parameters. The individual parameters which are relevant here will depend on the developer. Of particular importance, however, is DEFS(pdOPT), which is the element-type number (the numerical equivalent of ELTNAM in subroutine ESOD).

DOFS() List of potentially active element nodal degrees of freedom.

NODES() Element nodal resequencing map. (This array will ordinarily not be used in this subroutine, as all nodal arrays have already been reordered into the developer's internal node sequence upon entry.

PARS() Array of element-developer's research parameters. Useful for defining variations on basic element types before making them standard options specified by ELTNAM. These parameters are currently input to ES processors through the command macrosymbol ES_PARS.

X() Element nodal coordinate vectors; in initial configuration, element basis and according to the developer's node sequence.

D() Element nodal displacement vectors; in element basis and developer's node sequence. If CTLS(pcCORO) > 0, then displacements are relative to corotational element frame. Otherwise, they are relative to initial element frame.

S() Stresses (continuum or resultant, depending on element class) at element integration points, expressed in element-developer's stress bases (see Sections 3.3 and 3.4 for details and examples).

STORE() Element private storage array defined using subroutine ESOI.

Output Arguments

FI() Element internal force vector, expressed in element basis and ordered according to developer's node sequence (see Sections 3.3 and 3.4).

STATUS Subroutine return status ($\geq 0 \Rightarrow$OK).

Relevant Processor Commands

FORM FORCE/INTERNAL

FORM FORCE/RESIDUAL

3.2.4 Subroutine ES0FB: Element Body Force Vector

Entry point ES0FB should compute contributions to the element external force vector due to distributed body loads (forces per unit mass).

Calling Sequence

call ES0FB (eltnum, ctls, defs, dofs, nodes, pars, x, d, inert, loadb, store, fb, status)

Input Arguments

ELTNUM Element sequence number. Useful only for diagnostic purposes.

CTLS() List of element control parameters. Of particular importance is entry CTLS(pcNLG), which if nonzero indicates that the body loads are "live", *i.e.*, displacement-dependent. Currently only live pressure loads are implemented; see subroutine ES0FP.

DEFS() List of element definition parameters. The individual parameters which are relevant here will depend on the developer. Of particular importance, however, is DEFS(pdOPT), which is the element-type number (the numerical equivalent of ELTNAM in subroutine ES0D).

DOFS() List of potentially active element nodal degrees of freedom.

NODES() Element nodal resequencing map. This array will ordinarily not be necessary in this subroutine, as all nodal arrays have already been reordered into the developer's internal node sequence upon entry.

PARS() Array of element-developer's research parameters. Useful for defining variations on basic element types before making them standard options specified by ELTNAM. These parameters are currently input to ES processors through the command macrosymbol ES_PARS.

X() Element nodal coordinate vectors; in initial configuration, element basis and according to the developer's node sequence.

D() Element nodal displacement vectors; in element basis and developer's node sequence. If CTLS(pcCORO) > 0, then displacements are relative to corotational element frame. Otherwise, they are relative to initial element frame.

INERT() Inertia coefficients at element integration points; depend on element type. For example, for shell elements, INERT is a three by DEFS(pdNIP) array containing the results from the following three integrals through the shell thickness: $\int_z \rho dz, \int_z \rho z dz, \int_z \rho z^2 dz$, at each element integration point.

LOADB() Body loads. An array of DEFS(pdNDOF) by DEFS(pdNEN) numbers defining the body-force vectors (per unit mass) at element nodes, expressed in the element-cartesian coordinate system. The nodal vectors are ordered in the developer's node sequence.

STORE() Element private storage array defined using subroutine ESOI.

Output Arguments

FB() Element body force vector, expressed in element basis and ordered according to developer's node sequence (see Sections 3.3 and 3.4).

STATUS Subroutine return status ($\geq 0 \Rightarrow$OK).

Relevant Processor Commands

FORM FORCE/EXTERNAL

FORM FORCE/RESIDUAL

3.2.5 Subroutine ES0FL: Element Line Force Vector

Entry point ES0FL should compute contributions to the element external force vector due to distributed line (edge) forces/moments per unit length.

Calling Sequence

call ES0FL (eltnum, ctls, defs, dofs, nodes, pars, x, d, loadl, store, fl, status)

Input Arguments

ELTNUM Element sequence number. Useful only for diagnostic purposes.

CTLS() List of element control parameters. Of particular importance is entry CTLS(pcNLG), which if nonzero indicates that the line loads are "live", *i.e.*, displacement-dependent. Currently only live pressure loads are implemented; see subroutine ES0FP.

DEFS() List of element definition parameters. The individual parameters which are relevant here will depend on the developer. Of particular importance, however, is DEFS(pdOPT), which is the element-type number (the numerical equivalent of ELTNAM in subroutine ES0D).

DOFS() List of potentially active element nodal degrees of freedom.

NODES() Element nodal resequencing map. This array will ordinarily not be necessary in this subroutine, as all nodal arrays have already been reordered into the developer's internal node sequence upon entry.

PARS() Array of element-developer's research parameters. Useful for defining variations on basic element types before making them standard options specified by ELTNAM. These parameters are currently input to ES processors through the command macrosymbol ES_PARS.

X() Element nodal coordinate vectors; in initial configuration, element basis and according to the developer's node sequence.

D() Element nodal displacement vectors; in element basis and developer's node sequence. If CTLS(pcCORO) > 0, then displacements are relative to corotational element frame. Otherwise, they are relative to initial element frame.

LOADL() Line loads. An array of DEFS(pdNDOF) by DEFS(pdNNLT) numbers defining the line-load vectors (force or moment per unit length) at element nodes, for each element line (edge) independently, and expressed in the element-cartesian coordinate system. For 1-D elements, the nodal vectors are ordered in the developer's node sequence. For 2-D and 3-D elements, which have more than one line (edge), the lines, and nodes along each line, are ordered as depicted in Figure 2.1b.

STORE() Element private storage array defined using subroutine ESOI.

Output Arguments

FL() Element line force vector, expressed in element basis and ordered according to developer's node sequence (see Sections 3.3 and 3.4).

STATUS Subroutine return status ($\geq 0 \Rightarrow$OK).

Relevant Processor Commands

FORM FORCE/EXTERNAL

FORM FORCE/RESIDUAL

3.2.6 Subroutine ES0FP: Element Pressure Force Vector

Entry point ES0FP should compute contributions to the element external force vector due to pressure loads (normal forces per unit area).

Calling Sequence

call ES0FP (eltnum, ctls, defs, dofs, nodes, pars, x, d, loadp, store, fp, status)

Input Arguments

ELTNUM Element sequence number. Useful only for diagnostic purposes.

CTLS() List of element control parameters. Of particular importance is entry CTLS(pcNLG), which if nonzero indicates that the pressure loads are "live", (*i.e.*, displacement-dependent), so that they remain normal to the *deformed* surface.

DEFS() List of element definition parameters. The individual parameters which are relevant here will depend on the developer. Of particular importance, however, is DEFS(pdOPT), which is the element-type number (the numerical equivalent of ELTNAM in subroutine ES0D).

DOFS() List of potentially active element nodal degrees of freedom.

NODES() Element nodal resequencing map. This array will ordinarily not be necessary in this subroutine, as all nodal arrays have already been reordered into the developer's internal node sequence upon entry.

PARS() Array of element-developer's research parameters. Useful for defining variations on basic element types before making them standard options specified by ELTNAM. These parameters are currently input to ES processors through the command macrosymbol ES_PARS.

X() Element nodal coordinate vectors; in initial configuration, element basis and according to the developer's node sequence.

D() Element nodal displacement vectors; in element basis and developer's node sequence. Relevant only if CTLS(pcNLL) > 0 *and* CTLS(pcNLG) > 0. If CTLS(pcCORO) > 0, then displacements are relative to current (corotational) element frame. Otherwise, they are relative to the initial element frame.

LOADP() Pressure loads. An array of DEFS(pdNEN) numbers defining the pressures at element nodes. For 2-D elements, the nodal pressures are ordered in the developer's node sequence. For 3-D elements, which have more than one surface, the surfaces, and nodes within each surface, are ordered as depicted in Figure 2.1b.

STORE() Element private storage array defined using subroutine ESOI.

Output Arguments

FP() Element pressure force vector, expressed in element basis and ordered according to developer's node sequence (see Sections 3.3 and 3.4).

STATUS Subroutine return status ($\geq 0 \Rightarrow$OK).

Relevant Processor Commands

FORM FORCE/EXTERNAL

FORM FORCE/RESIDUAL

3.2.7 Subroutine ES0FS: Element Surface Force Vector

Entry point ES0FS should compute contributions to the element external force vector due to general surface forces/moments per unit area.

Calling Sequence

call ES0FS (eltnum, ctls, defs, dofs, nodes, pars, x, d, loads, store, fs, status)

Input Arguments

ELTNUM Element sequence number. Useful only for diagnostic purposes.

CTLS() List of element control parameters. Of particular importance is entry CTLS(pcNLG), which if nonzero indicates that the surface loads are "live", *i.e.*, displacement-dependent. Currently only live pressure loads are implemented; see subroutine ES0FP.

DEFS() List of element definition parameters. The individual parameters which are relevant here will depend on the developer. Of particular importance, however, is DEFS(pdOPT), which is the element-type number (the numerical equivalent of ELTNAM in subroutine ES0D).

DOFS() List of potentially active element nodal degrees of freedom.

NODES() Element nodal resequencing map. This array will ordinarily not be necessary in this subroutine, as all nodal arrays have already been reordered into the developer's internal node sequence upon entry.

PARS() Array of element-developer's research parameters. Useful for defining variations on basic element types before making them standard options specified by ELTNAM. These parameters are currently input to ES processors through the command macrosymbol ES_PARS.

X() Element nodal coordinate vectors; in initial configuration, element basis and according to the developer's node sequence.

D() Element nodal displacement vectors; in element basis and developer's node sequence. If CTLS(pdCORO) > 0, then displacements are relative to corotational element frame. Otherwise, they are relative to initial element frame.

LOADS() Surface loads. An array of DEFS(pdNDOF) by DEFS(pdNEN) numbers defining the traction vectors at element nodes, expressed in the element-cartesian coordinate system. For 2-D elements, the nodal vectors are ordered in the developer's node sequence. For 3-D elements, which have more than one surface, the surfaces, and nodes within each surface, are ordered as depicted in Figure 2.1b.

STORE() Element private storage array defined using subroutine ESOI.

Output Arguments

FS() Element surface force vector, expressed in element basis and ordered according to developer's node sequence (see Sections 3.3 and 3.4).

STATUS Subroutine return status ($\geq 0 \Rightarrow$OK).

Relevant Processor Commands

FORM FORCE/EXTERNAL

FORM FORCE/RESIDUAL

3.2.8 Subroutine ES0I: Element Initialization

Entry point ES0I is an optional mechanism for the developer to precompute selected element data at the beginning of an analysis (using the ES command INITIALIZE), and have it automatically stored-in and retrieved-from the database by the ES processor shell. For example, it may be convenient (or economical) for some elements to precompute basic shape/interpolation function arrays that do not vary with the solution. Note, however, that in some cases it may be even faster (in terms of overall turn-around time) to recompute such data in each of the other element kernel routines, rather than to have the ES shell perform the extra I/O operations associated with the present subroutine.

Calling Sequence

call ES0I (eltnum, ctls, defs, dofs, nodes, pars, x, store, status)

Input Arguments

ELTNUM Element sequence number. (Useful only for diagnostic purposes.)

CTLS() List of element control parameters. Of particular importance are entries CTLS(pcNL*), which indicate the types of nonlinearity to be included in subsequent calculations.

DEFS() List of element definition parameters. The individual parameters which are relevant here will depend on the developer. Of particular importance, however, is DEFS(pdOPT), which is the element-type number, and DEFS(pdSTOR), which is the developer-prescribed length of output array STORE.

DOFS() List of potentially active element nodal degrees of freedom.

NODES() Element node resequencing map. This mapping will ordinarily not be necessary in this subroutine, as all nodal arrays have already been reordered into the developer's internal sequence upon entry.

PARS() Array of element-developer's research parameters. Useful for defining variations on basic element types before making them standard options specified by ELTNAM. These parameters are currently input to ES processors through the command macrosymbol ES_PARS.

X() Element nodal coordinate vectors; in initial configuration, element basis and developer's node sequence.

Output Arguments

STORE() Element private storage array. The length of this array, *i.e.*, the number of entries, should have been defined in subroutine ESOD, and hence should be contained in DEFS(pdSTOR).

STATUS Subroutine return status ($\geq 0 \Rightarrow$OK).

Relevant Processor Commands

INITIALIZE

Remark 3.7 Presently, it is necessary for all developers to provide a version of subroutine ESOI, regardless of whether or not initialization data is to be stored. If no data is to be stored, then simply set DEFS(pdSTOR) = 0 in subroutine ESOD, and insert only a RETURN statement in subroutine ESOI. (Note: This requirement will probably be eliminated in the future.)

3.2.9 Subroutine ES0KG: Element Geometric Stiffness Matrix

Entry point ES0KG should compute the element geometric stiffness matrix in the element intrinsic coordinate basis. This function is important in both linear buckling-eigenvalue analysis and in nonlinear analysis (as a contribution to the tangent stiffness matrix). Inputs to this routine include stresses and displacements. By definition, the geometric stiffness is an explicit function of the stresses. However, the element displacement vector is provided as well for those elements that employ explicit geometric nonlinearity (in addition to the basic corotational operations built into the ES processor shell, as described in Chapter 4.)

Calling Sequence

call ES0KG (eltnum, ctls, defs, dofs, nodes, pars, x, d, s, store, kg, status)

Input Arguments

ELTNUM Element sequence number. Useful only for diagnostic purposes.

CTLS() List of element control parameters. Of particular importance is entry CTLS(pcNLG), which specifies the level of geometric nonlinearity to employ for stiffness computation — and hence determines whether or not the element displacements (input argument D) should be used.

DEFS() List of element definition parameters. The individual parameters which are relevant here will depend on the developer. Of particular importance, however, is DEFS(pdOPT), which is the element-type number (the numerical equivalent of ELTNAM).

DOFS() List of potentially active element nodal degrees of freedom.

NODES() Element node resequencing map. This mapping will ordinarily not be necessary in this subroutine, as all nodal arrays will already have been reordered into the developer's internal sequence upon entry.

PARS() Array of element-developer's research parameters. Useful for defining variations on basic element types before making them standard options specified by ELTNAM. These parameters are currently input to ES processors through the command macrosymbol ES_PARS.

X() Element nodal coordinate vectors; in initial configuration, element basis and developer's node sequence.

D() Element nodal displacement vectors; in element basis and developer's node sequence. If CTLS(pcCORO) $>$ 0, then displacements are relative to corotational element frame. Otherwise, they are relative to initial element frame. For linear analysis or low-order geometrically nonlinear analysis (CTLS(pcNLG) $<$ 2), displacements are typically ignored.

S() Stresses (or resultants, depending on element class) at element integration points, expressed in element stress bases.

STORE() Element private storage array defined using subroutine ESOI.

Output Arguments

KG() Element geometric stiffness matrix; stored in upper triangular form, and ordered according to the developer's node sequence. Components should be in terms of the element basis, which is the same basis as that used for the nodal coordinates (input argument X). All resorting and transformations required for eventual output to the database will be performed by the ES shell. See Sections 3.3 and 3.4 for more details and examples.

STATUS Subroutine return status ($\geq 0 \Rightarrow$OK).

Relevant Processor Commands

FORM STIFFNESS/GEOM

FORM STIFFNESS/TANG

3.2.10 Subroutine ES0KL: Element Load Stiffness

Entry point ES0KL should compute the element *load* stiffness matrix in the element intrinsic coordinate basis. This function is important for both (i) nonlinear analyses and (ii) linear eigenvalue (buckling or vibration) analyses, in which *displacement-dependent external loads* are present (*e.g.*, follower forces or hydrostatic pressure). While in the nonlinear regime, the inclusion of the load stiffness matrix primarily affects only the *rate* of convergence to the correct solution (*i.e.*, its absence may result in more iterations and/or smaller load steps); in the linear regime, omission of the load stiffness may result in a significant loss of accuracy — especially in the lowest buckling loads or vibration frequencies. Currently, the only type of load stiffness implemented is that due to live pressure loads.

Calling Sequence

call ES0KL (eltnum, ctls, defs, dofs, nodes, pars, x, d, loadp, store, kl, status)

Input Arguments

ELTNUM Element sequence number. Useful only for diagnostic purposes.

CTLS() List of element control parameters. Of particular importance is entry CTLS(pcNLG), which specifies the level of geometric nonlinearity to employ for stiffness computation — and hence determines whether or not the element displacements (input argument D) should be used.

DEFS() List of element definition parameters. The individual parameters which are relevant here will depend on the developer. Of particular importance, however, is DEFS(pdOPT), which is the element-type number (the numerical equivalent of ELTNAM).

DOFS() List of potentially active element nodal degrees of freedom.

NODES() Element node resequencing map. This mapping will ordinarily not be necessary in this subroutine, as all nodal arrays will already have been reordered into the developer's internal sequence upon entry.

PARS() Array of element-developer's research parameters. Useful for defining variations on basic element types before making them standard options specified by ELTNAM. These parameters are currently input to ES processors through the command macrosymbol ES_PARS.

X() Element nodal coordinate vectors; in initial configuration, element basis and developer's node sequence.

D() Element nodal displacement vectors; in element basis and developer's node sequence. If CTLS(pcCORO) $>$ 0, then displacements are relative to corotational element frame. Otherwise, they are relative to initial element frame. For linear analysis or low-order geometrically nonlinear analysis (CTLS(pcNLG) $<$ 2), displacements are typically ignored.

LOADP() Pressure loads. An array of DEFS(pdNEN) numbers defining the "live" pressures at element nodes. For 2-D elements, the nodal pressures are ordered in the developer's node sequence. For 3-D elements, which have more than one surface, the surfaces, and nodes within each surface, are ordered as depicted in Figure 2.1b.

STORE() Element private storage array defined using subroutine ESOI.

Output Arguments

KL() Element load stiffness matrix; stored in upper triangular form, and ordered according to the developer's node sequence. Components should be in terms of the element basis which is the same basis as that used for the nodal coordinates (input argument X). All resorting and transformations required for eventual output to the database will be performed by the ES shell. See Sections 3.3 and 3.4 for more details and examples.

STATUS Subroutine return status ($\geq 0 \Rightarrow$OK).

Relevant Processor Commands

FORM STIFFNESS/TANGENT

FORM STIFFNESS/GEOMETRIC

FORM STIFFNESS/LOAD

3.2.11 Subroutine ES0KM: Element Material Stiffness Matrix

Entry point ES0KM should compute the element *material* stiffness matrix in the element intrinsic coordinate basis. This function is important in all forms of analysis except for explicit transient dynamics. In linear analysis, it is assembled into the primary operator matrix; it assembles into one of the primary matrices in both buckling and vibration eigenvalue analysis, and it usually constitutes the dominant contribution to the tangent stiffness matrix in nonlinear analysis. Inputs to this routine include the material (integrated) constitutive matrix and the element displacement vector. By definition, the material stiffness is an explicit function of the constitutive coefficients. However, the element displacement vector is provided as well for those elements that employ explicit geometric nonlinearity. Such nonlinearity may not be required for large-rotation analyses if the built-in corotational option is invoked by the ES processor user (see Chapter 4).

For assumed-displacement type elements, the material stiffness matrix can usually be expressed in the form:

$$\mathbf{K}^{matl} = \int_{Vol} \mathbf{B}^T \mathbf{C} \mathbf{B} \, dV$$

where **B** is the incremental strain-displacement interpolation matrix, and **C** is the generalized constitutive matrix (corresponding to subroutine argument C). The volume integral is typically approximated by numerical integration, with **C** evaluated only at integration points. For beam or shell elements, the integrals may become one-dimensional (curve) or two-dimensional (surface) domains, respectively, with the **C** matrix representing a constitutive matrix that has been *preintegrated* over the cross-section, and corresponding to the tangent operator relating incremental stress and strain resultants.

Calling Sequence

call ES0KM (eltnum, ctls, defs, dofs, nodes, pars, x, d, c, store, km, status)

Input Arguments

ELTNUM Element sequence number. Useful only for diagnostic purposes.

CTLS() List of element control parameters. Of particular importance is entry CTLS(pcNLG), which specifies the level of geometric nonlinearity to employ for stiffness computation — and hence determines whether or not the element displacements (input argument D) are to be used.

DEFS() List of element definition parameters. The individual parameters which are relevant here will depend on the developer. Of particular importance, however, is DEFS(pdOPT), which is the element type number (the numerical equivalent of ELTNAM).

DOFS() List of potentially active element nodal degrees of freedom.

NODES() Element node resequencing map. This mapping will ordinarily not be necessary in this subroutine, as all nodal arrays have already been reordered into the developer's internal sequence upon entry.

PARS() Array of element-developer's research parameters. Useful for defining variations on basic element types before making them standard options specified by ELTNAM. These parameters are currently input to ES processors through the command macrosymbol ES_PARS.

X() Element nodal coordinate vectors; in initial configuration, element basis and developer's node sequence.

D() Element nodal displacement vectors; in element basis and developer's node sequence. If CTLS(pcCORO) > 0, then displacements are relative to corotational element frame. Otherwise, they are relative to initial element frame. For linear analysis or low-order geometrically nonlinear analysis (CTLS(pcNLG) < 2), displacements are typically ignored.

C() Constitutive matrices (continuum or resultant, depending on element class), at element integration points, expressed in element developer's stress bases. Individual constitutive matrices are stored as full, square (dimensioned NSTR by NSTR) matrices at each integration point, with

rows and columns corresponding to the stress and strain arrays, S and E, respectively. See Sections 3.3 and 3.4 for details and examples.

STORE() Element private storage array defined using subroutine ESOI.

Output Arguments

KM() Element material stiffness matrix; stored in upper triangular form, and ordered according to the developer's node sequence. Components should be in terms of the element intrinsic basis, which is the same basis as that used for the nodal coordinates (input argument X) and displacements (input argument D). All resorting and transformations required for eventual output to the database will be performed by the ES processor shell. See Sections 3.3 and 3.4 for details and examples.

STATUS Subroutine return status ($\geq 0 \Rightarrow$OK).

Relevant Processor Commands

FORM STIFFNESS/MATL

FORM STIFFNESS/TANG

3.2.12 Subroutine ES0MC: Element Consistent Mass Matrix

Entry point ES0MC should compute the element consistent mass matrix, employing the inertia tensor coefficients (argument INERT) as input.

Calling Sequence

call ES0MC (eltnum, ctls, defs, dofs, nodes, pars, x, d, inert, store, mc, status)

Input Arguments

ELTNUM Element sequence number. (Useful only for diagnostic purposes.)

CTLS() List of element control parameters. Of particular importance is entry CTLS(pcNLG), which specifies the level of geometric nonlinearity to employ for stiffness computation — and hence determines whether or not the element displacements (input argument D) are to be used.

DEFS() List of element definition parameters. The individual parameters which are relevant here will depend on the developer. Of particular importance, however, is DEFS(pdOPT), which is the element-type number (the numerical equivalent of ELTNAM).

DOFS() List of potentially active element nodal degrees of freedom.

NODES() Element node resequencing map. This mapping will ordinarily not be necessary in this subroutine, as all nodal arrays have already been reordered into the developer's internal sequence upon entry.

PARS() Array of element-developer's research parameters. Useful for defining variations on basic element types before making them standard options specified by ELTNAM. These parameters are currently input to ES processors through the command macrosymbol ES_PARS.

X() Element nodal coordinate vectors; in initial configuration, element basis and developer's node sequence.

D() Element nodal displacement vectors; in element basis and developer's node sequence. If CTLS(pcCORO) > 0, then displacements are relative to corotational element frame. Otherwise, they are relative to initial element frame. For linear analysis or low-order geometrically nonlinear analysis (CTLS(pcNLG) < 2), displacements may typically be ignored.

INERT() Inertia coefficients at element integration points; depend on element type. For example, for shell elements, INERT is a three by DEFS(pdNIP) array containing the results from the following three integrals through the shell thickness: $\int_z \rho dz, \int_z \rho z dz, \int_z \rho z^2 dz$, at each element integration point.

STORE() Element private storage array defined using subroutine ES0I.

Output Arguments

MC() Element consistent mass matrix; stored in upper triangular form, and ordered according to the developer's node sequence. Components should be in terms of the element intrinsic basis — which is the same basis as that used for the nodal coordinates (input argument X) and displacements (input argument D). All re-sorting and transformations required for eventual output to the database will be performed by the ES processor shell. See Sections 3.3 and 3.4 for details and examples.

STATUS Subroutine return status ($\geq 0 \Rightarrow$OK).

Relevant Processor Commands

FORM MASS/CONSISTENT

3.2.13 Subroutine ES0MD: Element Diagonal Mass Matrix

Entry point ES0MD should compute the element diagonal or lumped mass matrix, employing the inertia tensor coefficents (argument INERT) as input.

Calling Sequence

call ES0MD (eltnum, ctls, defs, dofs, nodes, pars, x, d, inert, store, md, status)

Input Arguments

ELTNUM Element sequence number. Useful only for diagnostic purposes.

CTLS() List of element control parameters. Of particular importance is entry CTLS(pcNLG), which specifies the level of geometric nonlinearity to employ for stiffness computation — and hence determines whether or not the element displacements (input argument D) are to be used.

DEFS() List of element definition parameters. The individual parameters which are relevant here will depend on the developer. Of particular importance, however, is DEFS(pdOPT), which is the element-type number (the numerical equivalent of ELTNAM).

DOFS() List of potentially active element nodal degrees of freedom.

NODES() Element node resequencing map. This mapping will ordinarily not be necessary in this subroutine, as all nodal arrays have already been reordered into the developer's internal sequence upon entry.

PARS() Array of element-developer's research parameters. Useful for defining variations on basic element types before making them standard options specified by ELTNAM. These parameters are currently input to ES processors through the command macrosymbol ES_PARS.

X() Element nodal coordinate vectors; in initial configuration, element basis and developer's node sequence.

D() Element nodal displacement vectors; in element basis and developer's node sequence. If CTLS(pcCORO) > 0, then displacements are relative to corotational element frame. Otherwise, they are relative to initial element frame. For linear analysis or low-order geometrically nonlinear analysis (CTLS(pcNLG) < 2), displacements are typically ignored.

INERT() Inertia coefficients at element integration points; depend on element type. For example, for shell elements, INERT is a three by DEFS(pdNIP) array containing the results from the following three integrals through the shell thickness: $\int_z \rho dz, \int_z \rho z dz, \int_z \rho z^2 dz$, at each element integration point.

STORE() Element private storage array defined using subroutine ESOI.

Output Arguments

MD() Element diagonal mass matrix; presently stored as a vector of length equal to the number of element equations (DEFS(pdNEE)). Components should be in terms of the element intrinsic basis, which is the same basis as that used for the nodal coordinates (input argument X) and displacements (input argument D). All resorting and transformations required for eventual output to the database will be performed by the ES processor shell. (See Sections 3.3 and 3.4 for details and examples.)

STATUS Subroutine return status ($\geq 0 \Rightarrow$OK).

Relevant Processor Commands

FORM MASS/DIAGONAL

3.2.14 Subroutine ES0N: Element Normal Vectors

Entry point ES0N should compute unit normal vectors — for plate/shell elements — at element nodes. It is required only if the developer has set DEFS(pdNORO) greater than zero in subroutine ES0D; in which case subroutine ES0N is invoked in response to ES processor command DEFINE FREEDOMS, to suppress extraneous normal-rotation ("drilling") freedoms for shell elements that do not support such freedoms.

Calling Sequence

call ES0N (eltnum, ctls, defs, dofs, nodes, pars, x, n, status)

Input Arguments

ELTNUM Element sequence number. (Useful only for diagnostic purposes.)

CTLS() List of element control parameters.

DEFS() List of element definition parameters. The individual parameters which are relevant here will depend on the developer. Of particular importance, however, is DEFS(pdOPT), which is the element-type number.

DOFS() List of potentially active element nodal degrees of freedom.

NODES() Element node resequencing map. This mapping will ordinarily not be necessary in this subroutine, as all nodal arrays have already been reordered into the developer's internal sequence upon entry.

PARS() Array of element-developer's research parameters. Useful for defining variations on basic element types before making them standard options specified by ELTNAM. These parameters are currently input to ES processors through the command macrosymbol ES_PARS.

X() Element nodal coordinate vectors; in initial configuration, element basis and developer's node sequence.

Output Arguments

N() Unit normal vectors at element nodes. Relevant for plate and shell elements only. The unit normal at a node should be defined as a unit vector normal to the element reference surface (tangent plane) at that node, defined in the developer's node sequence, and expressed in the intrinsic element coordinate basis. See Section 3.3 Glossary for more details.

STATUS Subroutine return status ($\geq 0 \Rightarrow$OK).

Relevant Processor Commands

DEFINE FREEDOMS

3.2.15 Subroutine ES0PD:

Entry point ES0PD may be used to postprocess nodal displacements. This feature is useful for superparametric elements and elements with *p*-extension.

Calling Sequence

call ES0PD (eltnum, ctls, defs, dofs, nodes, pars, x, d, status)

Input Arguments

ELTNUM Element sequence number. (Useful only for diagnostic purposes.)

CTLS() List of element control parameters. Of particular importance is entry CTLS(pcNLG), which specifies the level of geometric nonlinearity to employ for strain computation.

DEFS() List of element definition parameters. The individual parameters which are relevant here will depend on the developer. Of particular importance, however, is DEFS(pdOPT), which is the element type number (the numerical equivalent of ELTNAM).

DOFS() List of potentially active element nodal degrees of freedom.

NODES() Element node resequencing map. This mapping will ordinarily not be necessary in this subroutine, as all nodal arrays have already been reordered into the developer's internal sequence upon entry.

PARS() Array of element-developer's research parameters. Useful for defining variations on basic element types before making them standard options specified by ELTNAM. These parameters are currently input to ES processors through the command macrosymbol ES_PARS.

X() Element nodal coordinate vectors; in initial configuration, element basis and developer's node sequence.

D() Element nodal displacement vectors; in element basis and developer's node sequence. If CTLS(pcCORO) > 0, then displacements are relative to corotational element frame. Otherwise, they are relative to initial element frame.

Output Arguments

STATUS Subroutine return status ($\geq 0 \Rightarrow$OK).

Relevant Processor Commands

FORM STRAIN

FORM STRESS

FORM FORCE/INTERNAL

FORM STIFFNESS/GEOMETRIC

FORM STIFFNESS/MATERIAL (if materially nonlinear)

FORM STIFFNESS/GEOMETRIC

3.2.16 Subroutine ES0PS:

Entry point ES0PS may be used to postprocess element stresses such as the recovery of the transverse shearing stresses using the equilibrium equations rather than the constitutive equations.

Calling Sequence

call ES0PS (eltnum, ctls, defs, dofs, nodes, pars, x, d, store, c, e, s, status)

Input Arguments

ELTNUM Element sequence number. (Useful only for diagnostic purposes.)

CTLS() List of element control parameters. Of particular importance is entry CTLS(pcNLG), which specifies the level of geometric nonlinearity to employ for strain computation.

DEFS() List of element definition parameters. The individual parameters which are relevant here will depend on the developer. Of particular importance, however, is DEFS(pdOPT), which is the element type number (the numerical equivalent of ELTNAM).

DOFS() List of potentially active element nodal degrees of freedom.

NODES() Element node resequencing map. This mapping will ordinarily not be necessary in this subroutine, as all nodal arrays have already been reordered into the developer's internal sequence upon entry.

PARS() Array of element-developer's research parameters. Useful for defining variations on basic element types before making them standard options specified by ELTNAM. These parameters are currently input to ES processors through the command macrosymbol ES_PARS.

X() Element nodal coordinate vectors; in initial configuration, element basis and developer's node sequence.

D() Element nodal displacement vectors; in element basis and developer's node sequence. If CTLS(pcCORO) > 0, then displacements are relative to corotational element frame. Otherwise, they are relative to initial element frame.

STORE() Element private storage array defined using subroutine ESOI.

C() Constitutive matrices (pointwise, or integrated over the cross-section, depending on element class), at element integration points, expressed in element stress bases. Individual constitutive matrices are stored as full, square (NSTR by NSTR) matrices, with rows and columns corresponding to the stress and strain arrays, S and E, respectively. See Sections 3.3–3.4 and Chapter 5 for details.

Output Arguments

E() Strains (continuum or resultant, depending on element class) at element integration points, expressed in element stress bases. Relevant only if DEFS(pdCNS) > 0. Input if DEFS(pdCNS) equals 3; output if DEFS(pdCNS) equals 1 or 2.

S() Stresses (or resultants, depending on element class) at element integration points, expressed in element stress bases.

STATUS Subroutine return status ($\geq 0 \Rightarrow$OK).

Relevant Processor Commands

FORM STRAIN

FORM STRESS

3.2.17 Subroutine ES0S: Element Stresses

Entry point ES0S should compute stresses at element integration points — but only for elements that either do not have the ability to compute strains (*e.g.*, assumed stress hybrid elements) or for element developer's who wish to perform their own constitutive calculations (see argument DEFS(pdCNS) in Section 3.3 and also in Chapter 5 for details). This routine is not required for elements that employ the standard constitutive interface, and hence provide strains using subroutine ES0E.

Calling Sequence

call ES0S (eltnum, ctls, defs, dofs, nodes, pars, x, d, store, c, e, s, status)

Input Arguments

ELTNUM Element sequence number. (Useful only for diagnostic purposes.)

CTLS() List of element control parameters. Of particular importance is entry CTLS(pcNLG), which specifies the level of geometric nonlinearity to employ for strain computation.

DEFS() List of element definition parameters. The individual parameters which are relevant here will depend on the developer. Of particular importance, however, is DEFS(pdOPT), which is the element type number (the numerical equivalent of ELTNAM).

DOFS() List of potentially active element nodal degrees of freedom.

NODES() Element node resequencing map. This mapping will ordinarily not be necessary in this subroutine, as all nodal arrays have already been reordered into the developer's internal sequence upon entry.

PARS() Array of element-developer's research parameters. Useful for defining variations on basic element types before making them standard options specified by ELTNAM. These parameters are currently input to ES processors through the command macrosymbol ES_PARS.

X() Element nodal coordinate vectors; in initial configuration, element basis and developer's node sequence.

D() Element nodal displacement vectors; in element basis and developer's node sequence. If CTLS(pcCORO) > 0, then displacements are relative to corotational element frame. Otherwise, they are relative to initial element frame.

STORE() Element private storage array defined using subroutine ESOI.

C() Constitutive matrices (pointwise, or integrated over the cross-section, depending on element class), at element integration points, expressed in element stress bases. Individual constitutive matrices are stored as full, square (NSTR by NSTR) matrices, with rows and columns corresponding to the stress and strain arrays, S and E, respectively. See Sections 3.3 through 3.4 and Chapter 5 for details.

Output Arguments

E() Strains (continuum or resultant, depending on element class) at element integration points, expressed in element stress bases. Relevant only if DEFS(pdCNS) > 0. Input if DEFS(pdCNS) = 3; output if DEFS(pdCNS) = 1 or 2.

S() Stresses (or resultants, depending on element class) at element integration points, expressed in element stress bases.

STATUS Subroutine return status ($\geq 0 \Rightarrow$OK).

Relevant Processor Commands

FORM STRAIN

FORM STRESS

FORM FORCE/INTERNAL

FORM STIFFNESS/GEOMETRIC

FORM STIFFNESS/MATERIAL (if materially nonlinear)

FORM STIFFNESS/GEOMETRIC

3.2.18 Subroutine ES0T: Element Transformations

Entry point ES0T should compute orthogonal transformation matrices relating the element-developer's stress basis at each element integration point to the element intrinsic basis (see Section 3.4 for examples of these coordinate bases). It is required for problems involving nonisotropic materials, in which strains, stresses and constitutive matrices may have to be transformed between the element stress basis and some other basis (*e.g.*, the material basis, the global basis). It may also be required even for isotropic materials, for post-processing stresses/strains in cases where the element bases are not aligned with directions that are of interest to the user. The developer is only required to relate the element stress bases to the element intrinsic basis which is the fixed basis in which the element nodal coordinates are provided as input. Subsequent transformations to other coordinate bases are handled automatically by the ES processor shell.

Calling Sequence

call ES0T (eltnum, ctls, defs, dofs, nodes, pars, x, store, tel, status)

Input Arguments

ELTNUM Element sequence number. Useful only for diagnostic purposes.

CTLS() List of element control parameters.

DEFS() List of element definition parameters. The individual parameters which are relevant here will depend on the developer. Of particular importance, however, is DEFS(pdOPT), which is the element type number, and DEFS(pdSTOR), which is the developer-prescribed length of output array STORE.

DOFS() List of potentially active element nodal degrees of freedom.

NODES() Element node resequencing map. This mapping will ordinarily not be necessary in this subroutine, as all nodal arrays have already been reordered into the developer's internal sequence upon entry.

PARS() Array of element-developer's research parameters. Useful for defining variations on basic element types before making them standard options specified by ELTNAM. These parameters are currently input to ES processors through the command macrosymbol ES_PARS.

X() Element nodal coordinate vectors; in initial configuration, element basis and developer's node sequence.

STORE() Element private storage array defined using subroutine ESOI.

Output Arguments

TEL() Transformations from element stress bases at integration points to element intrinsic basis. Dimensioned as (3,3,NIP), where NIP is the number of integration points per element, so that TEL(i,j,p) is defined as the dot product of the ith element intrinsic basis vector with the jth element stress basis vector at element integration point p. In other words, at a given integration point, the columns of TEL(3,3) are simply the element stress basis vectors: $\hat{\mathbf{e}}_1^s, \hat{\mathbf{e}}_2^s, \hat{\mathbf{e}}_3^s$, expressed in the element intrinsic basis (*i.e.*, the same basis as that used for the components of the element nodal coordinates: input argument X).

STATUS Subroutine return status ($\geq 0 \Rightarrow$OK).

3.2.19 Subroutine ES0XP:

Entry point ES0XP should compute extrapolation coefficients that allow quantities, such as stresses, evaluated at the quadrature points of the element to be extrapolated to other points within the element (*e.g.*, to the element nodes).

Calling Sequence

call **ES0XP** (eltnum, ctls, defs, dofs, nodes, pars, x, store, extrap, status)

Input Arguments

ELTNUM Element sequence number. Useful only for diagnostic purposes.

CTLS() List of element control parameters. Of particular importance is entry CTLS(pcNLG), which specifies the level of geometric nonlinearity to employ for strain computation.

DEFS() List of element definition parameters. The individual parameters which are relevant here will depend on the developer. Of particular importance, however, is DEFS(pdOPT), which is the element type number (the numerical equivalent of ELTNAM).

DOFS() List of potentially active element nodal degrees of freedom.

NODES() Element node resequencing map. This mapping will ordinarily not be necessary in this subroutine, as all nodal arrays have already been reordered into the developer's internal sequence upon entry.

PARS() Array of element-developer's research parameters. Useful for defining variations on basic element types before making them standard options specified by ELTNAM. These parameters are currently input to ES processors through the command macrosymbol ES_PARS.

X() Element nodal coordinate vectors; in initial configuration, element basis and developer's node sequence.

STORE() Element private storage array defined using subroutine ES0I.

Output Arguments

EXTRAP() Extrapolation coefficients dimensioned as MAXNIP by MAXNEN.

STATUS Subroutine return status ($\geq 0 \Rightarrow$OK).

Relevant Processor Commands

FORM STRAIN

FORM STRESS

3.3 Glossary of Standard ES Kernel Arguments

The following pages contain the formal definitions for each of the argument variables appearing in the standard kernel entry points. The arguments are arranged alphabetically, and include specification of the argument type and length. Also included in this Glossary are definitions for the various parameters used to dimension arguments that happen to be arrays. At the end of the Glossary is a table that shows the correspondences between various kernel arguments and processor macrosymbols, so that the developer is better able to trace the data flow from the user to the processor for purposes of software verification.

KERNEL ARGUMENT GLOSSARY

Argument	*Dimension*	*Type*	*Definition*
C	(NSTR,NSTR,NIP)	F	Constitutive matrices at element integration points. C(i, j, p) is the constitutive coefficient relating the jth incremental strain component to the ith incremental stress component at element integration point p.
CTLS	(20)	I	List of element control parameters.
CTLS(pcCORO)	(1)	I	Corotation switch; employed by ES processor shell for automatic treatment of geometric nonlinearity due to large rotations. Relevant only if CTLS(pcNLG) > 0. (See Chapter 4 for an explanation of these options.) 0 ⇒ Off: corotational operations will be skipped. 1 ⇒ Low-Order Option: basic corotational transformations will be employed to enable large rotations. 2 ⇒ High-Order Option: a more accurate (and expensive) treatment of large rotations and consistent linearization than option 1 will be employed.
CTLS(pcNLG)	(1)	I	Geometric nonlinearity switch. 0 ⇒ Off: problem is geometrically linear (small displacements/rotations). 1 ⇒ Low-Order Option: problem is geometrically nonlinear, but developer should use linear element strain-displacement relations. Meaningful only if CTLS(pcCORO) > 0, so that large rotations are handled automatically by the corotational algorithm.

KERNEL ARGUMENT GLOSSARY (continued)

Argument	*Dimension*	*Type*	*Definition*
CTLS(pcNLG)	(1)	I	(continued) 2 ⇒ High-Order Option: problem is geometrically nonlinear and developer should use *nonlinear* element strain-displacement relations. May be used in conjunction with CTLS(pcCORO) > 0 to obtain higher-order accuracy for beam and shell elements that employ moderate-rotation strain-displacement relations. 3 ⇒ same as 2 plus finite strains are expected; hence, deformation gradients must be provided using subroutine ES0DG.
CTLS(pcNLL)	(1)	I	Load nonlinearity switch; 1 ⇒ element loads are displacement dependent; 0 ⇒ element loads are not displacement dependent.
CTLS(pcNLM)	(1)	I	Material nonlinearity switch; 1 ⇒ element is materially nonlinear; 0 ⇒ element materially linear.
CTLS(pcSREF)	(1)	I	Stress reference frame switch; > 0 ⇒ stress transformation required (nonisotropic material); = 0 ⇒ no stress transformation required (isotropic material).
DEFS	(20)	I	List of intrinsic element definition parameters.

KERNEL ARGUMENT GLOSSARY (continued)

Argument	*Dimension*	*Type*	*Definition*
DEFS(pdC)	(1)	I	Continuity of interelement displacement field; *e.g.*, 0 ⇒ C^0 (displacement continuity only) 1 ⇒ C^1 (displacement and slope continuity) Currently used to indicate (i) whether transverse shear strains are intrinsic to beam/shell element constitutive behavior, and (ii) how to process high-order corotational option (CTLS(pcCORO)=2).
DEFS(pdCLAS)	(1)	I	Element class. Currently valid classes: idBEAM; idSHEL, idSOLI, idWILD (See Section 3.4 for examples.) idBEAM ⇒ Beam element; "stress" components correspond to: {N_x (axial force), M_z, M_y (bending moments), M_x (torsional moment) }, plus, if DEFS(pdNSTR) = 6, {Q_y, Q_z} (transverse shear forces). idSHEL ⇒ Shell element; "stress" components correspond to: {N_x, N_y, N_{xy}} (membrane force resultants), {M_x, M_y, M_{xy}} (bending moment resultants), plus, if DEFS(pdNSTR) = 8, {Q_x, Q_y} (transverse-shear force resultants). idSOLI ⇒ Solid element; stress components = {$\sigma_x, \sigma_y, \sigma_z, \sigma_{yz}, \sigma_{zx}, \sigma_{xy}$}. idWILD ⇒ Nonstandard element class. *– currently not implemented –* Note: Standard element classes can be degenerated into special subclasses using the *stress sequence* argument: STRSEQ. For example, rod elements are a subclass of beam elements; plate elements are a subclass of shell elements; and plane stress elements are a subclass of solid elements.

KERNEL ARGUMENT GLOSSARY (continued)

Argument	*Dimension*	*Type*	*Definition*
DEFS(pdCNS)	(1)	I	Constitutive interface option. 0 ⇒element developer will supply strain routine (ES0E) only; stress and constitutive matrix functions will be performed using the built-in ES constitutive utilities. (Note: Current constitutive utilities employ linear constitutive matrices generated by processor LAU. This will be replaced by a standard interface to the generic constitutive processor — see Section 5.4.) 1 ⇒element developer will supply combined stress/strain routine only (ES0S); constitutive matrix functions will be performed using the built-in ES constitutive utilities. (This option is intended primarily for *assumed stress* hybrid elements. 2 ⇒same as option 1, except developer will supply a constitutive matrix routine as well (ES0C). 3 ⇒element developer will supply strain (ES0E), stress (ES0S) and constitutive-matrix (ES0C) routines. This option is for developers who prefer to supply their own constitutive utilities, even though their elements are of the assumed displacement or strain variety.
DEFS(pdDIM)	(1)	I	Element intrinsic spatial dimensionality; 1 if DEFS(pdCLAS) = idBEAM 2 if DEFS(pdCLAS) = idSHEL 2 or 3 if DEFS(pdCLAS) = idSOLI

KERNEL ARGUMENT GLOSSARY (continued)

Argument	*Dimension*	*Type*	*Definition*
DEFS(pdNDOF)	(1)	I	Number of freedoms per element node. Currently valid options:
			2 ⇒ 2-D solid elements (u, v)
			3 ⇒ 3-D solid elements (u, v, w)
			6 ⇒ beam, plate or shell elements $(u, v, w, \theta_x, \theta_y, \theta_z)$
DEFS(pdNEE)	(1)	I	Number of element equations; = DEFS(pdNEN) × DEFS(pdNDOF).
DEFS(pdNEN)	(1)	I	Number of element nodes.
DEFS(pdNIP)	(1)	I	Number of element integration points; *i.e.*, points at which stresses (or stress resultants) are stored.
DEFS(pdNSTR)	(1)	I	Number of stress components per integration point. Currently valid options:
			8 for DEFS(pdCLAS)=idSHEL and DEFS(pdC) = 0 (or 1);
			6 for DEFS(pdCLAS)=idSOLI;
			6 for DEFS(pdCLAS)=idSHEL and DEFS(pdC)=1;
			6 for DEFS(pdCLAS)=idBEAM and DEFS(pdC)=0 (or 1);
			4 for DEFS(pdCLAS)=idBEAM and DEFS(pdC) = 1;
			3 for DEFS(pdCLAS)=idSOLI and DEFS(pdDIM) = 2

KERNEL ARGUMENT GLOSSARY (continued)

Argument	*Dimension*	*Type*	*Definition*
DEFS(pdNORO)	(1)	I	Element normal-rotation parameter. Relevant only for automatic degree of freedom suppression of plate/shell elements. Indicates minimum angle (in degrees) between shell element normal vector and any computational basis vector at each element node below which the corresponding rotational degrees of freedom should be suppressed if no other elements are attached. 0 ⇒Element has normal-rotation ("drilling") stiffness; do not automatically suppress any rotational degrees of freedom. >0 ⇒Element does not have normal-rotation ("drilling") stiffness; assume normal rotational stiffness exists about any computational axis that makes an angle of at least DEFS(pdNORO) degrees with the element normal vector at an element node.
DEFS(pdOPT)	(1)	I	Element-type option number. It is the numerical equivalent of ELTNAM and is provided so that the developer need only decode the character string in ELTNAM once in subroutine ES0D.
DEFS(pdPARS)	(1)	I	Number of parameters in PARS array.

KERNEL ARGUMENT GLOSSARY (continued)

Argument	*Dimension*	*Type*	*Definition*
DEFS(pdPROJ)	(1)	I	Rigid-body *projection* option. May be used to automatically remove (most of) the spurious energy generated by some elements during infinitesimal rigid-body motion. This is performed by operating on the element stiffness and force arrays with a projection matrix (or *projector*). The projector, and its derivative, can have a beneficial effect on element accuracy in both linear and geometrically nonlinear regimes. 0 ⇒Off: no projection. 1 ⇒Low-order Option: basic projection. 2 ⇒High-order Option: basic projection plus a correction to the geometric stiffness. NOTE: To switch the projector on as a function of processor macrosymbol ES_PROJ, the developer should set: DEFS(pdPROJ) = CTLS(pcPROJ).
DEFS(pdSHAP)	(1)	I	Shape of element surface used to define coordinate triad; currently = idTRIA (triangle) or idQUAD (quadrilateral). Note that for 3-D solid elements, idTRIA would be used for wedge or pyramid-type elements, whose triangular surface is used to define the element $x_e - y_e$ plane; while idQUAD would be used for wedge, pyramid or "brick" elements for which a quadrilateral surface is used to define the element $x_e - y_e$ plane. The definition of these axes is determined by the element node sequence (NODES).
DEFS(pdSTOR)	(1)	I	Number of variables to be stored/retrieved for the element developer in array STORE.

KERNEL ARGUMENT GLOSSARY (continued)

Argument	*Dimension*	*Type*	*Definition*
DEFS(pdTGE)	(1)	I	Element coordinate basis option; affects convention used to define the transformation from the element basis to the global basis (T_{ge}). 0 or 1 ⇒ Define x_e axis using corner nodes 1 and 2 (see Fig. 3.1). 2 ⇒ Define y_e axis using corner nodes 1 and 4 (see Fig. 3.1).
DEFS(pdTWIS)	(1)	I	Sign of twisting curvature for shell elements. +1 ⇒twist based on continuum definition of shear strain -1 ⇒twist based on the negative of the continuum definition (Note: The default convention for constitutive matrices output from processor LAU corresponds to the -1 option. Hence processor ES must compensate for this reversal if DEFS(pdTWIS) = +1.)
DEFS(pdES*)	(1)	I	Implementation status of routine ES*. 0 ⇒Not Yet Implemented 1 ⇒Implemented
DEFS(pdESD)	(1)	I	Implementation status of routine ES0D.
DEFS(pdESE)	(1)	I	Implementation status of routine ES0E.
DEFS(pdESFI)	(1)	I	Implementation status of routine ES0FI.
DEFS(pdESFB)	(1)	I	Implementation status of routine ES0FB.
DEFS(pdESFS)	(1)	I	Implementation status of routine ES0FS.
DEFS(pdESI)	(1)	I	Implementation status of routine ES0I.
DEFS(pdESKG)	(1)	I	Implementation status of routine ES0KG.
DEFS(pdESKL)	(1)	I	Implementation status of routine ES0KL.
DEFS(pdESKM)	(1)	I	Implementation status of routine ES0KM.

KERNEL ARGUMENT GLOSSARY (continued)

Argument	*Dimension*	*Type*	*Definition*
DEFS(pdESMC)	(1)	I	Implementation status of routine ES0MC.
DEFS(pdESMD)	(1)	I	Implementation status of routine ES0MD.
DEFS(pdESPD)	(1)	I	Implementation status of routine ES0PD.
DEFS(pdESS)	(1)	I	Implementation status of routine ES0S.
DEFS(pdEST)	(1)	I	Implementation status of routine ES0T.
DEFS(pdESN)	(1)	I	Implementation status of routine ES0N.

NOTE: The above DEFS(pdES*) parameters have not yet been implemented in the ES processor shell.

KERNEL ARGUMENT GLOSSARY (continued)

Argument	*Dimension*	*Type*	*Definition*
D	(NDOF,NEN)	F	Element nodal displacement vectors. D(k,a) is the displacement corresponding to the kth degree of freedom at element node a; expressed in the element local basis. If CTLS(pcNLC) > 0, D is relative to the current corotated element basis, with rigid-body motion subtracted; otherwise, D is relative to the initial element basis.
DOFS	(NDOF,NEN)	I	Element degree of freedom pattern at element nodes. Set DOFS(i,a) = 1 if the ith standard nodal degree of freedom at element node a is supported by the element. The standard nodal degree of freedom sequence is: $u, v, w, \theta_x, \theta_y, \theta_z$ — all expressed in the element local basis: $\{x_e, y_e, z_e\}$. For most elements, DOFS is a table full of ones. However, zeroes may be useful for elements that do not have the same number of degrees of freedom at every node; or those with superfluous (*e.g.*, geometric) nodes. Note: This table is eventually intended to allow an arbitrary sequence of degrees of freeom at each node; currently only subsets of the standard sequence are allowed.
E	(NSTR,NIP)	F	Strains at element integration points. E(i,p) corresponds the ith strain component at element integration point p; expressed in the integration-point local (stress) bases. The order and nature of the strain components depend on the element class (as defined in DEFS(pdCLAS)). Engineering as opposed to tensor shear strain components are assumed.

KERNEL ARGUMENT GLOSSARY (continued)

Argument	*Dimension*	*Type*	*Definition*
ELTNAM	(1)	C	Element type name; input to element definition subroutine (ESOD) only, as a branch point on which to define element parameters (*e.g.*, DEFS, DOFS, NODES and PARS).
ELTNUM	(1)	I	Element number. May be used by kernel routines for diagnostic purposes in case of an error.
FB	(NEE)	F	Element body-force vector. Represent consistent "nodal" forces due to external body loads such as gravity. FB(p) corresponds to the pth element body-force component; expressed in the element intrinsic basis (see Section 3.4). For standard element types, the element equations (1 through NEE) are arranged nodally, as if the FORTRAN array were dimensioned (NDOF,NEN). In this case, FB(i, a) is the force component corresponding to the ith degree of freedom at element node a (according to the *developer's* node sequence).
FI	(NEE)	F	Element internal-force vector. Represent "nodal" forces due to internal stress distribution. FI(p) corresponds to the pth element internal-force component; expressed in the element intrinsic basis (see Section 3.4). For standard element types, the element equations (1 through NEE) are arranged nodally, as if the FORTRAN array were dimensioned (NDOF,NEN). In this case, FI(i, a) is the force component corresponding to the ith degree of freedom at element node a (according to the *developer's* node sequence).

KERNEL ARGUMENT GLOSSARY (continued)

Argument	*Dimension*	*Type*	*Definition*
FL	(NEE)	F	Element line-force vector. Represents consistent nodal forces due to distributed line loads (LOADL). Stored in the same way as FB, FI, etc.
FP	(NEE)	F	Element pressure-force vector. Represents consistent nodal forces due to distributed pressure loads (LOADP). Stored in the same way as FB, FI, etc.
FS	(NEE)	F	Element surface-force vector. Represent consistent nodal forces due to distributed surface loads (LOADS). Stored in the same way as FB, FI, etc.
id*	(1)	I	Parameters used to define DEFS(pdCLAS); *e.g.*, idBEAM, idSHEL, idSOLI. The specific integer values for these parameters (which are of little relevance to the element developer) are defined in "INCLUDE" block ES0PTR.INC.
INERT	(NINERT,NIP)	F	Coefficients of integrated inertia tensor evaluated at element integration points, for computing element mass matrix. Depends on element type. For shell elements, NINERT = 3, and INERT(1,i) = $\int_z \rho dz$, INERT(2,i) = $\int_z \rho z dz$, and INERT(3,i) = $\int_z \rho z^2 dz$ — each at element integration point i, where ρ is the density (mass per unit volume), and the integral is over the shell thickness (z is the shell thickness coordinate). For beam elements, INERT contains the 6×6 integrated cross-section inertia matrix (see dataset PROP.BTAB.1.101) at each integration point. For solid elements, INERT simply contains the density (mass per unit volume) at each integration point.

KERNEL ARGUMENT GLOSSARY (continued)

Argument	*Dimension*	*Type*	*Definition*
KG	(NUT)	F	Element geometric stiffness matrix (upper triangular portion). The pth logical row and qth logical column of this matrix is the stiffness coefficient relating the element qth incremental displacement component to the element pth incremental force component; expressed in the element intrinsic basis. The element equations (1 through NEE) are arranged in correspondence with the element force and displacement vectors, but only the upper triangle of the matrix is represented (due to symmetry). Thus, this array should contain stiffness components in the order $K_{11}, K_{12}, K_{22}, K_{13}, K_{23}, K_{33}, \ldots, K_{nee,nee}$.
KM	(NUT)	F	Element material stiffness matrix. Stored in the same format as KG (see above).
KL	(NUT)	F	Element load stiffness matrix. Stored in the same format as KG (see above).
LOADB	(NDOF,NEN)	F	Element body load vectors at nodes.
LOADL	(NDOF,NNLT)	F	Element line load vectors at nodes.
LOADP	(NNST)	F	Element pressures at nodes.
LOADS	(NDOF,NNST)	F	Element surface load (traction) vectors at node.
MC	(NUT)	F	Element consistent mass matrix. The pth logical row and qth logical column of this matrix is the mass coefficient relating the element qth acceleration component to the element pth inertial force component; expressed in the element intrinsic basis. Stored in the same format as KG (see above).

KERNEL ARGUMENT GLOSSARY (continued)

Argument	*Dimension*	*Type*	*Definition*
MD	(NEE)	F	Element diagonal or lumped mass matrix MD(p) corresponds to the mass coefficient relating the element pth acceleration component to the element pth (inertial) force component; expressed in the element local basis.
N	(3,NEN)	F	Unit normal vectors at element nodes; expressed in element (x_e, y_e, z_e) basis. Relevant only for plate/shell elements.
NDOF	(1)	I	Number of DOFS per element node. (Same as DEFS(pdNDOF))
NEE	(1)	I	Number of element equations. (Same as DEFS(pdNEE))
NEN	(1)	I	Number of element nodes. (Same as DEFS(pdNEN))
NIP	(1)	I	Number of element integration points. (Same as DEFS(pdNIP))
NNLT	(1)	I	Total number of nodes on all element lines; nodes are counted independently on each element line. For example, for a 4-node shell element, NNLT = 8 (2 nodes per line times 4 lines per element). See Figure 2.1b for element line/node ordering conventions.
NNST	(1)	I	Total number of nodes on all element surfaces; nodes are counted independently on each element surface. For example, for an 8-node solid element, NNST = 24 (4 nodes per surface times 6 surfaces per element). See Figure 2.1b for element surface/node ordering conventions.

KERNEL ARGUMENT GLOSSARY (continued)

Argument	*Dimension*	*Type*	*Definition*
NODES	(NEN)	I	Element node sequence mapping. NODES(a) is the standard (external) element node number corresponding to element developer's (internal) node a. See Section 3.4 for examples of standard element node sequences.
NSTR	(1)	I	Number of stress components per element integration points. (Same as DEFS(pdNSTR))
NUT	(1)	I	Number of entries in the upper triangle of an element matrix (NUT = NEE×(NEE + 1)/2).
PARS	(nPARS)	F	Array of element research parameters.
pc*	(1)	I	Pointers to CTLS array. Specific values of these indices (which are of little relevance to the element developer) are defined in "INCLUDE" block ES0PTR.INC.
pd*	(1)	I	Pointers to DEFS array. Specific values of these indices (which are of little relevance to the element developer) are defined in "INCLUDE" block ES0PTR.INC.
S	(NSTR,NIP)	F	Stresses at element integration points. S(i,p) = the ith stress component at element integration point p; expressed in the element's local integration point bases. The order and nature of the stress components depend on the element class (see DEFS(pdCLAS)).
STATUS	(1)	I	Subroutine return status. STATUS $\geq 0 \Rightarrow$ OK; STATUS $< 0 \Rightarrow$ error condition.

KERNEL ARGUMENT GLOSSARY (concluded)

Argument	*Dimension*	*Type*	*Definition*
STORE	(nSTORE)	F	developer's private element storage array. This array is computed during initialization by subroutine ES0I automatically stored in the database and later retrieved by the processor shell (ES0) for use by other kernel routines.
STRSEQ	(NSTR)	I	Stress component sequence mapping. This array allows elements to be implemented with arbitrary subsets of the stress components defined for the standard element classes (see DEFS(pdCLAS)). For example, for a rod element, the developer would set DEFS(pdCLAS) = idBEAM, NSTR=1 and STRSEQ(1)=1. – *CURRENTLY NOT IMPLEMENTED* –
T	(3,3,NIP)	F	Transformations from element stress bases at integration points to element intrinsic basis.
X	(3,NEN)	F	Element nodal coordinates. X(i, a) corresponds to ith spatial coordinate at element node a; expressed in the element local basis and corresponding to the undeformed configuration.

3.4 Examples of Specific Element Types

In this section, guidelines for the preparation of the standard kernel routines/arguments, described in the preceding sections, are given for specific element types. The element types considered here are not meant to be exhaustive, but merely to provide examples with which most element developers can identify. As mentioned in the Introduction to this Chapter, implementation of other element types not mentioned here may require one of the following approaches: (i) straightforward extension of the examples presented for standard elements; (ii) use of the "wild" element approach for nonstandard elements; (iii) application of pressure on the ES processor shell *architect* to extend the standard framework; or (iv) development of a special-purpose ES processor shell by the element developer.

Recall that the terms *standard* and *nonstandard* (or "wild") elements are defined as follows. *Standard* elements are fully recognized by the ES processor shell, so that various standard operations, such as:

- Coordinate transformations
- Corotational updates (for geometric nonlinearity)
- Constitutive processing (for linear and nonlinear materials)
- Automatic Freedom Suppression

may be performed *automatically* — by the shell. On the other hand, *wild* elements are treated as "black boxes", that must perform all such operations on their own — *i.e.*, at the kernel level. Thus, the developer is confronted with the usual trade-off between generality and simplicity: If the element fits into one of the *standard* molds, then the developer's responsibility will be less. If the element is too exotic ("wild") for the standard conventions, then the developer must do it all.

The following subsections are organized according to the number of intrinsic element spatial dimensions (1-D, 2-D, 3-D). Elements that do not fit within the framework of the standard 1-D, 2-D and 3-D elements discussed in Sections 3.4.1 through 3.4.3, may be implemented as "wild" elements, as described in Section 3.4.4.

Remark 3.1 The coordinate systems and node-numbering conventions described below for *standard* (*i.e.*, non-"wild") elements should not require the developer to make major changes to existing element computational routines. For example, the developer may employ his/her own internal node-numbering conventions, as long as he/she provides a *mapping* between the standard (user) numbering scheme and the developer numbering scheme. This mapping is the NODES array (Section 3.3), which is defined using kernel routine ES0D (Section 3.2) and is used by the ES processor shell to automatically resort data as it passes to and from the standard kernel routines. Furthermore, the intrinsic element coordinate system (x_e, y_e, z_e), which is defined uniquely by the element shape and the user node numbers, may be viewed by the element developer as a *global* coordinate system. The main use of this coordinate system is by the built-in corotational utilities, which are described in Chapter 4. (For beam elements, the intrinsic element coordinate basis is also used to orient cross-section properties.) The only connection the developer has to make with the intrinsic element coordinate system is through the kernel subroutine ES0T which must provide the transformation between this frame and the element *stress* coordinate system, which the developer is free to choose arbitrarily (see Chapter 5).

3.4.1 1-D Elements

One-dimensional elements are defined as those elements whose nodes lie on a common *curve*, such that only one intrinsic coordinate is required to locate intermediate (*e.g.*, integration) points. Examples of 1-D elements are *rod* elements (*i.e.*, straight or curved elements that may be generally oriented in three-dimensional space, but whose only deformation mode is stretching† and *beam* elements (*i.e.*, straight or curved elements generally oriented in three-dimensional space, which can deform by stretching, bending, and twisting).

3.4.1.1 User Node Numbers and Coordinate Systems for 1-D Elements

Figure 3.1a shows how the *end-point* node numbers, as defined by the user, determine the intrinsic element coordinate system (x_e, y_e, z_e) for standard 1-D elements. Figure 3.1b shows complete *user* node-numbering sequences for various low-order and high-order 1-D elements.

Note that the straight line connecting user nodes 1 and 2 always defines the x_e axis — regardless of whether the element is curved or not, and independent of how many nodes the element has. The y_e and z_e axes are defined by the user, using a number of options described in the Testbed User's Manual (under processor TAB; the MREF command). The important convention for the developer to note is that *the z_e axis is expected to lie in the plane of the element cross-section at all points along the element length — at least in the initial configuration.* If the developer uses the z_e axis and the cross-section normal vector to complete a triad defining the element stress coordinate system at element integration points, this can avoid extraneous transformations of stress, strain and constitutive resultants by the ES processor shell.

† When used for geometrically nonlinear analysis, these elements are often called *cable* elements.

3.4.1.2 Developer Node Numbers for 1-D elements (NODES Array)

The standard (user) node numbering sequence for 1-D elements is intended to facilitate model generation (for users). To facilitate element development, the developer may use an internal node-numbering sequence which is different from the standard one. The correspondence between the two sequences should be conveyed by the developer through the NODES array (see Section 3.3), which must be defined in standard kernel routine ES*D.

The NODES array gives the user node index as a function of the developer node index. The following examples show some common uses of NODES for 1-D elements.

Example 1.

If the element developer's node order is identical to the standard order (shown in Figure 3.1b), then the developer should set

$$\text{NODES}(a) = a \quad (a = 1, \ldots, \text{NEN})$$

in kernel routine ES0D; where NEN = 2 or 3 or 4, etc.

Example 2.

If the developer's node sequence for a 3-node 1-D element is as shown in Figure 3.2, then the developer should set:

$$\text{NODES}(1) = 1$$

$$\text{NODES}(2) = 3$$

$$\text{NODES}(3) = 2$$

in kernel routine ES0D.

3.4.1.3 Rod Elements

Distinct conventions for rod elements, *i.e.*, 1-D elements with axial stiffness only, are *currently not implemented* as a standard ES processor option. However, the developer is free to implement rod elements as a special case of *beam elements* (Section 3.4.1.2), wherein certain partitions of the integrated constitutive matrix are ignored, and all rotational degrees of freedom are suppressed.

3.4.1.4 Beam Elements

Beam elements are defined as 1-D elements (*i.e.*, with one intrinsic coordinate variable), which are embedded in three-dimensional space, such that all six standard nodal degrees of freedom (three translations and three rotations) are potentially active, and for which stress *resultants* (stresses integrated over the cross-section) are employed instead of continuum stress components. This general definition contains rod elements, cable elements and both straight and curved beam elements as special cases.

Remark 3.2 While resultant stresses and strains are considered the intrinsic element parameters, continuum stress and strain components may be used in the constitutive relations — especially for nonlinear materials (see Chapter 5).

3.4.1.4.1 Beam-Element Intrinsic Parameters (the DEFS Array)

The following settings should be used when defining the DEFS argument array (by kernel subroutine ES0D) for beam elements:

$$
\begin{aligned}
\text{DEFS(pdCLAS)} &= \text{idBEAM} \\
\text{DEFS(pdDIM)} &= 1 \\
\text{DEFS(pdNDOF)} &= 6 \\
\text{DEFS(pdC)} &= \begin{cases} 1 \text{ for } C^1 \text{ elements} \\ 0 \text{ for } C^0 \text{ elements} \end{cases} \\
\text{DEFS(pdNSTR)} &= \begin{cases} 4 \text{ for } C^1 \text{ elements} \\ 6 \text{ for } C^0, \text{ or } C^1, \text{ elements} \end{cases} \\
\text{DEFS(pdPROJ)} &= \text{CTLS(pcPROJ)} \quad \text{(optional)}
\end{aligned}
$$

where the array indices, pd*, and the constant, idBEAM, are defined in all of the standard kernel routines using built-in parameter statements in the include block ES0PTR.INC.

Other DEFS array entries should be set according to the descriptions given in the standard kernel argument glossary (Section 3.3).

3.4.1.4.2 Beam-Element Nodal Freedoms (the DOFS Array)

The DOFS array, which defines (using kernel subroutine ES0D) the potentially active nodal freedoms for an element type, would typically be set by the developer as follows for a beam element:

$$\text{DOFS(6,NEN)} = \begin{array}{c} \\ u \\ v \\ w \\ \theta_x \\ \theta_y \\ \theta_z \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & \ldots & \text{NEN} \end{array} \\ \begin{pmatrix} 1 & 1 & \ldots & 1 \\ 1 & 1 & \ldots & 1 \\ 1 & 1 & \ldots & 1 \\ 1 & 1 & \ldots & 1 \\ 1 & 1 & \ldots & 1 \\ 1 & 1 & \ldots & 1 \end{pmatrix} \end{array}$$

where the ones in the table indicate that all translational freedoms (u, v, w) and rotational freedoms $(\theta_x, \theta_y, \theta_z)$ are potentially active at every element node. The directions of these degrees of freedom are interpreted as corresponding to the computational nodal basis used to express the *assembled* system of equations. Thus, it would probably not be meaningful to selectively turn off individual translation or rotation components at specific nodes using the DOFS array. However, it would make sense to permanently suppress *all* translational freedoms or *all* rotational freedoms at selected nodes, or to suppress *both* translational and rotational freedoms, *e.g.*, at nodes which are used only for *geometric* definition.

3.4.1.4.3 Beam-Element Strains (The E Array)

For beam elements, the standard convention for the strain array, E(NSTR,NIP), is as follows. For C^1 beam elements (DEFS(pdC)=1 and DEFS(pdNSTR)=4):

$$\text{E(4,NIP)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & \ldots & \text{NIP} \end{array} \\ \begin{pmatrix} \bar{\epsilon}_x(\xi_1) & \bar{\epsilon}_x(\xi_2) & \ldots & \bar{\epsilon}_x(\xi_{\text{NIP}}) \\ \kappa_z(\xi_1) & \kappa_z(\xi_2) & \ldots & \kappa_z(\xi_{\text{NIP}}) \\ \kappa_y(\xi_1) & \kappa_y(\xi_2) & \ldots & \kappa_y(\xi_{\text{NIP}}) \\ \alpha(\xi_1) & \alpha(\xi_2) & \ldots & \alpha(\xi_{\text{NIP}}) \end{pmatrix} \end{array}$$

where ξ_p is the element axial coordinate (curve parameter) at integration point p (NIP is the total number of element integration points); and where $\bar{\epsilon}_x$ is the *axial* strain, κ_z and κ_y are the *bending* strains (changes of curvature in the x-y plane and x-z plane, respectively), and α is the *torsional strain* (*i.e.*, unit twist).

Similarly, for C^0 beam elements (DEFS(pdC)=0 and Defs(pdNSTR)=6):

$$\text{E(6,NIP)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & \ldots & \text{NIP} \end{array} \\ \begin{pmatrix} \bar{\epsilon}_x(\xi_1) & \bar{\epsilon}_x(\xi_2) & \ldots & \bar{\epsilon}_x(\xi_{\text{NIP}}) \\ \kappa_z(\xi_1) & \kappa_z(\xi_2) & \ldots & \kappa_z(\xi_{\text{NIP}}) \\ \kappa_y(\xi_1) & \kappa_y(\xi_2) & \ldots & \kappa_y(\xi_{\text{NIP}}) \\ \alpha(\xi_1) & \alpha(\xi_2) & \ldots & \alpha(\xi_{\text{NIP}}) \\ \gamma_y(\xi_1) & \gamma_y(\xi_2) & \ldots & \gamma_y(\xi_{\text{NIP}}) \\ \gamma_z(\xi_1) & \gamma_z(\xi_2) & \ldots & \gamma_z(\xi_{\text{NIP}}) \end{pmatrix} \end{array}$$

where γ_y and γ_z are some "average" measures of the *transverse-shear* strains in the $x-y$ and $x-z$ planes, respectively, in addition to the axial, bending, and torsional strains introduced above.

The strain directions (*i.e.*, the orientation of the cross-section coordinates, y, z — x is assumed to be normal to the cross-section) and the meaning of the axial (curve) coordinate,

ξ, are left to the developer's discretion, and should be documented by the developer in an appropriate section of the Testbed User's Manual (see ref. 4).

Remark 3.3 The transverse-shear strain resultants, γ_y and γ_z, are interpreted as deriving from *engineering* shear strains, *i.e.*, they are a factor of two larger than the corresponding tensorial shear strains (ϵ_{xy} and ϵ_{xz}).

Remark 3.4 The type of strain-displacement relations used (*i.e.*, linear or nonlinear) are also up to the developer. If the built-in corotational option is selected by the user, then — as long as the strains remain small — it is not really necessary for the developer to provide nonlinear strain-displacement relations, regardless of the magnitude of the rotations. However, it may be desirable to include a nonlinear option for improved accuracy (see kernel input argument CTLS(pcCORO) and Chapter 4).

3.4.1.4.4 Beam-Element Stresses (the S Array)

For beam elements, the standard convention for the stress array, S(NSTR,NIP), is a set of stress *resultants*, defined by integrating the continuum stresses over the beam cross-section, which are (incrementally) work-conjugate to the resultant strain measures described above. Thus, the number of stress resultants depends on whether or not the element exhibits transverse-shear strains (*i.e.*, C^0 versus C^1 displacement compatibility).

Thus, for C^1 beam elements (DEFS(pdC)=1 and DEFS(pdNSTR) = 4):

$$\text{S(4,NIP)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & \ldots & \text{NIP} \end{array} \\ \begin{pmatrix} N_x(\xi_1) & N_x(\xi_2) & \ldots & N_x(\xi_{\text{NIP}}) \\ M_z(\xi_1) & M_z(\xi_2) & \ldots & M_z(\xi_{\text{NIP}}) \\ M_y(\xi_1) & M_y(\xi_2) & \ldots & M_y(\xi_{\text{NIP}}) \\ T(\xi_1) & T(\xi_2) & \ldots & T(\xi_{\text{NIP}}) \end{pmatrix} \end{array}$$

where ξ_p is the beam-element axial coordinate (*i.e.*, curve parameter) at integration point p (NIP is the total number of integration points); and where the *axial force*, N_x, the two *bending moments*, M_z and M_y, and the *torsional moment*, T, might be defined as follows:

$$\begin{Bmatrix} N_x \\ M_z \\ M_y \\ T \end{Bmatrix} = \int_A \begin{Bmatrix} \sigma_{xx} \\ \sigma_{xx}\, y \\ \sigma_{xx}\, z \\ \sigma_{xz}\, y - \sigma_{xy}\, z \end{Bmatrix} dA$$

Similarly, for C^0 beam elements (DEFS(pdC)=0 and DEFS(pdNSTR) = 6), the first four stress resultants would be defined exactly as for C^1 elements, with the additional two components being the *transverse-shear* stress resultants, *i.e.*,

$$\text{S(6,NIP)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & \ldots & \text{NIP} \end{array} \\ \begin{pmatrix} N_x(\xi_1) & N_x(\xi_2) & \ldots & N_x(\xi_{\text{NIP}}) \\ M_z(\xi_1) & M_z(\xi_2) & \ldots & M_z(\xi_{\text{NIP}}) \\ M_y(\xi_1) & M_y(\xi_2) & \ldots & M_y(\xi_{\text{NIP}}) \\ T(\xi_1) & T(\xi_2) & \ldots & T(\xi_{\text{NIP}}) \\ Q_y(\xi_1) & Q_y(\xi_2) & \ldots & Q_y(\xi_{\text{NIP}}) \\ Q_z(\xi_1) & Q_z(\xi_2) & \ldots & Q_z(\xi_{\text{NIP}}) \end{pmatrix} \end{array}$$

where the *transverse-shear forces* might be defined as follows:

$$\begin{Bmatrix} Q_y \\ Q_z \end{Bmatrix} = \int_A \begin{Bmatrix} \sigma_{xy} \\ \sigma_{xz} \end{Bmatrix} dA$$

The stress directions (*i.e.*, the orientation of the cross-section coordinates, y, z — x is assumed to be normal to the cross-section) and the meaning of the axial (curve) coordinate, ξ, are left to the developer's discretion, and should be documented by the developer in an appropriate section of the Testbed User's Manual (see ref. 4).

Remark 3.5 The type of stress components assumed (*e.g.*, nominal or true) depends on the constitutive algorithm (see Chapter 5). For problems involving small strains, the components can be interpreted as true (*i.e.*, Cauchy) stresses.

Remark 3.6 It is sometimes desirable to allow C^1 beam elements to compute transverse-shear stress resultants. However, since C^1 elements, by definition, lack transverse-shear *strains*, the transverse-shear resultants must be recovered using a separate equilibrium post-processing step. For convenience, such quantities are stored in a separate array, SX, which is referred to only in kernel routine ESOPS (*currently not implemented*).

3.4.1.4.5 Beam-Element Constitutive Matrix (The C Array)

For beam elements, the standard convention for the constitutive array, C(NSTR,NSTR,NIP), is consistent with the definitions for the stress (S) and strain (E) arrays given above. The tangent constitutive matrix relating incremental strains to incremental stresses at integration point p is thus defined for C^1 beam elements by:

$$\mathrm{C}(4,4,p) = \begin{array}{c} \\ N_x \\ M_z \\ M_y \\ T \end{array} \begin{array}{c} \begin{array}{cccc} \overline{\epsilon}_x & \kappa_z & \kappa_y & \alpha \end{array} \\ \begin{pmatrix} \overline{C}_{11}(\xi_p) & \overline{C}_{12}(\xi_p) & \overline{C}_{13}(\xi_p) & \overline{C}_{14}(\xi_p) \\ & \overline{C}_{22}(\xi_p) & \overline{C}_{23}(\xi_p) & \overline{C}_{24}(\xi_p) \\ & & \overline{C}_{33}(\xi_p) & \overline{C}_{34}(\xi_p) \\ sym. & & & \overline{C}_{44}(\xi_p) \end{pmatrix} \end{array}$$

and for C^0 beam elements by:

$$\mathrm{C}(6,6,p) = \begin{array}{c} \\ N_x \\ M_z \\ M_y \\ T \\ Q_y \\ Q_z \end{array} \begin{array}{c} \begin{array}{cccccc} \overline{\epsilon}_x & \kappa_z & \kappa_y & \alpha & \gamma_y & \gamma_z \end{array} \\ \begin{pmatrix} \overline{C}_{11}(\xi_p) & \overline{C}_{12}(\xi_p) & \overline{C}_{13}(\xi_p) & \overline{C}_{14}(\xi_p) & \overline{C}_{15}(\xi_p) & \overline{C}_{16}(\xi_p) \\ & \overline{C}_{22}(\xi_p) & \overline{C}_{23}(\xi_p) & \overline{C}_{24}(\xi_p) & \overline{C}_{25}(\xi_p) & \overline{C}_{26}(\xi_p) \\ & & \overline{C}_{33}(\xi_p) & \overline{C}_{34}(\xi_p) & \overline{C}_{35}(\xi_p) & \overline{C}_{36}(\xi_p) \\ & & & \overline{C}_{44}(\xi_p) & \overline{C}_{45}(\xi_p) & \overline{C}_{46}(\xi_p) \\ & & & & \overline{C}_{55}(\xi_p) & \overline{C}_{56}(\xi_p) \\ sym. & & & & & \overline{C}_{66}(\xi_p) \end{pmatrix} \end{array}$$

where the $\overline{C}_{ij}$ are integrated (over-the-cross-section) constitutive coefficients, which depend on both the material and section properties. The actual definitions will depend on the constitutive processing option selected (see Chapter 5).

As with the stress and strain arrays (S and E), the interpretation of the directions used for the components and the choice of the cross-section and axial coordinates are left up to the element developer. The transformations between the intrinsic material coordinate system (*i.e.*, the system used for constitutive calculations) and the element developer's stress/strain system (*i.e.*, the system used for the C, S and E arrays) are performed automatically by the ES processor shell, as described in Chapter 5.

3.4.1.4.6 Beam-Element Displacement Vector (The D Array)

The displacement vector for a beam element can be viewed as a singly dimensioned array, of length NEE = NDOF×NEN = 6×NEN (see glossary in Section 3.3), or as a two-dimensional array, defined as follows:

$$\text{D(6,NEN)} = \begin{bmatrix} u_e^1 & u_e^2 & \dots & u_e^{\text{NEN}} \\ v_e^1 & v_e^2 & \dots & v_e^{\text{NEN}} \\ w_e^1 & w_e^2 & \dots & w_e^{\text{NEN}} \\ \theta_{xe}^1 & \theta_{xe}^2 & \dots & \theta_{xe}^{\text{NEN}} \\ \theta_{ye}^1 & \theta_{ye}^2 & \dots & \theta_{ye}^{\text{NEN}} \\ \theta_{ze}^1 & \theta_{ze}^2 & \dots & \theta_{ze}^{\text{NEN}} \end{bmatrix}$$

where u_e^a, v_e^a, w_e^a are the translations in the x_e, y_e, z_e directions, respectively, at element node a; and $\theta_{xe}^a, \theta_{ye}^a, \theta_{ze}^a$ are rotations about the corresponding axes. (See definition of *intrinsic element basis* in Figure 3.1b)

The node sequence here refers to the *developer's* order — not to the user's. It is assumed for plane-elements, that the global problem is strictly two-dimensional, so that all tranformations between the element intrinsic basis (x_e, y_e, z_e) and the global Cartesian basis (x_g, y_g, z_g) — or an alternate computational basis at nodes (x_c, y_c, z_c) are such that the respective z axes are parallel.

Remark 3.7 The D array is input to various standard kernel routines, such as ES0E (strains), ES0FI (internal forces) and ES0KM, ES0KG (material/geometric stiffness). The components are always resolved into the element intrinsic basis, and for geometrically nonlinear analysis, with corotation turned on, the displacements are relative to the moving element frame and hence should be "moderately small" (see Chapter 4).

3.4.1.4.7 Beam-Element Force Vectors (The F Arrays)

The force vector for a beam element — either internal (FI) or external (FB or FS) — should be output from the standard kernel routines (ES0FI, ES0FB or ES0FS) as an array that is in one-to-one correspondence with the element displacement vector. Thus:

$$\mathrm{F(6,NEN)} = \begin{bmatrix} F^1_{x_e} & F^2_{x_e} & \dots & F^{\mathrm{NEN}}_{x_e} \\ F^1_{y_e} & F^2_{y_e} & \dots & F^{\mathrm{NEN}}_{y_e} \\ F^1_{z_e} & F^2_{z_e} & \dots & F^{\mathrm{NEN}}_{z_e} \\ M^1_{x_e} & M^2_{x_e} & \dots & M^{\mathrm{NEN}}_{x_e} \\ M^1_{y_e} & M^2_{y_e} & \dots & M^{\mathrm{NEN}}_{y_e} \\ M^1_{z_e} & M^2_{z_e} & \dots & M^{\mathrm{NEN}}_{z_e} \end{bmatrix}$$

where $F^a_{x_e}, F^a_{y_e}, F^a_{z_e}$ are the forces in the x_e, y_e and z_e directions at element node a; and $M^a_{x_e}, M^a_{y_e}$ and $M^a_{z_e}$ are the moments about the corresponding axes. The developer's node sequence is implied.

3.4.1.4.8 Beam-Element Stiffness Matrices (The K Arrays)

The stiffness matrix for a beam-element — either material (KM), geometric (KG) or load (KL) — should be output from the standard kernel routines (ES0KM, ES0KG or ES0KL) in the following *upper-triangular* form:

$$\mathrm{K}\left((NEE^2+NEE)/2\right) = \begin{array}{c} \\ F^1_{x_e} \\ F^1_{y_e} \\ F^1_{z_e} \\ M^1_{x_e} \\ M^1_{y_e} \\ M^1_{z_e} \\ \vdots \\ M^{\mathrm{NEN}}_{z_e} \end{array} \begin{array}{c} \begin{array}{cccccccc} u^1_e & v^1_e & w^1_e & \theta^1_{x_e} & \theta^1_{y_e} & \theta^1_{z_e} & \dots & \theta^{\mathrm{NEN}}_{z_e} \end{array} \\ \begin{pmatrix} K_{11} & K_{12} & K_{13} & K_{14} & K_{15} & K_{16} & \dots & K_{1,\mathrm{NEE}} \\ & K_{22} & K_{23} & K_{24} & K_{25} & K_{26} & \dots & K_{2,\mathrm{NEE}} \\ & & K_{33} & K_{34} & K_{35} & K_{36} & \dots & K_{3,\mathrm{NEE}} \\ & & & K_{44} & K_{45} & K_{46} & \dots & K_{4,\mathrm{NEE}} \\ & & & & K_{55} & K_{56} & & K_{5,\mathrm{NEE}} \\ & & & & & K_{66} & \dots & K_{6,\mathrm{NEE}} \\ & & & & & & \ddots & \vdots \\ & & & & & & \dots & K_{\mathrm{NEE,NEE}} \end{pmatrix} \end{array}$$

where NEE = 6 × NEN and the numbers are stored as one partial column after another (see glossary in Section 3.3), and the node numbers correspond to the developer's node sequence, not the user's.

3.4.2 2-D Elements

Two-dimensional elements are defined as those elements whose nodes lie on a common *surface*, such that only two intrinsic coordinates are required to locate intermediate (*e.g.*, integration) points. Examples of 2-D elements are plane elements (*e.g.*, plane-stress, plane-strain or axi-symmetric solid elements), plate elements (*i.e.*, elements that lie in a plane, but whose displacement field is transverse to that plane), and general shell elements (*i.e.*, elements that lie on a surface in three-dimensional space and whose deformation can involve both translations and rotations in all three directions) — see Figure 3.3.

3.4.2.1 User Node Numbers and Coordinate Systems for 2-D Elements

Figure 3.3a shows how the *corner node* numbers, as defined by the user, determine the *intrinsic* coordinate system (x_e, y_e, z_e) for standard 2-D elements. The node-numbering sequence, which always starts with the corner nodes, for various standard 2-D elements is shown in Figure 3.3b.

Note that all standard 2-D elements are assumed to have either a triangular or a quadrilateral planform (corresponding to kernel argument DEFS(pdSHAP) = idTRIA or idQUAD), with either the first 3 or the first 4 nodes, respectively, used to define the element intrinsic coordinate system (x_e, y_e, z_e). As shown in Figure 3.3, the first 3 or 4 nodes must be the *corner* nodes, with a *counter-clockwise* convention used to define the positive z_e direction.

For both triangles and quadrilaterals, there are two options for orienting the x_e and y_e axes relative to the element edges. The origin is always at node 1. Then if the developer sets DEFS(pdTGE) = 0 or 1 (default), the x_e axis is selected as the line connecting nodes 1 and 2. Alternatively, if DEFS(pdTGE) = 2, the y_e axis is selected as the line connecting nodes 1 and 4 (for quads), or 3 (for triangles). Figure 3.3c illustrates these options.

Note that for *quadrilateral* elements, the four corner nodes do not necessarily lie in the same plane. Hence the z_e axis is constructed as the normal to an "average" plane, given by the cross-product of the two diagonals: $\overline{1\text{-}3}$ and $\overline{2\text{-}4}$. Then, depending on the option selected by DEFS(pdTGE), either the x_e or y_e axis becomes the *projection* of the appropriate edge onto this average plane. This frame adjustment is used to improve element accuracy, especially in conjunction with the corotation/projection operators described in Chapter 4.

3.4.2.2 Developer Node Numbers for 2-D Elements (NODES Array).

The standard (user) node numbering sequence for 2-D elements is intended to facilitate model generation (for users). To facilitate element development, the developer may use an internal node-numbering sequence which is different from the standard one. The correspondence between these two sequences must be conveyed by the developer through the NODES array (see Section 3.3), which the developer is asked to define in standard kernel routine ES0D.†

The NODES array should give the user node index as a function of the developer node index. The following examples show some of the most common uses of the NODES array for 2-D elements:

Example 3.

If the element developer's node order is identical to the standard order (shown in Figure 3.3b), then the developer should set

$$\text{NODES}(a) = a \quad (a = 1, \ldots, \text{NEN})$$

in kernel routine ES0D; where NEN = 3 or 4 or 9 or 16, etc.

† NOTE: There is no equivalent to the NODES array for renumbering edges, or nodes within edges, in the definition of element line loads (input argument LOADL in subroutine ES0FL). There the developer must use the standard numbering conventions illustrated in Figure 2.1b.

Example 4.

If the developer's node sequence for a 4-node quadrilateral element is as shown in Figure 3.4a, then the developer should set:

NODES(1) = 1

NODES(2) = 2

NODES(3) = 4

NODES(4) = 3

in kernel routine ES0D.

Example 5.

If the developer's node sequence for a 9-node quadrilateral element is as shown in Figure 3.4b, then the developer should set:

NODES(1) = 1

NODES(2) = 3

NODES(3) = 9

NODES(4) = 7

NODES(5) = 2

NODES(6) = 6

NODES(7) = 8

NODES(8) = 4

NODES(9) = 5

in kernel routine ES0D.

3.4.2.3 Plane (Stress/Strain) Elements

Plane elements are defined as 2-D elements that are embedded in a two-dimensional solid continuum. Typically, such elements have (i) only two translational degrees of freedom at each node, and (ii) only three out of the original six continuum strain/stress components contributing to the element strain energy. Other useful possiblities (not described here) include axisymmetric solid elements (which have four strain/stress components), and generalized plane strain elements (which have all six stress/strain components and all three translational degrees of freedom per node).

3.4.2.3.1 Plane-Element Intrinsic Parameters (the DEFS Array)

The following settings should be used when defining the DEFS argument array (using kernel subroutine ES0D) for plane elements:

```
DEFS(pdCLAS) = idSOLI
DEFS(pdDIM)  = 2
DEFS(pdNDOF) = 2
DEFS(pdNSTR) = 3
```

where the array indices, pd*, and the constant, idSOLI, are defined in all of the standard kernel routines using built-in parameter statements (see include block ES0PTR.INC).

Other DEFS array entries should be set according to the descriptions given in the standard kernel argument glossary (Section 3.3).

3.4.2.3.2 Plane-Element Nodal Freedoms (the DOFS Array)

The DOFS array, which defines (using kernel subroutine ESOD) the potentially active nodal degrees of freedoms for an element type, would typically be set by the developer as follows for a plane element:

$$\text{DOFS(2,NEN)} = \begin{array}{c} \\ u \\ v \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & \ldots & \text{NEN} \end{array} \\ \begin{pmatrix} 1 & 1 & \ldots & 1 \\ 1 & 1 & \ldots & 1 \end{pmatrix} \end{array}$$

where the ones in the table indicate that all in-plane translational displacement freedoms (u and v) are potentially active at every element node. The directions of these u and v degrees of freedom are interpreted as corresponding to the computational nodal basis used to express the *assembled* system of equations. Thus, it would probably not be meaningful to turn off individual u or v element nodal degrees of freedom using the DOFS array since the element developer has no control over which computational basis will be selected it is selected by the user as a global attribute. However, it may be useful to permanently suppress *both* u and v at selected nodes, *e.g.*, if some of the nodes are used only for *geometric* definition.

3.4.2.3.3 Plane-Element Strains (The E Array)

For plane stress/strain elements, the standard convention for the strain array, E(3,NIP), is as follows:

$$\text{E(3,NIP)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & \ldots & \text{NIP} \end{array} \\ \begin{pmatrix} \epsilon_{xx}(\xi_1) & \epsilon_{xx}(\xi_2) & \ldots & \epsilon_{xx}(\xi_{\text{NIP}}) \\ \epsilon_{yy}(\xi_1) & \epsilon_{yy}(\xi_2) & \ldots & \epsilon_{yy}(\xi_{\text{NIP}}) \\ \epsilon_{xy}(\xi_1) & \epsilon_{xy}(\xi_2) & \ldots & \epsilon_{xy}(\xi_{\text{NIP}}) \end{pmatrix} \end{array}$$

where $\xi_p = \{\xi_p, \eta_p\}$ are the element surface coordinates at integration point p, and x, y refer to the element stress basis: x_s, y_s; both of which are defined by the developer.

Remark 3.8 The "normal" strain component, ϵ_{zz}, is assumed to be zero for the plane strain case, and is computed using the constitutive relations for the plane stress case.

3.4.2.3.4 Plane-Element Stresses (the S Array)

For plane stress/strain elements, the standard convention for the stress array, S(3,NIP), is as follows:

$$\mathrm{S(3,NIP)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & \ldots & \mathrm{NIP} \end{array} \\ \begin{pmatrix} \sigma_{xx}(\xi_1) & \sigma_{xx}(\xi_2) & \ldots & \sigma_{xx}(\xi_{\mathrm{NIP}}) \\ \sigma_{yy}(\xi_1) & \sigma_{yy}(\xi_2) & \ldots & \sigma_{yy}(\xi_{\mathrm{NIP}}) \\ \sigma_{xy}(\xi_1) & \sigma_{xy}(\xi_2) & \ldots & \sigma_{xy}(\xi_{\mathrm{NIP}}) \end{pmatrix} \end{array}$$

where $\boldsymbol{\xi}_p = \{\xi_p, \eta_p\}$ are the element surface coordinates at integration point p, and x, y refer to the element stress basis: x_s, y_s, both of which are defined by the developer.

Remark 3.9 The "normal" stress component, σ_{zz} is assumed to be zero for the plane stress case, and is computed using the constitutive relations for the plane strain case.

Remark 3.10 If the developer has selected the constitutive option: DEFS(pdCNS) equals 0 or 2, the element stress array will be computed automatically using the ES processor shell (see Chapter 5).

3.4.2.3.5 Plane-Element Constitutive Matrix (The C Array)

For plane stress/strain elements, the standard convention for the constitutive array, C(3,3,NIP), is as follows. The tangent constitutive matrix relating incremental strains to incremental stresses at integration point p is defined by

$$\mathrm{C(3,3,NIP)} = \begin{array}{c} \\ \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{xy} \end{array} \begin{array}{c} \begin{array}{ccc} \epsilon_{xx} & \epsilon_{yy} & \epsilon_{xy} \end{array} \\ \begin{pmatrix} C_{11}(\boldsymbol{\xi}_p) & C_{12}(\boldsymbol{\xi}_p) & C_{13}(\boldsymbol{\xi}_p) \\ & C_{22}(\boldsymbol{\xi}_p) & C_{23}(\boldsymbol{\xi}_p) \\ sym. & & C_{33}(\boldsymbol{\xi}_p) \end{pmatrix} \end{array}$$

where the values of the coefficients C_{ij} depend on both the material properties and whether a plane-stress or plane-strain hypothesis is employed. As with the stress and strain arrays (S and E), the interpretation of the directions used for the components and the choice of surface coordinates are left up to the element developer.

Remark 3.11 The transformations between the material coordinate system and the element developer's local stress/strain system are performed automatically by the ES processor shell (see Chapter 5).

Remark 3.12 If the developer has selected the constitutive option: DEFS(pdCNS) equals 0 or 1, the constitutive matrix will be generated automatically by the ES processor shell.

3.4.2.3.6 Plane-Element Displacement Vector (The D Array)

The displacement vector for a plane element can be viewed as a singly dimensioned array, of length NEE = NDOF×NEN = 2×NEN (see glossary in Section 3.3), or as a two-dimensional array, defined as follows:

$$\text{D(2,NEN)} = \begin{bmatrix} u_e^1 & u_e^2 & \dots & u_e^{\text{NEN}} \\ v_e^1 & v_e^2 & \dots & v_e^{\text{NEN}} \end{bmatrix}$$

where u_e^a and v_e^a are the translations in the x_e and y_e directions (see Figure 3.3a) at element node a. The node sequence here refers to the *developer's* order — not the user's (Figure 3.3b). It is assumed for plane elements (with DEFS(pdNDOF) = 2) that the global problem is strictly two-dimensional, so that all tranformations between the element intrinsic basis (x_e, y_e, z_e) and the global Cartesian basis (x_g, y_g, z_g) — or an alternate computational nodal basis (x_c^a, y_c^a, z_c^a) — are such that the respective z axes are parallel.

Remark 3.13 The D array is input to various standard kernel routines, such as ES0E (strains), ES0FI (internal forces) and ES0KM/ES0KG (material/geometric stiffness). The components are always resolved in the element intrinsic basis, and for geometrically nonlinear analysis with corotation turned on, the displacements are relative to the moving element frame and hence should be "moderately small" in the absence of finite strains (see Chapter 4).

3.4.2.3.7 Plane-Element Force Vector (The F Arrays)

The force vector for a plane-element either internal (FI) or external (FB or FS) should be output from the standard kernel routines (ES0FI, ES0FB or ES0FS) as an array that is in one-to-one correspondence with the element displacement vector. Thus,

$$\mathrm{F(2,NEN)} = \begin{bmatrix} F^1_{x_e} & F^2_{x_e} & \cdots & F^{\mathrm{NEN}}_{x_e} \\ F^1_{y_e} & F^2_{y_e} & \cdots & F^{\mathrm{NEN}}_{y_e} \end{bmatrix}$$

where $F^a_{x_e}$ and $F^a_{y_e}$ are the forces in the x_e and y_e directions at element node a, and the developer's node sequence is implied.

3.4.2.3.8 Plane-Element Stiffness Matrices (The K Arrays)

The stiffness matrix for a plane-element — either material (KM), geometric (KG) or load (KL) — should be output from the standard kernel routines (ES0KM, ES0KG or ES0KL) in the following *upper-triangular* form:

$$\mathrm{K}\big((NEE^2+NEE)/2\big) = \begin{array}{c} \\ F^1_{x_e} \\ F^1_{y_e} \\ \vdots \\ F^{\mathrm{NEN}}_{x_e} \\ F^{\mathrm{NEN}}_{y_e} \end{array} \begin{array}{c} \begin{array}{cccccc} u^1_e & v^1_e & \cdots & u^{\mathrm{NEN}}_e & v^{\mathrm{NEN}}_e \end{array} \\ \begin{pmatrix} K_{11} & K_{12} & \cdots & K_{1,\mathrm{NEE}-1} & K_{1,\mathrm{NEE}} \\ & & \ddots & & \vdots \\ & & & & K_{\mathrm{NEE,NEE}} \end{pmatrix} \end{array}$$

where NEE = 2 × NEN and the numbers are stored as one partial column after another (see glossary in Section 3.3), and the node numbers correspond to the developer's node sequence, not the user's.

3.4.2.4 Plate Elements

Distinct conventions for plate elements are *currently not implemented* as a standard ES processor option. However, the developer is free to implement plate elements as a special case of *shell elements* (see Section 3.4.2.3), wherein certain partitions of the integrated constitutive matrix are ignored, and certain nodal degrees of freedom are suppressed by the user.

3.4.2.5 Shell Elements

Shell elements are defined as 2-D elements (*i.e.*, two intrinsic surface coordinates) that are embedded in a three-dimensional spatial setting, such that all six standard nodal freedoms (three translations and three rotations) are, in general, potentially active; and for which stress *resultants* (*i.e.*, stresses integrated through the shell thickness) are employed for internal force and stiffness calculations instead of continuum stress components. This general definition contains plate elements, membrane elements, and both flat and curved shell elements as special cases.

Remark 3.14 While resultant stresses and strains are considered the intrinsic element parameters, continuum stress and strain components may be used in the constitutive relations — especially for nonlinear materials (see Chapter 5).

Remark 3.15 So-called degenerated solid elements, which employ rotational freedoms — but use continuum stress components — would be considered 3-D elements (see Section 3.4.3).

3.4.2.5.1 Shell-Element Intrinsic Parameters (the DEFS Array)

The following settings should be used when defining the DEFS argument array (using kernel subroutine ES0D) for shell elements:

$$
\begin{aligned}
&\text{DEFS(pdCLAS)} = \text{idSHEL}\\
&\text{DEFS(pdDIM)} = 3\\
&\text{DEFS(pdNDOF)} = 6\\
&\text{DEFS(pdC)} = \begin{cases} 1 \text{ for } C^1 \text{ elements} \\ 0 \text{ for } C^0 \text{ elements} \end{cases}\\
&\text{DEFS(pdNSTR)} = \begin{cases} 6 \text{ for } C^1 \text{ elements} \\ 8 \text{ for } C^0 \text{ elements} \end{cases}\\
&\text{DEFS(pdPROJ)} = 1 \quad \text{(recommended)}
\end{aligned}
$$

where the array indices, pd*, and the constant, idSHEL, are defined in all of the standard kernel routines using built-in parameter statements (see include block ES0PTR.INC).

Other DEFS array entries should be set according to the descriptions given in the standard kernel argument glossary (Section 3.3).

3.4.2.5.2 Shell-Element Nodal Freedoms (the DOFS Array)

The DOFS array, which defines (using kernel subroutine ESOD) the potentially active nodal degrees of freedom for an element type, would typically be set by the developer as follows for a shell element:

$$\text{DOFS(6,NEN)} = \begin{array}{c} \\ u \\ v \\ w \\ \theta_x \\ \theta_y \\ \theta_z \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & \ldots & \text{NEN} \end{array} \\ \begin{pmatrix} 1 & 1 & \ldots & 1 \\ 1 & 1 & \ldots & 1 \\ 1 & 1 & \ldots & 1 \\ 1 & 1 & \ldots & 1 \\ 1 & 1 & \ldots & 1 \\ 1 & 1 & \ldots & 1 \end{pmatrix} \end{array}$$

where the ones in the table indicate that all translational freedoms (u, v, w) and rotational freedoms $(\theta_x, \theta_y, \theta_z)$ are potentially active at every element node. The directions of these freedoms are interpreted as corresponding to the computational nodal basis used to express the *assembled* system of equations. Thus, it would probably not be meaningful to turn off individual translation or rotation components at specific nodes using the DOFS array. However, it would make sense to permanently suppress *all* translational degrees of freedom or *all* rotational degrees of freedom at selected nodes; or to suppress *both* translational and rotational degrees of freedom, *e.g.*, at nodes which are used only for *geometric* definition.

3.4.2.5.3 Shell-Element Strains (The E Array)

For shell elements, the standard convention for the strain array, E(NSTR,NIP), is as follows. For C^1 shell elements (DEFS(pdC)=1 and DEFS(pdNSTR)=6):

$$\mathrm{E(6,NIP)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & \ldots & \mathrm{NIP} \end{array} \\ \begin{pmatrix} \bar{\epsilon}_{xx}(\xi_1) & \bar{\epsilon}_{xx}(\xi_2) & \ldots & \bar{\epsilon}_{xx}(\xi_{\mathrm{NIP}}) \\ \bar{\epsilon}_{yy}(\xi_1) & \bar{\epsilon}_{yy}(\xi_2) & \ldots & \bar{\epsilon}_{yy}(\xi_{\mathrm{NIP}}) \\ \bar{\epsilon}_{xy}(\xi_1) & \bar{\epsilon}_{xy}(\xi_2) & \ldots & \bar{\epsilon}_{xy}(\xi_{\mathrm{NIP}}) \\ \kappa_{xx}(\xi_1) & \kappa_{xx}(\xi_2) & \ldots & \kappa_{xx}(\xi_{\mathrm{NIP}}) \\ \kappa_{yy}(\xi_1) & \kappa_{yy}(\xi_2) & \ldots & \kappa_{yy}(\xi_{\mathrm{NIP}}) \\ \kappa_{xy}(\xi_1) & \kappa_{xy}(\xi_2) & \ldots & \kappa_{xy}(\xi_{\mathrm{NIP}}) \end{pmatrix} \end{array}$$

where $\bar{\epsilon}_{\alpha\beta}$ are the *reference-surface* strains, and $\kappa_{\alpha\beta}$ are the *curvature* changes.

Similarly, for C^0 shell elements (DEFS(pdC=0) and DEFS(pdNSTR)=8):

$$\mathrm{E(8,NIP)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & \ldots & \mathrm{NIP} \end{array} \\ \begin{pmatrix} \bar{\epsilon}_{xx}(\xi_1) & \bar{\epsilon}_{xx}(\xi_2) & \ldots & \bar{\epsilon}_{xx}(\xi_{\mathrm{NIP}}) \\ \bar{\epsilon}_{yy}(\xi_1) & \bar{\epsilon}_{yy}(\xi_2) & \ldots & \bar{\epsilon}_{yy}(\xi_{\mathrm{NIP}}) \\ \bar{\epsilon}_{xy}(\xi_1) & \bar{\epsilon}_{xy}(\xi_2) & \ldots & \bar{\epsilon}_{xy}(\xi_{\mathrm{NIP}}) \\ \kappa_{xx}(\xi_1) & \kappa_{xx}(\xi_2) & \ldots & \kappa_{xx}(\xi_{\mathrm{NIP}}) \\ \kappa_{yy}(\xi_1) & \kappa_{yy}(\xi_2) & \ldots & \kappa_{yy}(\xi_{\mathrm{NIP}}) \\ \kappa_{xy}(\xi_1) & \kappa_{xy}(\xi_2) & \ldots & \kappa_{xy}(\xi_{\mathrm{NIP}}) \\ \gamma_x(\xi_1) & \gamma_x(\xi_2) & \ldots & \gamma_x(\xi_{\mathrm{NIP}}) \\ \gamma_y(\xi_1) & \gamma_y(\xi_2) & \ldots & \gamma_y(\xi_{\mathrm{NIP}}) \end{pmatrix} \end{array}$$

where γ_α are some "average" measure of the *transverse-shear* strains through the shell thickness.

The strain directions (*i.e.*, the meaning of the x and y subscripts) and the surface coordinate conventions (*i.e.*, the meaning of ξ, η) are developer-selected, and should be described in an appropriate section of the Testbed Users Manual (see ref. 4).

Remark 3.16 The "normal" strain component, ϵ_{zz}, is assumed to be irrelevant due to the plane stress condition, σ_{zz}. However, normal strains can usually be recovered through the constitutive relations.

Remark 3.17 All shear resultants: $\bar{\epsilon}_{xy}, \kappa_{xy}, \gamma_x, \gamma_y$, are interpreted as deriving from *engineering* shear strains, *i.e.*, they are a factor of two larger than the corresponding tensorial shear strains.

Remark 3.18 The type of strain-displacement relations used (*i.e.*, linear or nonlinear) are also up to the developer. If the built-in corotational option is selected by the user, then — as long as the strains remain small — it is not really necessary for the developer to provide nonlinear strain-displacement relations, regardless of the magnitude of the rotations. However, it may be desirable to include a nonlinear option for improved accuracy (see kernel input argument CTLS(pcCORO) and Chapter 4).

3.4.2.5.4 Shell-Element Stresses (the S Array)

For shell-elements, the standard convention for the stress array, S(NSTR,NIP), is a set of stress *resultants*, which are (incrementally) work-conjugate to the resultant strain measures described above. Thus, the number of intrinsic stress resultants depends on whether or not the element exhibits transverse-shear strains (*i.e.*, C^0 versus C^1 displacement compatibility).

For C^1 shell elements (DEFS(pdC)=1 and DEFS(pdNSTR)=6):

$$\mathrm{S(6,NIP)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & \cdots & \mathrm{NIP} \end{array} \\ \begin{pmatrix} n_{xx}(\xi_1) & n_{xx}(\xi_2) & \cdots & n_{xx}(\xi_{\mathrm{NIP}}) \\ n_{yy}(\xi_1) & n_{yy}(\xi_2) & \cdots & n_{yy}(\xi_{\mathrm{NIP}}) \\ n_{xy}(\xi_1) & n_{xy}(\xi_2) & \cdots & n_{xy}(\xi_{\mathrm{NIP}}) \\ m_{xx}(\xi_1) & m_{xx}(\xi_2) & \cdots & m_{xx}(\xi_{\mathrm{NIP}}) \\ m_{yy}(\xi_1) & m_{yy}(\xi_2) & \cdots & m_{yy}(\xi_{\mathrm{NIP}}) \\ m_{xy}(\xi_1) & m_{xy}(\xi_2) & \cdots & m_{xy}(\xi_{\mathrm{NIP}}) \end{pmatrix} \end{array}$$

where $\xi_p = \{\xi_p, \eta_p\}$ are the element surface coordinates at integration point p, and where the direct stress resultants, $n_{\alpha\beta}$, and moment stress resultants $m_{\alpha\beta}$ are typically defined as follows:†

$$n_{\alpha\beta} = \int_z \sigma_{\alpha\beta} dz \qquad m_{\alpha\beta} = \int_z z\,\sigma_{\alpha\beta} dz$$

in which α and β range between 1 and 2, corresponding to x and y, respectively.

For C^0 shell elements (DEFS(pdC)=0 and DEFS(pdNSTR)=8), the first six stress resultants are defined exactly as for C^1 elements, with the additional two components being

---

† It is not necessary for the developer to take these definitions literally unless the standard constitutive interface is to be employed (see Chapter 5), in which case the precise relationship between resultant and continuum stresses and strains must adhere to the conventions described herein.

the transverse-shear stress resultants, *i.e.*,

$$
\mathrm{S(8,NIP)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array}
\begin{array}{c}
\begin{array}{cccc} 1 & 2 & \cdots & \mathrm{NIP} \end{array} \\
\begin{pmatrix}
n_{xx}(\xi_1) & n_{xx}(\xi_2) & \cdots & n_{xx}(\xi_{\mathrm{NIP}}) \\
n_{yy}(\xi_1) & n_{yy}(\xi_2) & \cdots & n_{yy}(\xi_{\mathrm{NIP}}) \\
n_{xy}(\xi_1) & n_{xy}(\xi_2) & \cdots & n_{xy}(\xi_{\mathrm{NIP}}) \\
m_{xx}(\xi_1) & m_{xx}(\xi_2) & \cdots & m_{xx}(\xi_{\mathrm{NIP}}) \\
m_{yy}(\xi_1) & m_{yy}(\xi_2) & \cdots & m_{yy}(\xi_{\mathrm{NIP}}) \\
m_{xy}(\xi_1) & m_{xy}(\xi_2) & \cdots & m_{xy}(\xi_{\mathrm{NIP}}) \\
q_x(\xi_1) & q_x(\xi_2) & \cdots & q_x(\xi_{\mathrm{NIP}}) \\
q_y(\xi_1) & q_y(\xi_2) & \cdots & q_y(\xi_{\mathrm{NIP}})
\end{pmatrix}
\end{array}
$$

where

$$q_\alpha = \int_z \sigma_{z\alpha} dz$$

The in-plane stress directions (*i.e.*, the meaning of the x and y subscripts) and the surface coordinate conventions (*i.e.*, the meaning of ξ, η) are developer-selected, and should be described in an appropriate section of the Testbed Users Manual (see ref. 4).

Remark 3.19 The "normal" stress component, σ_{zz}, is assumed to be zero for standard shell elements; hence, there is no corresponding stress resultant.

Remark 3.20 The type of stress components assumed (*e.g.*, nominal or true) depends on the constitutive algorithm. For problems involving small strains, the components can be interpreted as true (*i.e.*, Cauchy) stresses.

Remark 3.21 The reader may be wondering if it is allowable for C^1 shell elements to compute transverse-shear stress resultants. The answer is yes; but since C^1 elements, by definition, lack transverse-shear *strains*, the transverse-shear resultants must be recovered using a separate equilibrium post-processing step. For convenience, such quantities are stored in a separate array, SX, which is referred to only in kernel routine ESOPS (*currently not implemented*).

3.4.2.5.5 Shell-Element Constitutive Matrix (The C Array)

For shell elements, the standard convention for the constitutive array, C(NSTR,NSTR,NIP), is consistent with the definitions for the stress (S) and strain (E) arrays given above. The tangent constitutive matrix relating incremental strains to incremental stresses at integration point p is thus defined for C^1 elements by:

$$
\mathrm{C}(6,6,p) = \begin{array}{c} \\ n_{xx} \\ n_{yy} \\ n_{xy} \\ m_{xx} \\ m_{yy} \\ m_{xy} \end{array}
\begin{array}{c}
\begin{array}{cccccc} \bar{\epsilon}_{xx} & \bar{\epsilon}_{yy} & \bar{\epsilon}_{xy} & \kappa_{xx} & \kappa_{yy} & \kappa_{xy} \end{array} \\
\begin{pmatrix}
\overline{C}_{11}(\xi_p) & \overline{C}_{12}(\xi_p) & \overline{C}_{13}(\xi_p) & \overline{C}_{14}(\xi_p) & \overline{C}_{15}(\xi_p) & \overline{C}_{16}(\xi_p) \\
 & \overline{C}_{22}(\xi_p) & \overline{C}_{23}(\xi_p) & \overline{C}_{24}(\xi_p) & \overline{C}_{25}(\xi_p) & \overline{C}_{26}(\xi_p) \\
 & & \overline{C}_{33}(\xi_p) & \overline{C}_{34}(\xi_p) & \overline{C}_{35}(\xi_p) & \overline{C}_{36}(\xi_p) \\
 & & & \overline{C}_{44}(\xi_p) & \overline{C}_{45}(\xi_p) & \overline{C}_{46}(\xi_p) \\
 & & & & \overline{C}_{55}(\xi_p) & \overline{C}_{56}(\xi_p) \\
 & sym. & & & & \overline{C}_{66}(\xi_p)
\end{pmatrix}
\end{array}
$$

and for C^0 elements by:

$$
\mathrm{C}(8,8,p) = \begin{array}{c} \\ n_{xx} \\ n_{yy} \\ n_{xy} \\ m_{xx} \\ m_{yy} \\ m_{xy} \\ q_x \\ q_y \end{array}
\begin{array}{c}
\begin{array}{cccccccc} \bar{\epsilon}_{xx} & \bar{\epsilon}_{yy} & \bar{\epsilon}_{xy} & \kappa_{xx} & \kappa_{yy} & \kappa_{xy} & & \end{array} \\
\begin{pmatrix}
\overline{C}_{11}(\xi_p) & \overline{C}_{12}(\xi_p) & \overline{C}_{13}(\xi_p) & \overline{C}_{14}(\xi_p) & \overline{C}_{15}(\xi_p) & \overline{C}_{16}(\xi_p) & 0 & 0 \\
 & \overline{C}_{22}(\xi_p) & \overline{C}_{23}(\xi_p) & \overline{C}_{24}(\xi_p) & \overline{C}_{25}(\xi_p) & \overline{C}_{26}(\xi_p) & 0 & 0 \\
 & & \overline{C}_{33}(\xi_p) & \overline{C}_{34}(\xi_p) & \overline{C}_{35}(\xi_p) & \overline{C}_{36}(\xi_p) & 0 & 0 \\
 & & & \overline{C}_{44}(\xi_p) & \overline{C}_{45}(\xi_p) & \overline{C}_{46}(\xi_p) & 0 & 0 \\
 & & & & \overline{C}_{55}(\xi_p) & \overline{C}_{56}(\xi_p) & 0 & 0 \\
 & & & & & \overline{C}_{66}(\xi_p) & 0 & 0 \\
 & & & & & & \overline{C}_{77}(\xi_p) & \overline{C}_{78}(\xi_p) \\
 & sym. & & & & & & \overline{C}_{88}(\xi_p)
\end{pmatrix}
\end{array}
$$

where the $\overline{C}_{ij}$ are integrated (through-the-thickness) constitutive coefficients, which depend on both material and section properties. The actual definitions will depend on the constitutive processing option selected (see Chapter 5).

As with the stress and strain arrays (S and E), the interpretation of the directions used for the components and the choice of surface coordinates are left up to the element developer. The transformations between the material coordinate system (*i.e.*, the system used for constitutive calculations) and the element developer's stress/strain system (*i.e.*, the system used for the C array) are performed automatically by the ES processor shell, as described in Chapter 5.

3.4.2.5.6 Shell-Element Displacement Vector (The D Array)

The displacement vector for a shell element can be viewed as a singly dimensioned array, of length NEE = NDOF×NEN = 6×NEN (see glossary in Section 3.3), or as a two-dimensional array, defined as follows:

$$\text{D(6,NEN)} = \begin{bmatrix} u_e^1 & u_e^2 & \dots & u_e^{\text{NEN}} \\ v_e^1 & v_e^2 & \dots & v_e^{\text{NEN}} \\ w_e^1 & w_e^2 & \dots & w_e^{\text{NEN}} \\ \theta_{x_e}^1 & \theta_{x_e}^2 & \dots & \theta_{x_e}^{\text{NEN}} \\ \theta_{y_e}^1 & \theta_{y_e}^2 & \dots & \theta_{y_e}^{\text{NEN}} \\ \theta_{z_e}^1 & \theta_{z_e}^2 & \dots & \theta_{z_e}^{\text{NEN}} \end{bmatrix}$$

where u_e^a, v_e^a, w_e^a are the translations in the x_e, y_e, z_e directions, respectively, at element node a; and $\theta_{x_e}^a, \theta_{y_e}^a$ and $\theta_{z_e}^a$ are rotations about the corresponding axes. (See definition of *intrinsic element basis* given in Figure 3.3a)

The node sequence here refers to the *developer's* order — not the user's (Figure 3.3b).

Remark 3.22 The D array is input to various standard kernel routines, such as ES0E (strains), ES0FI (internal forces) and ES0KM/ES0KG (material/geometric stiffness). The components are always resolved in the element intrinsic basis, and, for geometrically nonlinear analysis with corotation turned on, the displacements are relative to the moving element frame and hence should be "moderately small" in the absence of finite strains (see Chapter 4).

3.4.2.5.7 Shell-Element Force Vectors (The F Arrays)

The force vector for a shell element either internal (FI) or external (FB or FS) should be output from the standard kernel routines (ES0FI, ES0FB or ES0FS) as an array that is

in one-to-one correspondence with the element displacement vector. Thus:

$$\mathrm{F(6,NEN)} = \begin{bmatrix} F^1_{xe} & F^2_{xe} & \dots & F^{\mathrm{NEN}}_{xe} \\ F^1_{ye} & F^2_{ye} & \dots & F^{\mathrm{NEN}}_{ye} \\ F^1_{ze} & F^2_{ze} & \dots & F^{\mathrm{NEN}}_{ze} \\ M^1_{xe} & M^2_{xe} & \dots & M^{\mathrm{NEN}}_{xe} \\ M^1_{ye} & M^2_{ye} & \dots & M^{\mathrm{NEN}}_{ye} \\ M^1_{ze} & M^2_{ze} & \dots & M^{\mathrm{NEN}}_{ze} \end{bmatrix}$$

where $F^a_{xe}, F^a_{ye}, F^a_{ze}$ are the forces in the x_e, y_e, z_e (intrinsic element basis) directions, respectively, at element node a; and $M^a_{xe}, M^a_{ye}, M^a_{ze}$ are the moments about the corresponding axes. The developer's node sequence is implied.

3.4.2.5.8 Shell-Element Stiffness Matrices (The K Arrays)

The stiffness matrix for a shell-element — either material (KM), geometric (KG), load (KL) or tangent (KT) — should be output from the standard kernel routines (ES0KM, ES0KG or ES0KL) in the following *upper-triangular* form:

$$\mathrm{K}\left((NEE^2+NEE)/2\right) = \begin{array}{c} \\ F^1_{xe} \\ F^1_{ye} \\ F^1_{ze} \\ M^1_{xe} \\ M^1_{ye} \\ M^1_{ze} \\ \vdots \\ M^{\mathrm{NEN}}_{ze} \end{array} \begin{array}{c} \begin{array}{cccccccc} u^1_e & v^1_e & w^1_e & \theta^1_{xe} & \theta^1_{ye} & \theta^1_{ze} & \dots & \theta^{\mathrm{NEN}}_{ze} \end{array} \\ \begin{pmatrix} K_{11} & K_{12} & K_{13} & K_{14} & K_{15} & K_{16} & \dots & K_{1,\mathrm{NEE}} \\ & K_{22} & K_{23} & K_{24} & K_{25} & K_{26} & \dots & K_{2,\mathrm{NEE}} \\ & & K_{33} & K_{34} & K_{35} & K_{36} & \dots & K_{3,\mathrm{NEE}} \\ & & & K_{44} & K_{45} & K_{46} & \dots & K_{4,\mathrm{NEE}} \\ & & & & K_{55} & K_{56} & \dots & K_{5,\mathrm{NEE}} \\ & & & & & K_{66} & \dots & K_{6,\mathrm{NEE}} \\ & & & & & & \ddots & \vdots \\ & & & & & & & K_{\mathrm{NEE,NEE}} \end{pmatrix} \end{array}$$

where NEE = 6 × NEN and the numbers are stored as one partial column after another (see glossary in Section 3.3), and the node numbers correspond to the developer's node sequence, not the user's.

3.4.3 3-D Solid Continuum Elements

Three-dimensional solid continuum elements are defined as those elements whose nodes define a solid volume of three-dimensional space, such that three intrinsic coordinates are required to locate intermediate (*e.g.*, integration) points. All three translational degrees of freedom are assumed to be present at element nodes (in general), and all six continuum stress/strain components are assumed to contribute to the element strain energy. Henceforth, these intrinsically 3-D elements will simply be referred to as *solid elements*.

3.4.3.1 User Node Numbers and Coordinate Systems for 3-D Elements

Figure 3.5a shows how the *corner node* numbers, as defined by the user, of one of the element's surfaces determine the *intrinsic* element coordinate system (x_e, y_e, z_e) for standard 3-D elements. Figure 3.5b shows the complete user node-numbering sequence, which always starts with the basic surface corner nodes, for various standard low-order and high-order 3-D elements.

Note that all standard 3-D elements are assumed to have at least one triangular or quadrilateral Note that all standard 3-D elements are assumed to have at least one triangular or quadrilateral surface (corresponding to kernel argument DEFS(pdSHAP) = idTRIA or idQUAD), with either the first 3 or the first 4 nodes, respectively, used to define the element intrinsic coordinate system (x_e, y_e, z_e). As shown in Figure 3.5, the first 3 or 4 nodes must be the *corner* nodes of that surface, with a *counter-clockwise* convention used to define the positve z_e direction.

The convention for establishing the intrinsic coordinate system is identical to that used for 2-D elements (Section 3.4.2.1), except based on the surface defined by the *first* 3 or 4 corner nodes. As in the 2-D case, the developer pararameter DEFS(pdTGE) can be used to switch the relationship of the x_e or y_e axes to the element edges. Furthermore, if the first surface of the 3-D element is a quadrilateral, an "average" plane is employed for the $x_e - y_e$ axes, to improve the performance of the element when used in conjunction with the corotational or projection operators described in Chapter 4.

3.4.3.2 Developer Node Numbers for 3-D Elements (NODES Array).

The standard or user node-numbering sequence for 3-D elements is intended to facilitate model generation (for users). To facilitate element development, the developer may use an internal node-numbering sequence which is different from the standard one. The correspondence between these two sequences must be conveyed by the developer through the NODES array (see Section 3.3), which the developer is asked to define in standard kernel routine ES0D.

The NODES array should give the user node index as a function of the developer node index. The following examples show some of the most common uses of the NODES array for 3-D elements:

Example 6.

If the element developer's node order is identical to the standard order (shown in Figure 3.5b), then the developer would set

$$\text{NODES}(a) = a \quad (a = 1, \ldots, \text{NEN})$$

in kernel routine ES0D; where NEN = 4 or 8 or 10 or 20, etc.

Example 7.

If the developer's node sequence for an 8-node "brick" element were as shown in Figure 3.6a, then the developer would set:

NODES(1) = 1

NODES(2) = 5

NODES(3) = 2

NODES(4) = 6

NODES(5) = 4

NODES(6) = 8

NODES(7) = 3

NODES(8) = 7

in kernel routine ES0D.

Example 8.

If the developer's node sequence for a 27-node quadrilateral element were as shown in Figure 3.6b, then the developer would set:

NODES(1) = 1	NODES(10) = 13	NODES(19) = 5
NODES(2) = 9	NODES(11) = 26	NODES(20) = 17
NODES(3) = 2	NODES(12) = 14	NODES(21) = 6
NODES(4) = 12	NODES(13) = 24	NODES(22) = 20
NODES(5) = 22	NODES(14) = 21	NODES(23) = 23
NODES(6) = 10	NODES(15) = 25	NODES(24) = 18
NODES(7) = 4	NODES(16) = 16	NODES(25) = 8
NODES(8) = 11	NODES(17) = 27	NODES(26) = 19
NODES(9) = 3	NODES(18) = 15	NODES(27) = 7

in kernel routine ES0D.

3.4.3.3 Solid (Continuum) Elements

Solid elements are defined as 3-D continuum elements which have three translational freedoms at each node, and all six stress components contributing to the strain energy at each integration point.

3.4.3.3.1 Solid-Element Intrinsic Parameters (the DEFS Array)

The following settings should be used when defining the DEFS argument array (using kernel subroutine ESOD) for solid elements:

```
DEFS(pdCLAS) = idSOLI
DEFS(pdDIM)  = 3
DEFS(pdNDOF) = 3
DEFS(pdNSTR) = 6
```

Other DEFS array entries should be set according to the descriptions given in the standard kernel argument glossary (Section 3.3).

3.4.3.3.2 Solid-Element Nodal Freedoms (the DOFS Array)

The DOFS array, which defines the potentially active nodal freedoms for an element type (in kernel subroutine ESOD), would typically be set by the developer as follows for a solid element:

$$\text{DOFS(3,NEN)} = \begin{array}{c} \\ u \\ v \\ w \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & \ldots & \text{NEN} \end{array} \\ \begin{pmatrix} 1 & 1 & \ldots & 1 \\ 1 & 1 & \ldots & 1 \\ 1 & 1 & \ldots & 1 \end{pmatrix} \end{array}$$

where the ones in the table indicate that all translational displacement degrees of freedom (u, v and w) are potentially active at every element node. The directions of these freedoms are interpreted as corresponding to the computational nodal basis used to express the *assembled* system of equations. Thus, it would probably not be meaningful to selectively turn off individual u, v or w element nodal freedoms using the DOFS array — since the

element developer has no control over which the computational basis will be selected; it is selected by the user as a global attribute. However, it may be useful to permanently suppress all three freedoms at selected nodes; *e.g.*, if some of the nodes are used only for *geometric* definition.

3.4.3.3.3 Solid-Element Strains (The E Array)

For solid elements, the standard convention for the strain array, E(6,NIP), is as follows:

$$\text{E(6,NIP)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & \ldots & \text{NIP} \end{array} \\ \begin{pmatrix} \epsilon_{xx}(\xi_1) & \epsilon_{xx}(\xi_2) & \ldots & \epsilon_{xx}(\xi_{\text{NIP}}) \\ \epsilon_{yy}(\xi_1) & \epsilon_{yy}(\xi_2) & \ldots & \epsilon_{yy}(\xi_{\text{NIP}}) \\ \epsilon_{zz}(\xi_1) & \epsilon_{zz}(\xi_2) & \ldots & \epsilon_{zz}(\xi_{\text{NIP}}) \\ \epsilon_{yz}(\xi_1) & \epsilon_{yz}(\xi_2) & \ldots & \epsilon_{yz}(\xi_{\text{NIP}}) \\ \epsilon_{zx}(\xi_1) & \epsilon_{zx}(\xi_2) & \ldots & \epsilon_{zx}(\xi_{\text{NIP}}) \\ \epsilon_{xy}(\xi_1) & \epsilon_{xy}(\xi_2) & \ldots & \epsilon_{xy}(\xi_{\text{NIP}}) \end{pmatrix} \end{array}$$

where $\boldsymbol{\xi}_p = \{\xi_p, \eta_p, \zeta_p\}$ are the element natural coordinates at integration point p; and x, y, z refer to the element stress basis, x_s, y_s, z_s. Both coordinate systems are defined by the developer.

Remark 3.23 The type of strain-displacement relations used (*i.e.*, linear or nonlinear) are also up to the developer. If the built-in corotational option is selected by the user, then — as long as the strains remain small — it is not really necessary for the developer to provide nonlinear strain-displacement relations, regardless of the magnitude of the rotations. However, it may be desirable to include a nonlinear option for improved accuracy (see kernel input argument CTLS(pcCORO) and Chapter 4). This is especially true for solid elements used to model a *compact* continuum, where the corotational option may not be particularly useful unless the whole structure (or substructure) is undergoing large rigid-body motion — otherwise large rotations will be accompanied by large strains, and a nonlinear strain measure will be mandatory.

3.4.3.3.4 Solid-Element Stresses (the S Array)

For solid elements, the standard convention for the stress array, S(6,NIP), is as follows:

$$\mathrm{S(6,NIP)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & \ldots & \mathrm{NIP} \end{array} \\ \begin{pmatrix} \sigma_{xx}(\xi_1) & \sigma_{xx}(\xi_2) & \ldots & \sigma_{xx}(\xi_{\mathrm{NIP}}) \\ \sigma_{yy}(\xi_1) & \sigma_{yy}(\xi_2) & \ldots & \sigma_{yy}(\xi_{\mathrm{NIP}}) \\ \sigma_{zz}(\xi_1) & \sigma_{zz}(\xi_2) & \ldots & \sigma_{zz}(\xi_{\mathrm{NIP}}) \\ \sigma_{yz}(\xi_1) & \sigma_{yz}(\xi_2) & \ldots & \sigma_{yz}(\xi_{\mathrm{NIP}}) \\ \sigma_{zx}(\xi_1) & \sigma_{zx}(\xi_2) & \ldots & \sigma_{zx}(\xi_{\mathrm{NIP}}) \\ \sigma_{xy}(\xi_1) & \sigma_{xy}(\xi_2) & \ldots & \sigma_{xy}(\xi_{\mathrm{NIP}}) \end{pmatrix} \end{array}$$

where $\xi_p = \{\xi_p, \eta_p \zeta_p\}$ are the element natural coordinates at integration point p; and x, y, z refer to the element stress basis, x_s, y_s, z_s. Both coordinate systems are defined by the developer.

Remark 3.24 If the developer has selected the constitutive option: DEFS(pdCNS) equals 0 or 2, the element stress array will be computed automatically using the ES processor shell.

Remark 3.25 The type of stress components assumed (*e.g.*, nominal or true) depends on the constitutive algorithm (see Chapter 5). For problems involving small strains, the components can be interpreted as true (*i.e.*, Cauchy) stresses.

3.4.3.3.5 Solid-Element Constitutive Matrix (The C Array)

For solid elements, the standard convention for the constitutive array, C(6,6,NIP), is consistent with the definition of the stress (S) and strain (E) arrays. Thus, the tangent constitutive matrix relating incremental strains to incremental stresses at integration point p

is defined by

$$
\mathrm{C(6,6,}p\mathrm{)} = \begin{array}{c} \\ \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{zz} \\ \sigma_{yz} \\ \sigma_{zx} \\ \sigma_{xy} \end{array}
\begin{array}{c}
\begin{array}{cccccc} \epsilon_{xx} & \epsilon_{yy} & \epsilon_{zz} & \epsilon_{yz} & \epsilon_{zx} & \epsilon_{xy} \end{array} \\
\begin{pmatrix}
C_{11}(\xi_p) & C_{12}(\xi_p) & C_{13}(\xi_p) & C_{14}(\xi_p) & C_{15}(\xi_p) & C_{16}(\xi_p) \\
 & C_{22}(\xi_p) & C_{23}(\xi_p) & C_{24}(\xi_p) & C_{25}(\xi_p) & C_{26}(\xi_p) \\
 & & C_{33}(\xi_p) & C_{34}(\xi_p) & C_{35}(\xi_p) & C_{36}(\xi_p) \\
 & & & C_{44}(\xi_p) & C_{45}(\xi_p) & C_{46}(\xi_p) \\
 & & & & C_{55}(\xi_p) & C_{56}(\xi_p) \\
 sym. & & & & & C_{66}(\xi_p)
\end{pmatrix}
\end{array}
$$

where the values of the coefficients C_{ij} depend on the material properties (see Chapter 5). As with the stress and strain arrays (S and E), the interpretation of the directions used for the components and the choice of surface coordinates are left up to the element developer.

Remark 3.26 The transformations between the material coordinate system and the element developer's stress/strain system are performed automatically by the ES processor shell (see Chapter 5).

Remark 3.27 If the developer has selected the constitutive option: DEFS(pdCNS) equals 0 or 1, the constitutive matrix will be generated automatically by the ES processor shell.

3.4.3.3.6 Solid-Element Displacement Vector (The D Array)

The displacement vector for a solid-element can be viewed as a singly dimensioned array, of length NEE = NDOF×NEN = 3×NEN (see glossary in Section 3.3), or as a two-dimensional array, defined as follows:

$$
\mathrm{D(3,NEN)} = \begin{bmatrix} u_e^1 & u_e^2 & \dots & u_e^{\mathrm{NEN}} \\ v_e^1 & v_e^2 & \dots & v_e^{\mathrm{NEN}} \\ w_e^1 & w_e^2 & \dots & w_e^{\mathrm{NEN}} \end{bmatrix}
$$

where u_e^a, v_e^a and w_e^a are the translations in the x_e, y_e and z_e directions, respectively, at element node a (see definition of *intrinsic element basis* given in Figure 3.6a). The node sequence here refers to the *developer's* order — not to user's (Figure 3.6b).

3.4.3.3.7 Solid-Element Force Vectors (The F Arrays)

The force vector for a solid-element either internal (FI) or external (FB or FS) should be output from the standard kernel routines (ES0FI, ES0FB or ES0FS) as an array that is in one-to-one correspondence with the element displacement vector. Thus:

$$\mathrm{F(3,NEN)} = \begin{bmatrix} F_{xe}^1 & F_{xe}^2 & \cdots & F_{xe}^{\mathrm{NEN}} \\ F_{ye}^1 & F_{ye}^2 & \cdots & F_{ye}^{\mathrm{NEN}} \\ F_{ze}^1 & F_{ze}^2 & \cdots & F_{ze}^{\mathrm{NEN}} \end{bmatrix}$$

where F_{xe}^a, F_{ye}^a and F_{ze}^a are the forces in the x_e, y_e and z_e directions at element node a. The developer's node sequence is assumed.

3.4.3.3.8 Solid-Element Stiffness Matrices (The K Arrays)

The stiffness matrix for a solid-element — either material (KM), geometric (KG), load (KL) or tangent (KT) — should be output from the standard kernel routines (ES0KM, ES0KG or ES0KL) in the following *upper-triangular* form:

$$\mathrm{K}\big((NEE^2 + NEE)/2\big) = \begin{array}{c} \\ F_{xe}^1 \\ F_{ye}^1 \\ F_{ze}^1 \\ \vdots \\ F_{ze}^{\mathrm{NEN}} \end{array} \begin{array}{c} \begin{array}{ccccc} u_e^1 & v_e^1 & w_e^1 & \cdots & w_e^{\mathrm{NEN}} \end{array} \\ \begin{pmatrix} K_{11} & K_{12} & K_{13} & \cdots & K_{1,\mathrm{NEE}} \\ & K_{22} & K_{23} & \cdots & K_{2,\mathrm{NEE}} \\ & & K_{33} & \cdots & K_{3,\mathrm{NEE}} \\ & & & \ddots & \vdots \\ & & & & K_{\mathrm{NEE,NEE}} \end{pmatrix} \end{array}$$

where NEE = 3 × NEN and the numbers are stored as one partial column after another (see glossary in Section 3.3), and the node numbers correspond to the developer's node sequence, not the user's.

3.4.4 Nonstandard ("Wild") Elements

CURRENTLY NOT IMPLEMENTED

3.5 Step-by-Step Installation of New ES Processors

The following sections describe how to create and test new structural-element (ES) processors. Step-by-step instructions are provided with operating-sytem dependencies indicated where appropriate. Currently, only UNIX and VMS operating systems are being supported.

3.5.1 Part 1: How to Create a New ES Processor

Step 1.1: **Create a Working Directory.** This directory will be used to build and store your new element (ES) processor. You can call it anything you like.

Step 1.2: **Copy ES Processor Source-Code Template.** The source code file containing templates (subroutine calling sequences, argument declarations and default error exits) for each of the *standard* ES kernel routines (Section 3.2) is called **es_cover.ams**, and resides in a separate directory with logical name **CSM_ES**. Copy it to your present working directory using:

```
cp $CSM_ES/es_cover.ams .
```
on UNIX

or:

```
copy CSM_ES:es_cover.ams *
```
on VMS

Step 1.3: **Fill in Source-Code Template.** This step is crucial. Chapter 3 of this manual contains all of the reference documentation needed to complete the standard kernel routines contained in **es_cover.ams**, including a description of each standard entry point (Section 3.2) and a glossary of standard subroutine arguments (Section 3.3). The standard kernel routines are referred to as "cover routines" because they are usually employed only as an interface between the ES processor shell and the element developer's private kernel subroutines. The most important cover routines to complete first are ES0D (element definition), ES0KM (element material stiffness) and ES0E (element strain) — or ES0S if

it is an assumed-stress element. These routines will enable you to perform basic linear static analysis. Later, the element developer will need to complete ES0KG (element geometric stiffness), ES0FI (element internal force), etc., to perform stability and nonlinear analysis. ES0FI is needed even for linear analysis in the presence of thermal loads or initial stress/strain effects. Note that the mnemonic parameters, pd* and pc*, which appear in the argument glossary (Section 3.3), and may be used to reference individual items in the DEFS and CTLS arrays, respectively, are automatically defined by an "include" block called **es0ptr.inc** residing in the **CSM_ES** directory.

Step 1.4: **Rename Source Files.** Once you have begun to update the **es_cover.ams** file by completing some of the cover routines, rename the file to **es*i*_cover.ams**, where the *i* represents an integer processor sequence number, between 1 and 99, which will be used to uniquely identify your new element processor. (Check with the NASA/Langley CSM staff to get a list of available processor numbers.) For example, if you were creating processor ES10, you would enter:

```
mv es_cover.ams es10_cover.ams
```

on UNIX

or:

```
rename es_cover.ams es10_cover.ams
```

on VMS

Additionally, it will be convenient to consolidate your own nonstandard element routines, *i.e.*, the real kernel, into a file called **es*i*_kernel.ams**, where *i* again denotes the processor number. If you need to have more than one kernel file, or use some lower-level utilities that you wish to keep in a separate file, that's okay — but you'll have to edit the makefile procedure accordingly in step 1.5.

Step 1.5: **Copy/Edit Standard "Makefile" to Create New ES Processor.** When you have finished enough of the standard cover routines to be able to run a test case, copy the standard "makefile" procedure residing in directory **CSM_ES** to your working directory, as follows:

```
cp $CSM_EXE/makefile.es makefile
```

on UNIX

or:

```
copy CSM_SAM:makefile_es.com *
```
on VMS

If you have only one kernel file, **es*i*_kernel.ams** -- and no extra utilities — you can use the "makefile" procedure as-is. Otherwise, you will have to add the extra file names as indicated within the "makefile".

Step 1.6: Execute "Makefile" to Create New ES Processor. To compile the source code in **es*i*_cover.ams**, **es*i*_kernel.ams** and any other kernel/utility files, simply enter:

```
make ES = esi
```
on UNIX

or:

```
@makefile_es esi
```
on VMS

where i denotes the processor sequence number. Only those source files which have been updated will be compiled, and a new executable — called **es*i*** — will be created (unless an updated one already exists). The link procedure within the "makefile" will automatically include the necessary object code from the standard ES processor shell and Testbed architectural utility libraries to create the ES executable.

Step 1.7: Recompile/Link as Necessary using "Makefile". Step 1.6 can be repeated as necessary to successfully compile and test your new ES processor. See Part 2 for instructions on how to test an ES processor once it has been created.

3.5.2 Part 2: How to Test a New ES Processor

Step 2.1: **Create a Working Directory.** This directory can be the same as the one used to create your new ES processor (see Part 1, Section 3.5.1). It will be used to store the Testbed input and output files involved in running a test case, as well as a copy of the Testbed Procedure Database. Copy this latter file to your working directory using:

```
cp $CSM_PRC/proclib.gal .
```
on UNIX

or:

```
copy CSM_PRC:proclib.gal *
```
on VMS

Then, be sure to add the name of the directory where your new ES processor executable resides (see Part 1) to the PATH environment variable (on UNIX), or logical file name (on VMS), called CSM_EXTP before you proceed to Step 2.2 for the first time.

Step 2.2: **Copy One of the Sample Problem Directories.** It is a good idea to start with one of the existing test cases (if this is feasible) rather than creating one from scratch. Even if you wish to create a test case from scratch, it will be useful to look at one of the existing cases just as an example. Existing test cases may be found under the Testbed directory with logical name **CSM_PRC**. This directory contains a subdirectory named **applications** with one subdirectory for each test case. For example, you will find subdirectories with names like **pinched_cyl**, **hinged_cyl**, **elastica**, etc. Copy the entire contents of one of these sub-directories to your working directory. For example:

```
cp $CSM_PRC/applications/pinched_cyl/* .
```
on UNIX

or:

```
set def CSM_PRC:
set def [.APPLICATIONS.PINCHED_CYL]
copy *.* "file specifications for your working directory"
```
on VMS

would copy the necessary files for running the pinched cylinder problem to your directory. These files include: **pinched_cyl.com, pinched_cyl.clp, pinched_cyl.log** and **pinched_cyl.dbr**. The ***.com** file is the "input" file and * designates the name of the test case; it contains both operating system commands to run the Testbed, and a Testbed *call directive to invoke the actual test case procedure file, ***.clp**. The ***.log** file is the printed output file corresponding to the ***.com** input file. It contains verbose processor and procedure printout useful for =debugging a new processor. Finally, the ***.dbr** file is a Testbed database file containing selected results from the test case and is particularly useful for quick verification of a new processor.

Step 2.3: **Edit the "Input" File.** The *call directive normally contains a number of procedure argument definitions which define selected input parameters for the testcase. For example, the element processor name(s), element type name(s) and various grid and geometric parameters may be valid procedure arguments. As a minimum, you will have to modify the element processor and element type arguments, which are called ES_PROC and ES_NAME, respectively. For example, if your new element processor is called ES10 and you wish to test a shell element that you have named SHL1, you would modify the file **pinched_cyl.com** so that

```
*call  pinched_cyl ( ES_PROC = ES10; ES_NAME = SHL1; ...)
```

was a part of the procedure calling sequence.

Step 2.4: **Submit/Run the Input File.** Once you have modified the ***.com** file to accommodate your new ES processor, you can run the test case by simply "invoking" the ***.com** file. For the pinched cylinder case, this command would take the form:

```
pinched_cyl.com  >  & pinched_cyl.LOG&
```

on UNIX

or:

```
submit  pinched_cyl.com
```

on VMS

where **pinched_cyl.LOG** is the designated output file. (Note that due to the case sensitivity of UNIX the ***.LOG** file will be treated as a different file than

the original *.**log** file; while on VMS, the original file will be called *.**LOG;1** and the new file will be called *.**LOG;2** by default.) In addition to this output file, the computed results — as well as most intermediate data — will be deposited in the global database. The name(s) of the global database file(s) will normally be of the form *.**DBC** and *.**DBR** where * denotes the name of the test case, the *.**DBC** file contains the intermediate and bulk solution data, and the *.**DBR** file contains selected results. (The database file names usually can be reset using the procedure arguments; conventions may vary among procedures.)

Step 2.5: **Examine the Output (*.LOG) Text File.** Compare the *.**LOG** file obtained using Step 2.4 with the *.**log** file (on UNIX), or the *.**LOG;1** file (on VMS), copied from the original applications directory in Step 2.2. Consult the Testbed User's Manual (ref. 4) if you have trouble interpreting processor output. The two levels of output will be intertwined in the log file (unless the directive *echo,off is used which will suppress the procedure output).

Step 2.6: **Examine the Output Database File.** Consult the Testbed Data Library Description (ref. 6) and Chapter 6 of this manual to interpret the contents of the datasets in the database (*e.g.*, *.**DBC** or *.**DBR** files) generated by the test case. You may compare the contents of the *.**DBR** file with that of the *.**dbr** file (on UNIX), or the *.**DBR**;1 file (on VMS), copied from the original applications directory in Step 2.2.

Step 2.7: **Go back to Part 1.** New software rarely works correctly the first time. Be prepared to iterate, and to incrementally update your new processor until it has become functionally complete and can be used with confidence on real applications.

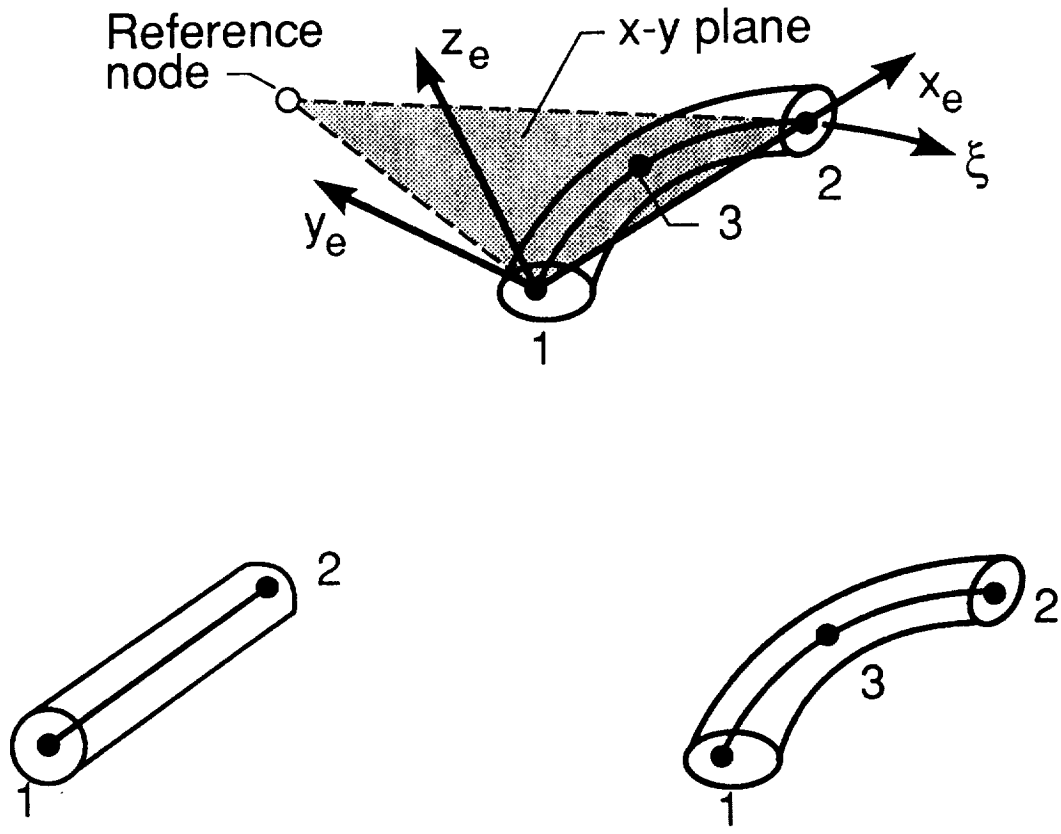


Figure 3.1 Conventions for Standard 1-D Elements.

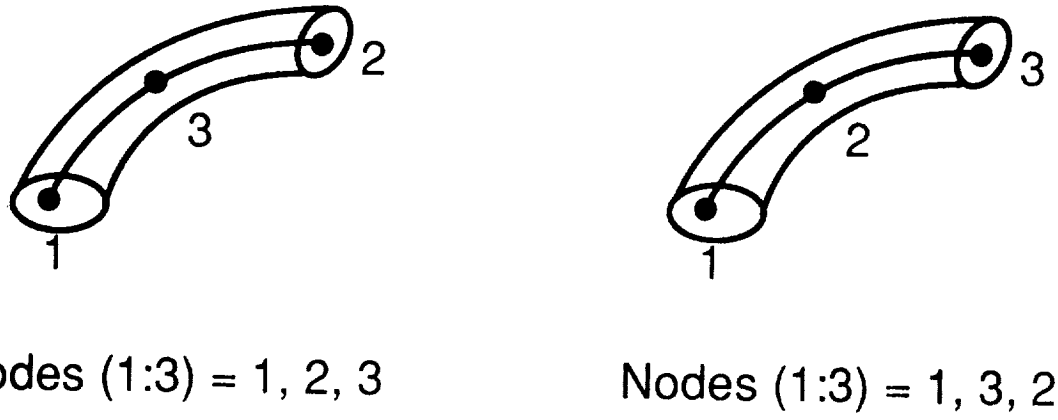


Figure 3.2 Examples of 1-D Elements with Non-Standard Node Numbering.

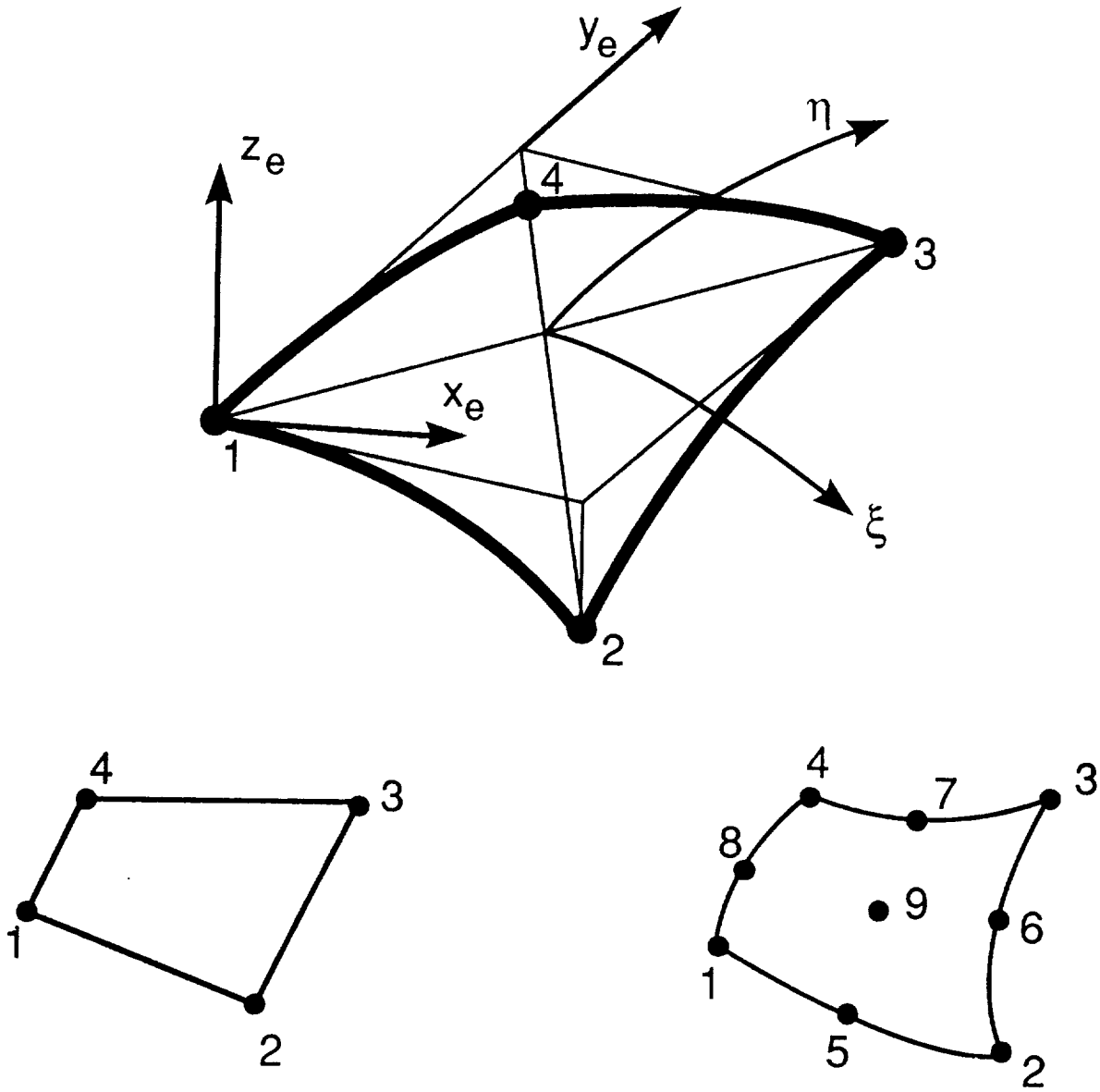


Figure 3.3 Conventions for Standard 2-D Elements.

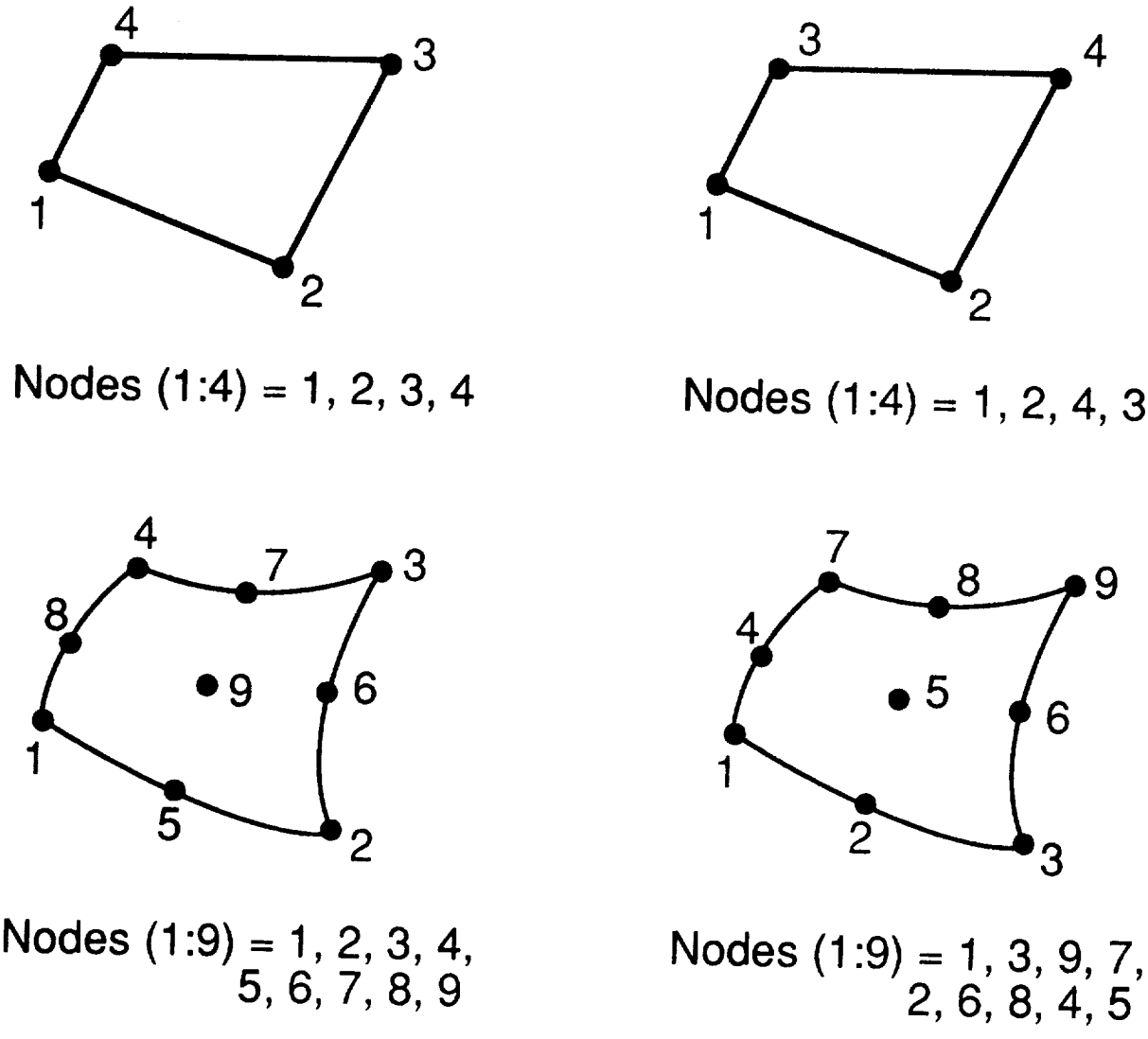


Figure 3.4 Examples of 2-D Elements with Non-Standard Node Numbering.

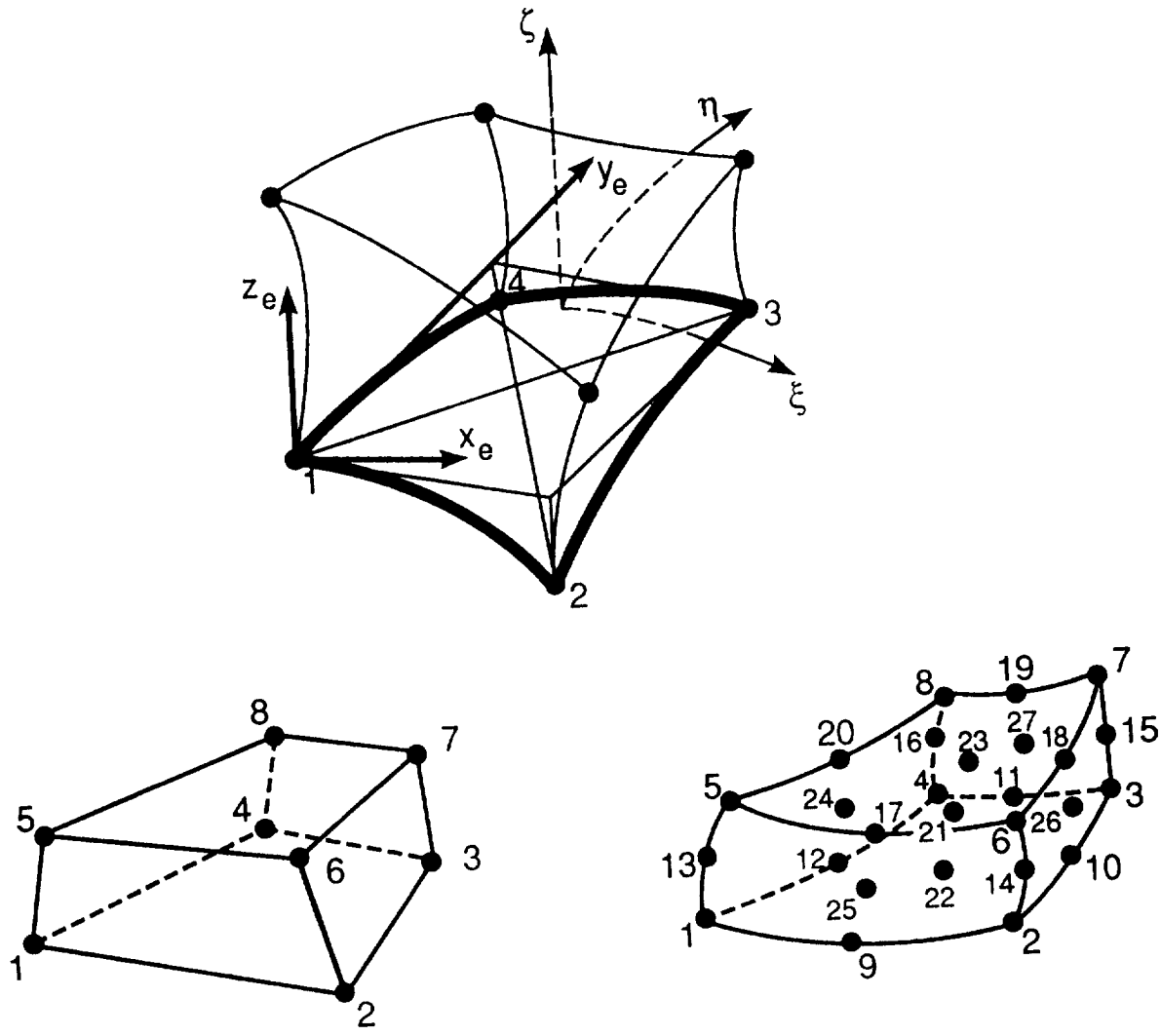
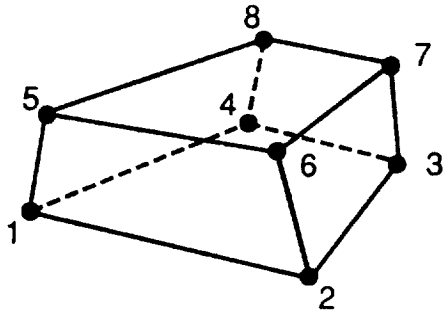
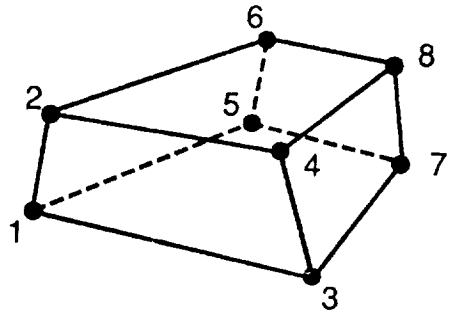


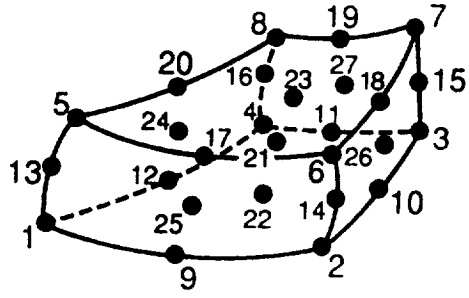
Figure 3.5 Conventions for Standard 3-D Elements.



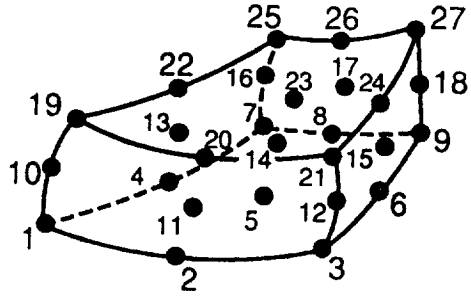
Nodes (1:8) = 1, 2, 3, 4,
5, 6, 7, 8



Nodes (1:8) = 1, 5, 2, 6,
4, 8, 3, 7



Nodes (1:27) = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27



Nodes (1:27) = 1, 9, 2, 12, 22, 10, 4, 11,
3, 13, 26, 14, 24, 21, 25, 16, 27,
5, 17, 6, 20, 23, 18, 8, 19

Figure 3.6 Examples of 3-D Elements with Non-Standard Node Numbering.

4. COROTATIONAL INTERFACE

CHAPTER OUTLINE

Section	*Title*	*Description*
4.1	**Overview**	Introduces users and developers to basic concepts of the corotational algorithm employed by the generic ES processor for handling geometric nonlinearity and intrinsic element behavior.
4.2	**Basic Corotational Theory**	Summarizes underlying mathematics of corotation, including the subtraction of large rigid-body rotations from element motion, the updating of nodal rotation triads to represent the current configuration uniquely, and the derivation of the rigid-body projection operator which may improve element behavior for both linear and nonlinear analysis.
4.3	**Built-in Corotational Options**	Summarizes how users and developers can avail themselves of the built-in corotational features of the generic ES processor. Describes the various low-order and high-order corotational options built into ES processors using the generic shell, including the *projection* option, which can be selected by the element developer as an automatic element feature.
4.4	**The Corotational Algorithm**	Step-by-step outline of how the corotational theory is implemented in the Testbed.
4.5	**Corotational Software Utilities**	Usage documentation is provided on each of the low-level corotational (CR) utilities currently employed by the ES processor shell.

4.1 Overview

Corotation is a method initially developed for making large rotations relative to an inertial frame *look* like small rotations at the element level. This is achieved by defining an element reference coordinate frame for each element that rotates with the element — as defined by its nodal coordinates. The rigid body motion of this frame is then "subtracted" from the total motion of the nodes, leaving *relative* translations and rotations that can be made arbitrarily small (for small strains) by simply refining the mesh, *i.e.*, by adding more element frames. For the rotational degrees of freedom, "subtracting" rigid body rotations really amounts to multiplication of orthogonal matrices (see Fig. 4.1), and the step-to-step updating of these matrices (or nodal triads) is done in a manner that preserves orthogonality.

Once nodal relative motions have been rendered sufficiently small, they may be used in simplified strain-displacement relations. For example, for shell elements, either linear strain-displacement relations or so-called moderate rotation nonlinear strain-displacement relations may be used in conjunction with the corotational procedure. The effect of using nonlinear versus linear element strain-displacement relations usually amounts to an increase in accuracy, which can alternately be achieved by adding more elements.

Corotation is not a new idea (references 6 and 7 and others were some of the innovators) but some new developments have been made (see refs. 8 and 9). First, the corotational methodology has been made more *element-independent* by generalizing it to beam, shell and solid elements and generic software utilities have been developed for all three classes. Second, a firm mathematical foundation has been established by deriving it from variational principles and identifying its association with a *projection* operator. By capitalizing on this projection operator idea, an element-independent, higher-order stiffness matrix has been derived through consistent linearization of the variational functional that ensures quadratic convergence in the context of a Newton-Raphson nonlinear solution procedure.

A surprising fringe benefit is that the projection matrix that emerges from the consistent linearization process corrects rigid body errors (*e.g.*, warping sensitivity) even for linear analysis (see Figure 4.2). This may have a profound effect on various old and new shell, or shell-oriented, element formulations (*e.g.*, certain hierarchically ($p-$) refined elements

which have implicit rigid body errors due to a higher-order representation of geometry than of displacement).

The following sections describe the basic corotational theory (Section 4.2), the built-in corotational options currently available in the Testbed using the generic structural-element (ES) processor (Section 4.3), and finally the algorithmic details (Section 4.4) and software utilities (Section 4.5) used to implement corotation in the Testbed.

4.2 Basic Corotational Theory

There are three essential ingredients to the corotational theory. The first, and perhaps most basic, ingredient is the *subtraction of large overall rigid-body motion* from the element displacement field. For beam and shell elements*, this results in *relative* displacements which are typically small enough to be used in conjunction with a moderate (or even small) rotation formulation of the element strain-displacement relations regardless of the size of total rotations. Furthermore, provided that the strains are small, the relative rotations can be rendered smaller and smaller by increasing the number of elements in a given region.**

The second ingredient of the corotational theory is the *rotation update algorithm* used to keep track of total rotations at the nodes. This procedure is necessary for computing the relative rotations described above, and can also be useful in specifying boundary conditions and graphically representing the deformed configuration.

The third ingredient of corotational theory is the so-called *projection operator* which emanates from consistently taking the first variation of the strain energy, expressed in terms of relative displacements, to obtain the nonlinear equilibrium equations (*i.e.*, the internal force vector). The projection operator also plays a prominent role in the form of the tangent stiffness matrix (or second variation), which is obtained by consistent linearization of the equilibrium equations. Finally, the projection operator so obtained corrects element rigid-body defects, *e.g.*, due to warping and non-isoparametric kinematics — even for linear analysis.

The following subsections summarize the mathematical theory associated with each of the above corotational ingredients.

---

* For solid continuum elements, the advantages of corotation are minimal except in cases where such elements are used to model shell structures. In such cases, corotation may be used in place of nonlinear strain-displacement relations, which may be convenient for the element developer — for example, in initial check-out of the element implementation.

** For finite strain analysis, the corotational formulation is not quite as useful, since the subtraction of rigid-body motion may yield relative rotations that are still large. Nevertheless, it may alleviate round-off errors when finite strains are superposed on large rigid-body motions, such as in flexible space-structure dynamics.

4.2.1 Subtraction of Large Rigid-Body Element Rotations

The objective here is to extract relative, or deformational, displacements at the element level which can be used in small- or moderate-strain/displacement relations, by subtracting the motion of a single rotation triad attached to each element. Referring to Figure 4.1, first the deformational translation vector and rotation matrix are defined at element node a, expressed in a fixed, global coordinate system, by

$$\begin{aligned} \mathbf{u}_a^{def} &= \mathbf{u}_a^{tot} - \mathbf{u}_a^{rig} \\ \mathbf{R}_a^{def} &= \mathbf{R}_a^{tot}[\mathbf{R}_a^{rig}]^T = \exp(\boldsymbol{\theta}_a^{def}) \end{aligned} \tag{4.1}$$

respectively, where the rigid-body components are defined as follows:

$$\begin{aligned} \mathbf{u}_a^{rig} &= \mathbf{u}_1^{tot} + (\mathbf{R}_a^{rig} - \mathbf{I})(\mathbf{x}_a^0 - \mathbf{x}_1^0) \\ \mathbf{R}^{rig} &= \mathbf{E}[\mathbf{E}^0]^T \end{aligned} \tag{4.2}$$

In the above expressions, $\mathbf{E}$ is the *element triad*, which corotates with the element (see Section 3.4 for its definition for specific element classes in terms of the element nodal coordinates), $\mathbf{x}$ is the position vector relative to a global origin, the superscript 0 denotes the initial configuration, and the subscript 1 denotes element node 1 — which is the origin of the element coordinate system attached to the element triad $\mathbf{E}$. Moreover, the total displacements are defined as follows (see Figure 4.1):

$$\begin{aligned} \mathbf{u}_a^{tot} &= \mathbf{x}_a - \mathbf{x}_a^0 \\ \mathbf{R}_a^{tot} &= \mathbf{S}_a[\mathbf{S}_a^0]^T \end{aligned} \tag{4.3}$$

where $\mathbf{S}_a$ is the nodal *surface triad*, which rotates with the structure at node a.

Finally, it is desirable to transform the deformational displacements derived above into the current element coordinate system, defined by the element triad, $\mathbf{E}$; and for the rotational components, to convert the rotation matrix to a rotation pseudo-vector that is an appropriate measure for conventional element strain-displacement relations. Thus, the computations for the relative translation vector are:

$$\boxed{\{\mathbf{u}_a^{def}\}_e = \mathbf{E}^T\,\mathbf{u}_a^{def}} \tag{4.4}$$

and for the relative rotation "vector":

$$\boxed{\{\boldsymbol{\theta}_a^{def}\}_e = \mathbf{E}^T\,\text{axial}\{\ln(\mathbf{R}_a^{def})\}} \tag{4.5}$$

where $\mathbf{ln(R)}$ is the natural logarithm of the matrix $\mathbf{R}$, which is a skew-symmetric matrix derived by taking the inverse of the exponential in equation (4.1), and axial{} denotes the axial vector corresponding to this latter matrix, *i.e.*,

$$\text{axial}\{\boldsymbol{\Omega}\} = \text{axial}\left\{\begin{bmatrix} 0 & -\Omega_3 & \Omega_2 \\ \Omega_3 & 0 & -\Omega_1 \\ \Omega_2 & \Omega_1 & 0 \end{bmatrix}\right\} = \begin{Bmatrix} \Omega_1 \\ \Omega_2 \\ \Omega_3 \end{Bmatrix}$$

An exact expression for the natural logarithm of a matrix is given in reference 9.

4.2.2 Updating of Large Nodal Rotations

An incremental nodal rotation update is used to obtain the total rotation matrix, $\mathbf{R}_a^{tot}$, appearing in equation (4.1). However, instead of updating $\mathbf{R}_a^{tot}$ directly, the nodal surface triad, $\mathbf{S}_a$, for the $k+1$ iteration is updated as follows:

$$\boxed{[\mathbf{S}^{k+1}]^a = \mathbf{exp}(\Delta\boldsymbol{\theta}_a^{tot})\,[\mathbf{S}^k]^a} \tag{4.6}$$

where an exact expression for the exponential of a matrix (Rodriguez' formula) is given later in equation (4.27). Once the nodal triad has been updated, the total rotation (connecting the initial and current nodal surface triads) is given by equation (4.3). Note that the incremental rotation update equation (4.5) is usually performed from one nonlinear solution iteration to the next, so that $\Delta\boldsymbol{\theta}_a^{tot}$ represents the iterative change in the rotational components of displacement, and appears explicitly in the iterative solution vector generated by the conventional Newton-Raphson solution algorithm.

4.2.3 Modification of Element Force and Stiffness (Projection)

The final ingredient of the corotational theory is more subtle than the two described previously. It arises from first substituting the expressions for the deformational displacements given by equation (4.1) into the element strain energy, and then consistently taking the first and second variations. The first variation leads to the element internal force vector, and the second variation, which is equivalent to consistent linearization, leads to the element stiffness matrix. Interestingly, by simply using the definitions introduced in equation (4.1), a corotational *projection* matrix, which subtracts incremental rigid-body motion, is also engendered by the first variation. This projection matrix modifies the basic element

internal-force and stiffness expressions by automatically correcting elements which do not initially satisfy the required zero rigid-body straining condition. Thus, dramatic improvements can be obtained for many conventional elements; for example, shell elements that are sensitive to warping, or any non-isoparametric elements which use higher-order geometrical approximations (*e.g.*, trigonometric) than displacement approximations (*e.g.*, low-order polynomials). Additionally, by incorporating the corotational theory in the derivation of the tangent stiffness matrix, and using consistent linearization, the optimal (quadratic) convergence rates are obtained when solving the nonlinear global equation system using a full Newton-Raphson iteration scheme.

A simple derivation of the projection operator can be made as follows (for details, refer to reference 9). First, express the element strain energy in conventional terms as:

$$U = \frac{1}{2} \int_V \epsilon^T \mathbf{C} \epsilon \, dV \tag{4.7}$$

where, ϵ, is the strain tensor, $\mathbf{C}$ is the constitutive tensor (assuming elastic behavior for simplicity) and V is the element volume. Now, expressing the element strains in terms of the *deformational* displacements only — which were derived in Section 4.2.1 — the strain-displacement relations can be written as:

$$\epsilon = \mathbf{B} \, \mathbf{d}_{def} \tag{4.8}$$

where $\mathbf{B}$ is the element strain-displacement matrix, and $\mathbf{d}_{def}$ may be partitioned as*

$$\mathbf{d}_{def} = \begin{Bmatrix} \mathbf{d}_{def}^1 \\ \mathbf{d}_{def}^2 \\ \vdots \\ \mathbf{d}_{def}^{nen} \end{Bmatrix} \tag{4.9}$$

where *nen* is the number of element nodes, and the deformational displacement vector at a particular node, a, is typically partitioned (for a beam or shell element) as:

* Note that $\mathbf{B}$ may itself be a function of $\mathbf{d}_{def}$, *i.e.*, if nonlinear element strain-displacement relations are used.

$$\mathbf{d}^a_{def} = \begin{Bmatrix} \mathbf{u}^a_{def} \\ \boldsymbol{\theta}^a_{def} \end{Bmatrix} \tag{4.10}$$

Next, the element internal force vector is obtained by taking the first variation of the strain energy with respect to *total* displacements which, by the chain rule, amounts to:

$$\mathbf{f}^{int} = \frac{\partial U}{\partial \mathbf{d}_{tot}} = [\frac{\partial \mathbf{d}_{def}}{\partial \mathbf{d}_{tot}}]^T \frac{\partial U}{\partial \mathbf{d}_{def}} \tag{4.11}$$

Thus,

$$\mathbf{f}^{int} = \mathbf{T}^T \bar{\mathbf{f}}^{int} \tag{4.12}$$

where

$$\bar{\mathbf{f}}^{int} = \frac{\partial U}{\partial \mathbf{d}_{def}} \tag{4.13}$$

and

$$\boxed{\mathbf{T} = \frac{\partial \mathbf{d}_{def}}{\partial \mathbf{d}_{tot}} = \mathbf{H}\mathbf{P}} \tag{4.14}$$

In the above expression, $\mathbf{P}$ is a *projection* matrix and has the property that

$$\mathbf{P}^2 = \mathbf{P} \tag{4.15}$$

and $\mathbf{H}$ is a *higher-order* matrix that reduces to the identity matrix for linear analysis, and can be approximated by the identity matrix in most situations.

Remark 4.1 The higher-order matrix, $\mathbf{H}$, derives from the subtle difference between the deformational rotation pseudo-vector, $\boldsymbol{\theta}_{def}$, which may be moderately large, and its incremental counterpart, $\delta\boldsymbol{\theta}_{def}$, which is considered infinitesimal. This difference appears when equation (4.14) is expanded into its translational and rotational parti-

tions, and then use the chain rule:

$$\begin{aligned}\left[\frac{\partial \mathbf{d}_{def}}{\partial \delta \mathbf{d}_{tot}}\right] &= \begin{bmatrix} \frac{\partial \mathbf{u}_{def}}{\partial \delta \mathbf{u}_{tot}} & \frac{\partial \mathbf{u}_{def}}{\partial \delta \boldsymbol{\theta}_{tot}} \\ \frac{\partial \boldsymbol{\theta}_{def}}{\partial \delta \mathbf{u}_{tot}} & \frac{\partial \boldsymbol{\theta}_{def}}{\partial \delta \boldsymbol{\theta}_{tot}} \end{bmatrix} \\ &= \begin{bmatrix} \frac{\partial \mathbf{u}_{def}}{\partial \delta \mathbf{u}_{tot}} & \mathbf{0} \\ \frac{\partial \boldsymbol{\theta}_{def}}{\partial \delta \boldsymbol{\theta}_{def}} \frac{\partial \delta \boldsymbol{\theta}_{def}}{\partial \delta \mathbf{u}_{tot}} & \frac{\partial \boldsymbol{\theta}_{def}}{\partial \delta \boldsymbol{\theta}_{def}} \frac{\partial \delta \boldsymbol{\theta}_{def}}{\partial \delta \boldsymbol{\theta}_{tot}} \end{bmatrix} \\ &= \underbrace{\begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \frac{\partial \boldsymbol{\theta}_{def}}{\partial \delta \boldsymbol{\theta}_{def}} \end{bmatrix}}_{\mathbf{H}} \underbrace{\begin{bmatrix} \frac{\partial \mathbf{u}_{def}}{\partial \delta \mathbf{u}_{tot}} & \mathbf{0} \\ \frac{\partial \delta \boldsymbol{\theta}_{def}}{\partial \delta \mathbf{u}_{tot}} & \mathbf{I} \end{bmatrix}}_{\mathbf{P}} \end{aligned} \tag{4.16}$$

where the notation of equation (4.13) has been refined by replacing $\mathbf{d}_{tot}$ by $\delta \mathbf{d}_{tot}$, to emphasize that the differentiation is with respect to the incremental quantities, which constitute the actual unknowns in the problem.

Note that the projection matrix, **P**, can be computed explicitly from equation (4.13), as shown in reference 9 and the Testbed implementaton follows this approach.

Finally, the element stiffness matrix is obtained by taking the second variation of the strain energy, equation (4.7), *i.e.*,

$$\mathbf{K} = \frac{\partial^2 U}{\partial \mathbf{d}_{tot} \partial \mathbf{d}_{tot}} = \frac{\partial \mathbf{f}^{int}}{\partial \mathbf{d}_{tot}} = \mathbf{T}^T \overline{\mathbf{K}} \mathbf{T} + \widehat{\mathbf{K}}(\mathbf{f}^{int}) \tag{4.17}$$

where

$$\overline{\mathbf{K}} = \frac{\partial^2 U}{\partial \mathbf{d}_{def} \partial \mathbf{d}_{def}} \tag{4.18}$$

is the basic element tangent stiffness matrix, and

$$\widehat{\mathbf{K}} = \frac{\partial \mathbf{T}^T}{\partial \mathbf{d}_{tot}} \bar{\mathbf{f}}^{int} \tag{4.19}$$

is a higher-order stiffness matrix arising from the need to differentiate the matrix **T**. This matrix ($\widehat{\mathbf{K}}$) may also be viewed as an extended geometric stiffness matrix.

In summary, the basic element force vector, $\bar{\mathbf{f}}^{int}$, and stiffness matrix, $\overline{\mathbf{K}}$, are transformed by **T**, and hence by the corotational projection matrix, **P**. The higher-order matrix, **H**, which premultiplies **P**, should be included for consistency, but is often unnecessary for practical applications. Additionally, a higher-order stiffness matrix, $\widehat{\mathbf{K}}$, emerges, which is an explicit function of the element internal force vector, $\bar{\mathbf{f}}^{int}$. Interestingly, $\widehat{\mathbf{K}}$, can be

substituted as an approximation to the geometric stiffness matrix for elements that do not provide their own intrinsic version.

From an implementation standpoint, it is interesting to note that all of the above modifications to the element force and stiffness arrays can be generated in an element-independent manner, *i.e.*, without knowing anything about the details of the element formulation (except of course the basic element type, *e.g.*, beam, shell, solid). This has made it possible to implement the corotational capabilities described herein as a built-in option for all elements associated with the Testbed's generic (ES) element processor.

4.3 Built-in Corotational Options

Three parameters are associated with the built-in corotational options that may be employed by ES processor users/developers. The first parameter, generically called "NL_GEOM", indicates whether geometric nonlinearity is present, and, if so, whether linear or nonlinear strain-displacement relations are to be used at the *element level*. The second parameter,"CORO", indicates whether corotation is to be employed to subtract out large rigid-body rotations for geometrically nonlinear analysis, and, if so, whether to employ an exact (consistent) or an approximate form for the element stiffness and internal force arrays (see Section 4.2.3). The last parameter, "PROJ", controls the use of the element corotational projection operator, which may improve the performance of some elements for linear as well as nonlinear analysis (also described in Section 4.2.3).

The mechanisms for setting these parameters — which are different and go by different names for users and developers (see Table 4.1) — and the various options allowed are described in the following sections. For clarity, geometrically linear and nonlinear analysis are discussed separately.

4.3.1 Geometrically Linear Analysis

For geometrically linear analyses, only one parameter related to corotation is relevant: the projection parameter ("PROJ"), which governs the use of the element projection operator. The other two parameters ("NL_GEOM" and "CORO") should be set to zero. This happens automatically if one of the standard linear analysis procedures, *e.g.*, L_STATIC or L_STABIL_1, is employed by the user.

4.3.1.1 The Geometric Nonlinearity Parameter (NL_GEOM)

For geometrically linear analyses, the macrosymbol ES_NL_GEOM should be set to 0 by the user before running the ES processor, using the directive:

```
*def/i ES_NL_GEOM = = 0
```

This value is currently the default setting within all ES processors. Note that this is equivalent to setting ES_NL_GEOM equal to zero in the call to the generic ES *procedure* (see Sections 2.3 and 2.4).

For element developers, the value of this parameter will be transmitted to the element kernel routines using the argument CTLS(pcNLG) — see Chapter 3.

4.3.1.2 The Corotation Parameter (CORO)

For geometrically linear analysis, the corotation parameter is irrelevant also. However, to be explicit, the user may wish to type the directive:

$$*\text{def/i ES\_CORO} = = 0$$

which is the default value. This action is equivalent to setting ES_CORO equal to zero in the call to the generic ES *procedure* (see Sections 2.3 and 2.4).

For element developers, the value of this parameter will be transmitted to the element kernel routines (Chapter 3) using the argument CTLS(pcCORO).

4.3.1.3 The Projection Parameter (PROJ)

For geometrically linear analysis, the projection parameter may be used to modify the element internal force vector, stiffness matrix, and displacements by multiplication by the projection matrix (see Section 4.2.3). This is not necessary for elements which satisfy the zero rigid-body straining condition (or, more strongly, equilibrium) exactly. Hence, the projection parameter is currently both a user option and a developer option; *i.e.*, the user may turn it on, but the developer may choose to ignore it for certain element types. The user's mechanism for selecting the projection operator is the macrosymbol, ES_PROJ (see Section 2.3). The developer's mechanism for allowing or inhibiting projection is the kernel argument DEFS(pdPROJ), which is set in kernel routine ES0D (see Sections 3.2 and 3.3). The value set by the user for macrosymbol ES_PROJ is conveyed to the ES processor kernel through the input argument CTLS(pcPROJ). Then, in standard kernel routine ES0D, the developer may either set DEFS(pdPROJ) equal to CTLS(pcPROJ), or set DEFS(pdPROJ) independently of CTLS(pcPROJ), depending on the developer's judgment.

The user will have to read the element developer's documentation (under the appropriate element processor section in Chapter 5 of the Testbed User's Manual, ref. 4) to determine how the ES_PROJ macrosymbol influences a particular element type. The value of DEFS(pdPROJ) is stored in the database, in dataset ES.SUMMARY, record ES_PROJ.i, where i is the sequence number of the element processor.

Legitimate values for the projection operator macrosymbol ES_PROJ are:

0 – No projection.

1 – Project element internal force, displacements, and material stiffness matrix, but not geometric stiffness matrix.

2 – Same as 1, and also project geometric stiffness matrix (relevant only for stability, analyses).

For linear analysis, projection of the element displacement, internal force and stiffness arrays means multiplication with the projection matrix as follows:

$$\begin{aligned} \mathbf{d} &= \mathbf{P}\,\overline{\mathbf{d}} \\ \mathbf{f}^{int} &= \mathbf{P}^T\,\overline{\mathbf{f}}^{int} \\ \mathbf{K}^{matl} &= \mathbf{P}^T\,\overline{\mathbf{K}}^{matl}\mathbf{P} \\ \mathbf{K}^{geom} &= \mathbf{P}^T\,\overline{\mathbf{K}}^{geom}\mathbf{P} + \hat{\mathbf{K}}(\mathbf{f}^{int}) \end{aligned} \tag{4.20}$$

where the barred quantities refer to the unprojected values. See Section 4.2.3 for more details.

The benefits of corotational projection for shell elements in linear analysis are shown in Figure 4.2, where various shell-element types are used to model the classical pinched cylinder problem, analyzed with a progressively distorted mesh. Notice that without projection most of these elements display serious degradation as the mesh is distorted (and warped); whereas, when projection is turned on, some of these elements become virtually insensitive to the mesh distortion.

4.3.2 Geometrically Nonlinear Analysis

4.3.2.1 The Geometric Nonlinearity Parameter (NL_GEOM)

For geometrically nonlinear analyses, the macrosymbol ES_NL_GEOM should be set to either 1 or 2 by the user before running the ES processor, using the directive:

*def/i ES_NL_GEOM = = { 1 | 2 }

This action is equivalent to setting ES_NL_GEOM equal to 1 or 2 in the call to the generic ES *procedure* (Section 2.3).

The difference between options 1 (first-order nonlinearity) and 2 (second-order nonlinearity) is as follows:

1 - The problem is geometrically nonlinear, but that linear strain-displacement relations are to be used at the element level. This option is meaningful only if corotation has been selected using the "CORO" parameter (Section 4.3.2.2).

2 - The problem is geometrically nonlinear, and that nonlinear strain-displacement relations are to be used at the element level. If corotation is also selected, this will improve accuracy; if not, the element may only be valid for moderately large rotations — depending on the element's local nonlinear formulation.

The element developer recovers the value of this parameter through the standard element kernel-routine argument CTLS(pcNLG) — see Section 3.3.

4.3.2.2 The Corotation Parameter (CORO)

For geometrically nonlinear analyses (ES_NL_GEOM = 1 or 2), the corotation parameter is used to invoke the corotational update algorithm described in Section 4.2. This parameter is set by the user using the macrosymbol ES_CORO,

*def/i ES_CORO = = { 0 | 1 | 2 }

where the values have the following meaning:

0 – Do not perform corotational updates (*i.e.*, do not subtract rigid body motions), and approximate **H** by **I** in equation (4.14).

1 – Perform corotational updates (*i.e.*, subtract rigid body motions), and approximate **H** by **I** in equation (4.14).

2 – Perform corotational updates, and use the exact representation of **H** in equation (4.14).

For element developers, the value of this parameter will be transmitted to the standard element kernel routines (Chapter 3) through the argument CTLS(pcCORO).

4.3.2.3 The Projection Parameter (PROJ)

For geometrically nonlinear analyses, the projection parameter has essentially the same meaning as for linear analyses, except that the projection matrix, **P**, is now a function of the displacements, and the material and geometric stiffness matrices are first added to form the tangent stiffness before projection is applied. Also, the displacements are not projected when they are large, but instead are subjected to the nonlinear corotational algorithm for subtracting rigid-body motion described in Section 4.2.1.

Legitimate values for the projection parameter are:

0 – No projection.

1 – Project element internal force and tangent stiffness matrix, but not the geometric stiffness matrix.

2 – Same as 1, and also project the geometric stiffness matrix (relevant only for stability analyses, *i.e.*, buckling eigenvalue analysis about a nonlinear prestress state).

Projection of the element internal force and stiffness arrays means multiplication with the projection matrix, as follows:

$$\begin{aligned} \mathbf{f}^{int} &= \mathbf{P}^T \mathbf{H}^T \bar{\mathbf{f}}^{int} \\ \mathbf{K}^{tang} &= \mathbf{P}^T \mathbf{H}^T \bar{\mathbf{K}}^{tang} \mathbf{H}\mathbf{P} + \hat{\mathbf{K}}(\mathbf{f}^{int}) \\ \mathbf{K}^{geom} &= \mathbf{P}^T \mathbf{H}^T \bar{\mathbf{K}}^{geom} \mathbf{H}\mathbf{P} + \hat{\mathbf{K}}(\mathbf{f}^{int}) \end{aligned} \tag{4.21}$$

where the barred quantities refer to the unprojected values, and

$$\bar{\mathbf{K}}^{tang} = \bar{\mathbf{K}}^{matl} + \bar{\mathbf{K}}^{geom} \tag{4.22}$$

In the above expressions, $\mathbf{H} = \mathbf{I}$ if the "CORO" parameter is less than or equal to one, and is different from the identity matrix only if the "CORO" parameter is set to two. See Section 4.2.3 for more details.

Furthermore, the geometric stiffness matrix, $\mathbf{K}^{geom}$, is relevant as a separate entity only for stability analysis — in this case about a nonlinear prestress state.

4.3.3 Summary of Options

The corotational options described in the preceding subsections are summarized in Tables 4.1 - 4.3. Table 4.1 gives the respective user and developer names associated with the generic parameters: NL_GEOM, CORO, and PROJ. Table 4.2 shows the consequences of setting each of these parameters to 0 (off), 1 (low) or 2 (high). Finally, Table 4.3 shows how the element displacement, internal force and stiffness arrays are modified as a function of the values of the PROJ, CORO and NL_GEOM parameters.

Table 4.1 Corotational Parameter Names

Parameter Type (Generic Name))	*User Parameter Name (Macrosymbol Name)*	*Developer Parameter Name (Subroutine Argument)*
Geometric Nonlinearity (NL_GEOM)	**ES_NL_GEOM**	**CTLS(pcNLG)**
Corotational Updates (CORO)	**ES_CORO**	**CTLS(pcCORO)**
Projection Operator (PROJ)	**ES_PROJ**	**CTLS(pcPROJ)*** **DEFS(pdPROJ)***

* CTLS(pcPROJ) is input and DEFS(pdPROJ) is output in subroutine ES0D.

Table 4.2 Corotational Parameter Values

Parameter	*Parameter Value*		
(Generic)	0	1	2
NL_GEOM	Linear Analysis	Lin. Strain-Disp. Relns	Nonlin. Strain-Disp. Relns
CORO	Non-Corotational	Corotation ($\mathbf{H} = \mathbf{I}$)	Corotation ($\mathbf{H} \neq \mathbf{I}$)
PROJ	No Projection	Project all but $\mathbf{K}^{geom}$	Project all incl. $\mathbf{K}^{geom}$

Table 4.3 Effect of Projection on Element Arrays

Linear Analysis (ES_NL_GEOM = ES_CORO = 0)			
Array	*PROJ = 0*	*PROJ = 1*	*PROJ = 2*
$\mathbf{d}_{def}$	$\bar{\mathbf{d}}$	$\mathbf{P}\,\bar{\mathbf{d}}$	$\mathbf{P}\,\bar{\mathbf{d}}$
$\mathbf{f}^{int}$	$\bar{\mathbf{f}}^{int}$	$\mathbf{P}^T\bar{\mathbf{f}}^{int}$	$\mathbf{P}^T\bar{\mathbf{f}}^{int}$
$\mathbf{K}^{matl}$	$\overline{\mathbf{K}}^{matl}$	$\mathbf{P}^T\overline{\mathbf{K}}^{matl}\mathbf{P}$	$\mathbf{P}^T\overline{\mathbf{K}}^{matl}\mathbf{P}$
$\mathbf{K}^{geom}$	$\overline{\mathbf{K}}^{geom}$	$\overline{\mathbf{K}}^{geom}$	$\mathbf{P}^T\overline{\mathbf{K}}^{geom}\mathbf{P}+\hat{\mathbf{K}}(\mathbf{f}^{int})$
Nonlinear Analysis (ES_NL_GEOM $\geq$ 0 ; ES_CORO = 1)			
Array	*PROJ = 0*	*PROJ = 1*	*PROJ = 2*
$\mathbf{d}_{def}$	$\mathcal{F}(\bar{\mathbf{d}})$	$\mathcal{F}(\bar{\mathbf{d}})$	$\mathcal{F}(\bar{\mathbf{d}})$
$\mathbf{f}^{int}$	$\bar{\mathbf{f}}^{int}$	$\mathbf{P}^T\bar{\mathbf{f}}^{int}$	$\mathbf{P}^T\bar{\mathbf{f}}^{int}$
$\mathbf{K}^{tang}$	$\overline{\mathbf{K}}^{tang}$	$\mathbf{P}^T\overline{\mathbf{K}}^{tang}\mathbf{P}+\hat{\mathbf{K}}(\mathbf{f}^{int})$	$\mathbf{P}^T\overline{\mathbf{K}}^{tang}\mathbf{P}+\hat{\mathbf{K}}(\mathbf{f}^{int})$
$\mathbf{K}^{geom}$	$\overline{\mathbf{K}}^{geom}$	$\overline{\mathbf{K}}^{geom}$	$\mathbf{P}^T\overline{\mathbf{K}}^{geom}\mathbf{P}+\hat{\mathbf{K}}(\mathbf{f}^{int})$
Nonlinear Analysis (ES_NL_GEOM $\geq$ 0 ; ES_CORO = 2)			
Array	*PROJ = 0*	*PROJ = 1*	*PROJ = 2*
$\mathbf{d}_{def}$	$\mathcal{F}(\bar{\mathbf{d}})$	$\mathcal{F}(\bar{\mathbf{d}})$	$\mathcal{F}(\bar{\mathbf{d}})$
$\mathbf{f}^{int}$	$\mathbf{H}^T\bar{\mathbf{f}}^{int}$	$\mathbf{P}^T\mathbf{H}^T\bar{\mathbf{f}}^{int}$	$\mathbf{P}^T\mathbf{H}^T\bar{\mathbf{f}}^{int}$
$\mathbf{K}^{tang}$	$\mathbf{H}^T\overline{\mathbf{K}}^{tang}\mathbf{H}$	$\mathbf{P}^T\mathbf{H}^T\overline{\mathbf{K}}^{tang}\mathbf{H}\mathbf{P}+\hat{\mathbf{K}}(\mathbf{f}^{int})$	$\mathbf{P}^T\mathbf{H}^T\overline{\mathbf{K}}^{tang}\mathbf{H}\mathbf{P}+\hat{\mathbf{K}}(\mathbf{f}^{int})$
$\mathbf{K}^{geom}$	$\mathbf{H}^T\overline{\mathbf{K}}^{geom}\mathbf{H}$	$\mathbf{H}^T\overline{\mathbf{K}}^{geom}\mathbf{H}$	$\mathbf{P}^T\mathbf{H}^T\overline{\mathbf{K}}^{geom}\mathbf{H}\mathbf{P}+\hat{\mathbf{K}}(\mathbf{f}^{int})$

Remark 4.2 $\mathcal{F}(\bar{\mathbf{d}})$ is an abstract functional notation used to denote the subtraction of rigid-body motion from the element displacement field, and corresponds to the nonlinear algorithm described in Section 4.2.1.

Remark 4.3 If PROJ = 2 and $\overline{\mathbf{K}}^{geom} = 0$, then $\mathbf{K}^{geom} \longrightarrow \hat{\mathbf{K}}(\mathbf{f}^{int})$, which can be used as a crude approximation to the geometric stiffness for elements whose geometric stiffness routine (ES0KG) has not yet been implemented.

4.4 The Corotational Algorithm

The following steps constitute the "element-independent" corotational algorithm currently implemented in the Testbed for geometrically nonlinear analysis. These steps roughly correspond to the algorithm originally implemented in the STAGS code (ref. 8). Additional theoretical details are presented in references 8 and 9, and references therein.

Step 1 Initialization. Construct the initial element corotational frame, $\mathbf{E}^0$, based on initial element nodal coordinates, $\{\mathbf{x}_g^0\}^a$, where g denotes the global Cartesian basis, e denotes the element basis and a is the element node index (see Figs. 3.1 and 3.3). Thus, $\mathbf{E}$ is an orthogonal 3×3 matrix, or triad, whose columns are unit vectors pointing in the directions of the element corotational axes, x_e, y_e, z_e. The element corotational coordinate system is defined in much the same way as the conventional SPAR element coordinate systems, except that for quadrilateral elements whose corner nodes do not all lie in a plane, an *average* plane is selected, as described in reference 8. Note that there is some element *type* dependency in that beam, plate and (curved) shell elements require different frame-selection conventions, but there are only a small number of such cases to distinguish.

Step 2 Frame Update. Update the element corotational frame at each load-step or iteration, to obtain $\mathbf{E}^k$ as a function of the *current* element nodal coordinates, $\{\mathbf{x}_g^k = \mathbf{x}_g^0 + \mathbf{u}_g^k\}^a$, where k is the current step or iteration counter, and $\mathbf{u}$ is the translational part of the nodal displacement vector. Note that $\mathbf{E}^k$ is defined in precisely the same manner as $\mathbf{E}^0$ except using the *current* set of nodal coordinates.

Step 3 Relative Translations. Compute *relative-translations* for element node a, $\{\mathbf{u}_e^k\}^a$, with respect to the updated corotational frame, and expressed in the basis, $\mathbf{E}^k$. That is,

$$\{\mathbf{u}_e^k\}^a = \{\mathbf{x}_e^k\}^a - \{\mathbf{x}_e^0\}^a \tag{4.23}$$

where

$$\{\mathbf{x}_e^0\}^a \equiv (\mathbf{E}^0)^T (\{\mathbf{x}_g^0\}^a - \{\mathbf{x}_g^0\}^1) \tag{4.24}$$

$$\{\mathbf{x}_e^k\}^a \equiv (\mathbf{E}^k)^T (\{\mathbf{x}_g^k\}^a - \{\mathbf{x}_g^k\}^1) \tag{4.25}$$

in which $\{\mathbf{x}_g^0\}^1$ and $\{\mathbf{x}_g^k\}^1$ are the initial and current coordinates, respectively, of the *origin* of the element corotational coordinate system (typically taken as element node 1). The relative translations $\{\mathbf{u}_e^k\}^a = \{u_e^a, v_e^a, w_e^a\}^T$ may be input directly to the element strain routine; but for beam/plate/shell elements, a corresponding set of relative *rotations* are needed first.

Step 4 Relative Rotations. Compute *relative-rotations*, $\{\boldsymbol{\theta}_e^k\}^a$, with respect to the updated corotational frame. *This step is crucial.* It is composed of three substeps. First, nodal rotation triads, $[\mathbf{S}]^a$, defined to be tangent to the deformed surface

at each node, are updated using the incremental rotation components obtained from the solution of the linearized equilibrium equations. Next, a relative-rotation *matrix*, $[\mathbf{R}_{def}]^a$, representing the *deformational* part of the total rotation is computed at each element node. Finally, $[\mathbf{R}_{def}]^a$ is converted to an approximate rotation *pseudo-vector*, $\{\boldsymbol{\theta}_e^k\}^a = \{\,\theta_x^a, \theta_y^a, \theta_z^a\,\}_e^T$. This pseudo-vector is the corotational equivalent of the rotation vector used in conventional small- or moderate-rotation beam/shell analysis. Together, $\{\boldsymbol{\theta}_e^k\}^a$ and $\{\mathbf{u}_e^k\}^a$ (from Step 3), complete the element nodal displacement vector, $\{\mathbf{d}_e^k\}^a$, which may then be input to a standard (*e.g.*, linear) element strain routine.

SubStep 4.1: <u>Global Rotation Updates</u>. This sub-step is actually performed at the *global* level, *i.e.*, *before* looping on individual elements. At each node, the transformation matrix that defines the nodal DOF directions is used as the initial *surface triad*, $[\mathbf{S}^0]^a$. The nodal surface triads are then updated at each load-step/iteration using the recursion formula

$$[\mathbf{S}^{k+1}]^a = \mathbf{Q}(\Delta\boldsymbol{\theta}^a)[\mathbf{S}^k]^a \tag{4.26}$$

where k is the iteration counter, $\Delta\boldsymbol{\theta}_a$ is the incremental nodal rotation vector emanating from the iterative solution vector (*i.e.*, the rotational components of $\Delta\mathbf{d}$) and $\mathbf{Q}$ is an incremental rotation matrix defined by Rodriguez's formula for the exponential of a matrix, *i.e.*,

$$\mathbf{Q}(\Delta\boldsymbol{\theta}) = \mathbf{I} + \frac{sin\,|\Delta\boldsymbol{\Theta}|}{|\Delta\boldsymbol{\Theta}|}\,\Delta\boldsymbol{\Theta} + \frac{1}{2}\left(\frac{sin\,|\Delta\boldsymbol{\Theta}/2|}{|\Delta\boldsymbol{\Theta}|/2}\right)^2 \Delta\boldsymbol{\Theta}^2 \tag{4.27}$$

in which $\Delta\boldsymbol{\Theta}$ is the skew-symmetric matrix corresponding to the vector $\Delta\boldsymbol{\theta} = \{\,\Delta\theta_1, \Delta\theta_2, \Delta\theta_3\,\}^T$, *i.e.*

$$\Delta\boldsymbol{\Theta} = \begin{bmatrix} 0 & -\Delta\theta_3 & \Delta\theta_2 \\ & 0 & -\Delta\theta_1 \\ skew & & 0 \end{bmatrix} \tag{4.28}$$

In practice, the triad update formula in equation (4.26) involving $[\mathbf{S}^k]^a$ and $[\mathbf{S}^{k+1}]^a$ is performed entirely in terms of equivalent rotation pseudo-vector, $\{\mathbf{p}^k\}^a$ and $\{\mathbf{p}^{k+1}\}^a$, so that only 3 numbers, rather than 9, need be stored at each node (*e.g.*, see ref. 8).

SubStep 4.2: <u>Element Relative-Rotation Matrix</u>. The relative, or *deformational*, rotation matrix at an element node is then constructed by "subtracting" the rigid-body contribution of the corotational frame, $\mathbf{E}^k$, from the total rotation of the nodal surface triads, $[\mathbf{S}^k]^a$, as follows. The total rotation of an arbitrary unit vector, $\{\hat{\mathbf{x}}\}^a$, attached to the surface at element node a is by definition:

$$\{\hat{\mathbf{x}}_{tot}^k\}^a = [\mathbf{R}_{tot}]^a\,\{\hat{\mathbf{x}}^0\}^a \qquad \text{where: } \boxed{[\mathbf{R}_{tot}]^a = [\mathbf{S}^k]^a\,([\mathbf{S}^0]^a)^T} \tag{4.29}$$

and the *rigid-body* part of the total rotation is defined by the motion of the corotational frame, *i.e.*,

$$\{\hat{\mathbf{x}}_{rig}^k\}^a = \mathbf{R}_{rig}\,\{\hat{\mathbf{x}}^0\}^a \text{ where: } \boxed{\mathbf{R}_{rig} = \mathbf{E}^k\,(\mathbf{E}^0)^T} \tag{4.30}$$

Thus, if the total rotation is expressed as the composition of rigid-body and deformational parts

$$\boxed{[\mathbf{R}_{tot}]^a = [\mathbf{R}_{def}]^a \, \mathbf{R}_{rig}} \tag{4.31}$$

then the deformational part can be expressed as

$$\boxed{[\mathbf{R}_{def}]^a = [\mathbf{R}_{tot}]^a \, \mathbf{R}_{rig}^T = [\mathbf{S}^k]^a \, ([\mathbf{S}^0]^a)^T \, \mathbf{E}^0 \, (\mathbf{E}^k)^T} \tag{4.32}$$

Finally, it is convenient to *transform* $[\mathbf{R}_{def}]^a$ from the global basis to the current *element* corotational basis using

$$[\mathbf{R}_{def}]_e^a = (\mathbf{E}^k)^T \, [\mathbf{R}_{def}]^a \, \mathbf{E}^k \tag{4.33}$$

SubStep 4.3: Element Relative-Rotation Pseudo-Vector. By using the *inverse* of the triad-update formula in equation (4.27), an approximate rotation pseudo-vector, $\boldsymbol{\theta}_e^a$, corresponding to $[\mathbf{R}_{def}]_e^a$ may be computed at each element node, a. Thus, solving

$$[\mathbf{R}_{def}]_e^a \;=\; \mathbf{Q}(\boldsymbol{\theta}_e^a) \tag{4.34}$$

for the *skew-symmetric matrix* corresponding to $\boldsymbol{\theta}_e^a$ yields:

$$\boldsymbol{\Theta}_e^a = \frac{sin^{-1}(|\mathbf{Z}|)}{|\mathbf{Z}|}\,\mathbf{Z}, \qquad \mathbf{Z} = \frac{([\mathbf{R}_{def}]_e^a - [\mathbf{R}_{def}]_e^{aT})}{\sqrt{1 + \text{trace}[\mathbf{R}_{def}]_e^a}} \tag{4.35}$$

from which the three vector components of $\boldsymbol{\theta}_e^a$ can be extracted according to equation (4.28).

Step 5 Strain Computation. Given the element nodal displacement vectors, $\mathbf{d}_e^a = \{\, \mathbf{u}_e^a, \boldsymbol{\Theta}_e^a \,\}^T$ — which are assumed to be "small" since they are measured relative to the corotational frame — standard element linear (infinitesimal-rotation) or nonlinear (moderate-rotation) strain-displacement routines may be employed to compute strains.

Step 6 Stress Computation. Either linear elastic or nonlinear (*e.g.*, elastic-plastic) constitutive relations can be used in conjunction with the strains obtained in Step 5 to compute element stresses. Additional information such as deformation gradients, would be needed to accommodate *finite strains*.

Step 7 Internal-Force/Stiffness Computation. Finally, the element internal-force and stiffness arrays are formed with respect to the *current* element corotational frame, $\mathbf{E}^k$, and are then transformed to the *computational* or nodal degree of freedom bases — *e.g.*, global Cartesian or shell-oriented — using the nodal submatrix transformations:

$$\mathbf{K}_c^{ab} = (\mathbf{T}_{ec}^a)^T \, \mathbf{K}_e^{ab} \, \mathbf{T}_{ec}^b \,, \qquad \mathbf{f}_c^a = (\mathbf{T}_{ec}^a)^T \, \mathbf{f}_e^a \tag{4.36}$$

where a, b denote element node numbers, and e, c denote the current element and computational bases, respectively. The transformation matrix, $\mathbf{T}^a_{ec}$, is obtained by the composition

$$\mathbf{T}^a_{ec} = \mathbf{E}^T \mathbf{T}^a_{gc} \tag{4.37}$$

where $\mathbf{T}^a_{gc}$ is the orthogonal transformation between the computational basis and the global Cartesian basis at node a.* The element arrays in equation (4.14) are ready for assembly into the corresponding *system* arrays.

Remark 4.4 The projection operations described in Section 4.2.3 are performed (optionally) on the element stiffness and internal force arrays *before* performing the transformations indicated in equation (4.32). For linear analysis, projection is the *only* corotational aspect of the solution algorithm (see Table 4.3).

* It is possible to use either a fixed or a *moving* computational basis at the nodes, *i.e.*, one that follows the surface — like $[\mathbf{S}]^a$ (see Step 4.1). However, in the present Testbed implementation, a *fixed* basis is assumed.

4.5 Corotational Software Utilities (CR*)

The following subroutine entry points (where the first two letters of each subroutine name are CR) are employed by the generic structural-element (ES) processor shell to perform all of the corotational functions described in the preceding sections. These utilities are invoked *automatically* by the processor shell; hence, there is no requirement that the element developer become familiar with them. They are described here only for the curious reader, and/or the researcher who wishes to call these utilities directly for some exotic element formulation.

Table 4.1-1 SUMMARY OF COROTATIONAL SOFTWARE UTILITIES

Utility Name	*Function*
CRDEFD	Computes deformational rotation *matrices*, $\mathbf{R}_{def}$, at element nodes. The rotation "vectors" generated by CRDEFR are extracted from these orthogonal matrices according to equation (4.34). This routine is for elements that can use the rotation matrices directly (*e.g.*, C^0 shell elements) to achieve greater accuracy.
CRDEFR	Computes deformational (strain-producing) nodal rotation "vectors", $\{\theta^a_{def}\}_e$, relative to element corotational frame.
CRDEFT	Computes deformational (strain-producing) nodal translation vectors, $\{\mathbf{u}^a_{def}\}_e$, relative to the element corotational frame.
CRFIC1	Modifies the internal-force vector, $\mathbf{f}^{int}$, for C^1 *shell* and *beam* elements, to improve consistency of the "first variation" with respect to the corotational hypothesis, using the matrix **H** in equation (4.21).
CRKC1	Modifies the tangent-stiffness matrix, **K**, for C^1 *shell* and *beam* elements, to obtain consistent linearization with respect to the corotational hypothesis; using the matrix **H** in equation (4.21).
CRPDEF	*Projects* the element displacement vector, $\overline{\mathbf{d}}^e$, using the matrix **P** in equation (4.20), to obtain deformationally consistent displacements, $\mathbf{d}^e_{def}$, for linear analysis only.
CRPRFB	Same as CRPROF, but for beam elements.
CRPRKB	Same as CRPROK, but for beam elements.
CRPROF	*Projects* the element internal-force vector, using the matrix **P** in equations (4.20) and (4.21), to eliminate the differential rigid-body motion of the element corotational frame. This can improve element performance for elements that do not satisfy rigid body invariance in advance.

Table 4.1-1 SUMMARY OF COROTATIONAL SOFTWARE UTILITIES

Utility Name	*Function*
CRPROK	*Projects* the element stiffness matrix (for plate/shell elements), using the matrix **P** in equations (4.20) and (4.21), to eliminate the differential rigid-body motion of the element corotational frame. Also, adds the effect of the *derivative* of the projection matrix in the consistent linearization of the element internal force vector, to obtain the higher order stiffness matrix: $\widehat{\mathbf{K}}(\mathbf{f}^{int})$.
CRTGE	Constructs/updates the element corotational frame (or triad), **E**, for plate, shell and solid elements.
CRTGEB	Updates the element corotational triad for beam elements.

4.5.1 Subroutine CRDEFD: Deformational Rotation Vector

Subroutine CRDEFD computes the orthogonal matrix at element nodes representing the deformational (strain-producing) part of nodal rotations.

Calling Sequence

call CRDEFD (e0, s0, psk, ek, d)

Input Arguments

E0(3,3)	Initial element triad, $\mathbf{E}^0$.
S0(3,3)	Initial surface triad at node a, $[\mathbf{S}^0]^a$.
PSK(3)	Current total-rotation pseudo-vector, $\boldsymbol{\theta}$, at node a, corresponding to the rotation of the surface triad relative to the initial configuration, *i.e.*, $\mathbf{R}_{tot} = \mathbf{S}\,[\mathbf{S}^0]^T$.
EK(3,3)	Current (or initial if NDOF = 0) element triad, $\mathbf{E}$.

Output Arguments

D(3,3)	Current deformational rotation matrix at node a. (Mathematical notation: $\mathbf{R}^a_{def}$)

4.5.2 Subroutine CRDEFR: Deformational Rotation Vector

Subroutine CRDEFR computes deformational (strain-producing) nodal rotation "vectors" relative to element corotational frame.

Calling Sequence

call CRDEFR (ndof, e0, s0, psk, ek, omega)

Input Arguments

NDOF	Number of *rotatational* degrees of freedom per node; currently must be equal to three, such that the rotational degrees of freedom correspond to $\theta^e_x, \theta^e_y, \theta^e_z$.
E0(3,3)	Initial element triad, $\mathbf{E}^0$.
S0(3,3)	Initial surface triad at node a, $[\mathbf{S}^0]^a$.
PSK(3)	Current total-rotation pseudo-vector, $\boldsymbol{\theta}$, at node a.
EK(3,3)	Current (or initial if NDOF = 0) element triad, $\mathbf{E}$.

Output Arguments

OMEGA(3)	Current deformational rotation "vector" at element node a. (Mathematical notation: $\boldsymbol{\theta}^e_a$).

4.5.3 Subroutine CRDEFT: Deformational Translations

Subroutine CRDEFT computes the *translational* part of the deformational (strain-producing) nodal displacement vectors for beam, shell or solid elements.

Calling Sequence

call CRDEFT (ndof, xg, xe, ek, ug, ix, nen, w, udef)

Input Arguments

NDOF	Total number of degrees of freedom per node (3 or 6). If NDOF=0, then UG is ignored and XE is output; otherwise, UG and XE are both used to compute UDEF. (NOTE: If NDOF> 0, then NDOF must be at least three, such that the first three degrees of freedom are the translations, u, v, w.)
XG(3,NEN)	Table of initial global coordinates at element nodes; XG(i, a) = the ith global coordinate of element node a, where $i = 1, 2, 3$ corresponds to x, y, z, and $a = 1$ — NEN, where NEN is the number of element nodes. (Mathematical notation: x_i^a)
XE(3,NEN)	Table of initial element-based coordinates at element nodes; XE(i, a) defines the ith element-based coordinate of element node a, where $i = 1, 2, 3$ corresponds to x_e, y_e, z_e, and $a = 1$ — NEN, where NEN is the number of element nodes. XE is input only if NDOF>0. (Mathematical notation: $x_{ia}^{(e)}$)
EK	Current (or initial if NDOF = 0) element triad (3×3). (Mathematical notation $\mathbf{E}^k$)
UG(NDOF,NEN)	Current global displacements at element nodes; UG(i,a) = $\{\mathrm{d}_i^a\}_e$; the index i ranges from 1 to NDOF and, if NDOF=6, corresponds to $u, v, w, \theta_x, \theta_y, \theta_z$. (Irrelevant if NDOF=0.)

IX(NEN) Element node number list index array giving *counterclockwise* order of nodes, corner nodes listed *first*. IX(a) is the number of the node that is the ath corner node in the standard counterclockwise ordering. Additional nodes can come in any order after the corner nodes. This array is used to establish the correspondence between the developer's arbitrary numbering scheme and the required ordering. (Note: triangles are treated just like quadrilaterals.)

NEN Number of element nodes represented in IX array.

W(6*NEN) Workspace array of at least six times NEN words.

Output Arguments

UDEF(NDOF,NEN) Current deformational translations at element nodes: UDEF(i,a) = $\{u_i^a\}_e^{def}, i = 1,3$. Note: If NDOF > 0, UDEF is dimensioned NDOF by NEN and rows 4—NDOF are unaffected by the call; however, if NDOF = 0, then UDEF is instead interpreted as the initial element-based coordinates at element nodes, *i.e.*, $\{\mathbf{x}_i^a\}_e^0$, and is dimensioned three by NEN.

4.5.4 Subroutine CRFIC1: C^1 Shell Element Force Correction

Subroutine CRFIC1 modifies the internal force vector for C^1 shell elements to improve consistency of the "first variation" with respect to the corotational hypothesis. This corresponds to premultiplication by the matrix, $\mathbf{H}^T$, as defined in equation (4.21).

Calling Sequence

call CRFIC1 (nen, udef, fi, hfi)

Input Arguments

NEN	Number of element nodes with rotational degrees of freedom. Each affected node is assumed to have three translational and three rotational degrees of freedom.
UDEF(6,NEN)	Current deformational displacements at element nodes: $u, v, w, \theta_x, \theta_y, \theta_z$ in turn for each node $a = 1$—NEN.
FI(6,NEN)	The internal force vector (first variation of the strain energy) ordered in the same way as UDEF.

Output Arguments

HFI(6,NEN)	Modified internal force vector, *i.e.*, premultiplied by the higher-order matrix, **H**. Ordered in the same way as UDEF and FI. (May be stored in the same array as FI)

4.5.5 Subroutine CRKC1: C^1 Shell Element Stiffness Correction

Subroutine CRKC1 modifies the element stiffness matrix for C^1 shell elements to obtain consistent linearization with respect to the corotational hypothesis. This corresponds to a congruent transformation by the matrix $\mathbf{H}^T$, as defined in equation (4.21).

Calling Sequence

call CRKC1 (nen, istab, udef, fi, k, w, hkh)

Input Arguments

NEN	Number of nodes in element with rotational degrees of freedom. Each node is assumed to have three translational and three rotational degrees of freedom. This subroutine will not work with elements with midside "deviational" nodes that have a nonstandard degree of freedom pattern.
ISTAB	If ISTAB = 0, compute modified *tangent* stiffness; otherwise, compute modified stability matrix.
UDEF(6,NEN)	Current deformational displacements at element nodes: $u, v, w, \theta_x, \theta_y, \theta_z$ in turn for each node a = 1—NEN.
FI(6,NEN)	Internal force vector (first variation of strain energy)) ordered like UDEF. This is the same FI as *input* to subroutine CRFIC1.
K(NSIZE)	The element stiffness matrix (in local element coordinates) stored in *lower triangle* form. NSIZE is equal to 6×NEN×(6×NEN+1)/2.
W(18*NEN)	Workspace.

Output Arguments

HKH(NSIZE)	Modified element stiffness matrix ordered in the same way as K. Obtained as $\mathbf{H}^T\overline{\mathbf{K}}\mathbf{H}$. (May be stored in the same array as K).

4.5.6 Subroutine CRPDEF: Project Displacements

Subroutine CRPDEF projects out spurious rigid-body contributions to the element displacement vector — for linear analysis only — by premultiplication by the matrix **P** in equation (4.20). This operation has an effect only for elements that do not satisfy the zero rigid-body strain requirement intrinsically.

Calling Sequence

call CRPDEF (nen, ndof, inode, ix, xe, u, uproj)

Input Arguments

NEN	Number of element nodes. Each node must currently have three translational and three rotational degrees of freedom.
NDOF	Number of degrees of freedom per node; currently must be equal to six.
INODE	If INODE=0, then the local x axis lies along the 1—2 line projected on the element plane. This choice corresponds to NODE = 2 and IXY = 1 in subroutine CRTGE. If INODE = 1, the y axis lies along the 1—4 line projected on the element plane. This choice corresponds to NODE = 4 and IXY = 2 in subroutine CRTGE.
IX(NEN)	Element node number index array giving *counterclockwise* order of nodes, corner nodes listed *first*. IX(a) is the number of the node that is the ath corner node in the standard counterclockwise ordering. Additional nodes can come in any order after the corner nodes. This array is used to establish the correspondence between the developer's arbitrary numbering scheme and the required ordering. Note: triangles are treated just like any other element.

XE(3,NEN) Table of initial element-based coordinates at element nodes; XE(i,a) defines the ith element-based coordinate of element node a, where $i = 1,2,3$ corresponds to x_e, y_e, z_e, and a =1—NEN, where NEN is the number of element nodes. (Mathematical notation: $x_{ia}^{(e)}$)

U(NDOF,NEN) Unprojected displacements at element nodes: $u, v, w, \theta_x, \theta_y, \theta_z$ in turn for each node a = 1—NEN, expressed in the element basis.

Output Arguments

UPROJ(ndof,nen) Projected element displacement vector. (May be the same array as U.)

4.5.7 Subroutine CRPRFB: Project Internal-Force for Beam

Subroutine CRPRFB *projects* the modified element internal force vector to eliminate the differential rigid-body motion of the element corotational frame and enforces self-equilibrium of element internal forces when the element does not initially satisfy this condition (required by patch test). This does not affect elements whose internal forces are self-equilibrated.

Calling Sequence

call CRPRFB (nen, nlin, xe, udef, e0, s0, psk, ek, fi, fiproj)

Input Arguments

NEN	Number of nodes in element. Each node has three translational and three rotational degrees of freedom.
NLIN	Set NLIN to zero for linear analysis. For beam formulations now being considered, it is unnecessary to call subroutine CRPRFB for linear analyses.
XE(3,NEN)	Table of initial element-based coordinates at element nodes; XE(i,a) defines the ith element-based coordinate of element node a, where $i = 1,2,3$ corresponds to x_e, y_e, z_e, and a =1—NEN, where NEN is the number of element nodes. (Mathematical notation: $x_{ia}^{(e)}$)
UDEF(6,NEN)	Current deformational displacements at element nodes: $u, v, w, \theta_x, \theta_y, \theta_z$ in turn for each node a = 1—NEN.
E0(3,3)	Initial element triad, $\mathbf{E}^0$.
S0(3,3)	Initial surface triad at node 1, *i.e.*, $[\mathbf{S}^0]^1$.
PSK(3)	Current total-rotation pseudo-vector, $\boldsymbol{\theta}$, at element node 1.

EK(3,3)	Current element triad, **E**.
FI(NDOF,NEN)	Element internal force vector. For C^1 elements, use the output of CRFIC1. Ordered the same way as UDEF.

Output Arguments

FIPROJ(NDOF,NEN)	Projected internal force vector satisfying infinitesimal rotational invariance. Arranged in the same way as argument FI. (FI and FIPROJ may be stored in the same array.)

4.5.8 Subroutine CRPRKB: Project Stiffness for Beam

Subroutine CRPRKB computes corrections to the element stiffness matrix in local element coordinates, arising from the differential motion of the element corotational frame. This step completes the consistent linearization for the element-independent corotational method for beam elements.

Calling Sequence

call CRPRKB (nen, nlin, istab, xe, udef, e0, s0, psk, ek, fiproj, k, kproj)

Input Arguments

NEN	Number of element nodes. Each node has three translational and three rotational degrees of freedom.
NLIN	Set to zero for linear analysis. For beam formulations now being considered subroutine CRPRKB need not be called for linear analyses.
ISTAB	Stability flag. Set to zero only if the linear (material) stiffness matrix is being projected by itself. Otherwise, set to one to indicate that the geometric stiffness matrix is included, and hence to add the extended geometric stiffness, $\hat{\mathbf{K}}(\mathbf{f}^{int})$.
XE(3,NEN)	Table of initial element-based coordinates at element nodes; XE(i,a) defines the ith element-based coordinate of element node a, where $i = 1,2,3$ corresponds to x_e, y_e, z_e, and a =1—NEN, where NEN is the number of element nodes. (Mathematical notation: $x_{ia}^{(e)}$)
UDEF(6,NEN)	Current deformational displacements at element nodes: $u, v, w, \theta_x, \theta_y, \theta_z$ in turn for each node $a = 1$—NEN.

E0(3,3)	Initial element triad, $\mathbf{E}^0$.
S0(3,3)	Initial surface triad at element node 1, *i.e.*, $\mathbf{S}_1^0$.
PSK(3)	Current total-rotation pseudo-vector at node 1, *i.e.*, $\boldsymbol{\theta}_1^0$.
EK(3,3)	Current element triad, $\mathbf{E}$.
FIPROJ(NDOF,NEN)	Internal forces from element. Use the output of subroutine CRPROF. Ordered the same way as UDEF.
K(NSIZE)	The element stiffness in local element coordinates. For C^1 elements, should be the output of subroutine CRKC1. Arranged in *lower triangle* order. NSIZE is equal to NDOF×NEN×(NDOF×NEN+1)/2.

Output Arguments

KPROJ(NSIZE)	Projected stiffness matrix. Arranged in the same way as K. (May even be the same array as K.)

4.5.9 Subroutine CRPROF: Project Internal-Force Vector

Subroutine CRPROF *projects* the element internal force vector (after it has optionally been modified using subroutine CRFIC1) to eliminate the differential rigid-body motion of the element corotational frame and enforces self-equilibrium of element internal forces when the element does not initially satisfy this condition (required by a patch test). This does not affect elements whose internal forces are self-equilibrated.

Calling Sequence

call CRPROF (nen. ndof, inode, nlin, ix, xe, udef, fi, fiproj)

Input Arguments

NEN	Number of element nodes.
NDOF	Number of degrees of freedom per node. Degree of freedom pattern must be regular, *i.e.*, the same for each node. There must be three *translational* degrees of freedom per node, and three *rotational degrees of freedom* per node — *i.e.*, currently, NDOF must equal six.
INODE	If INODE=0, then the local x axis lies along the 1—2 line projected on the element plane. This choice corresponds to NODE = 2 and IXY = 1 in subroutine CRTGE. If INODE = 1, the y axis lies along the 1—4 line projected on the element plane. This choice corresponds to NODE = 4 and IXY = 2 in subroutine CRTGE.
NLIN	Flag set to zero for linear analysis and set to one for geometrically nonlinear analysis. Determines whether the displacements, UDEF, will be used in the construction of the projection matrix, **P**. For linear analysis, only the initial configuration is used; for nonlinear analysis the displacements are used to update the configuration.

IX(NEN) — Element node number index array giving *counterclockwise* order of nodes, corner nodes listed *first*. IX(a) is the number of the node that is the ath corner node in the standard counterclockwise ordering. Additional nodes can come in any order after the corner nodes. This array is used to establish the correspondence between the developer's arbitrary numbering scheme and the required ordering. Note: Triangles are treated just like quadrilaterals.

XE(3,NEN) — Table of initial element-based coordinates at element nodes; XE(i, a) defines the ith element-based coordinate of element node a, where $i = 1, 2, 3$ corresponds to x_e, y_e, z_e, and a =1—N, where N is the number of element nodes. (Mathematical notation: $\{x_i^a\}_e^0$)

UDEF(NDOF,NEN) — Current deformational displacements at element nodes: *e.g.*, if NDOF equals six, the degrees of freedom are $u, v, w, \theta_x, \theta_y, \theta_z$ for each node $a = 1$—NEN.

FI(NDOF,NEN) — Element internal force vector. For C^1 elements, use the output of subroutine CRFIC1. (Ordered the same way as argument UDEF.)

Output Arguments

FIPROJ(NDOF,NEN) — Projected internal force vector satisfying infinitesimal rotational invariance. Arranged in the same way as FI. (Note: FIPROJ and FI may be stored in the same array.)

4.5.10 Subroutine CRPROK: Project Stiffness Matrix

Subroutine CRPROK computes corrections to the element stiffness matrix (in local element coordinates) arising from the differential motion of the element corotational frame. This step completes the consistent linearization of the the element-independent corotational formulation; and will yield an exact tangent stiffness matrix in many cases (see ref. 8).

Calling Sequence

call CRPROK (nen, ndof, inode, nlin, istab, ix, xe, udef, fiproj, k, kproj)

Input Arguments

NEN	Number of element nodes.
NDOF	Number of degrees of freedoms per node. Degree of freedom pattern must be regular, *i.e.*, the same for each node. Currently, there must be three *translational* degrees of freedom and three *rotational degrees of freedom* per node, *i.e.*, NDOF = 6.
INODE	If INODE=0, then the local x axis lies along the 1—2 line projected on the element plane. This choice corresponds to NODE = 2 and IXY = 1 in subroutine CRTGE. If INODE = 1, the y axis lies along the 1—4 line projected on the element plane. This choice corresponds to NODE = 4 and IXY = 2 in subroutine CRTGE.
NLIN	Geometric nonlinearity flag. Set to zero for linear analysis and set to one for geometrically nonlinear analysis. Determines whether the displacements, UDEF, will be used in the construction of the projection matrix, **P**. For linear analysis, only the initial configuration is used; for nonlinear analysis the displacements are used to update the configuration. Additionally, NLIN = 1 triggers the generation

	of the extended geometric stiffness, $\widehat{\mathbf{K}}(\mathbf{f}^{int})$, regardless of the value of ISTAB, below.
ISTAB	Stability flag. Set to zero only if the linear (material) stiffness matrix is being projected by itself. Otherwise, set to one — to indicate that the geometric stiffness matrix is included, and hence to add the extended geometric stiffness, $\widehat{\mathbf{K}}(\mathbf{f}^{int})$.
IX(NEN)	Element node number list index array giving *counterclockwise* order of nodes, corner nodes listed *first*. IX(a) is the number of the node that is the ath corner node in the standard counterclockwise ordering. Additional nodes can come in any order after the corner nodes. This array is used to establish the correspondence between the developer's arbitrary numbering scheme and the required ordering. Note: triangles are treated just like quadrilateral element.
XE(3,NEN)	Table of initial element-based coordinates at element nodes; XE(i,a) defines the ith element-based coordinate of element node a, where $i = 1,2,3$ corresponds to x_e, y_e, z_e, and a =1—NEN, where NEN is the number of element nodes. (Mathematical notation: $\{x_i^a\}_e^0$)
UDEF(NDOF,NEN)	Current deformational displacements at element nodes: *e.g.*, if NDOF equals six, the degrees of freedom are $u, v, w, \theta_x, \theta_y, \theta_z$ for each node $a = 1$—NEN.
FIPROJ(NDOF,NEN)	Internal forces from element. Use the output of CRPROF. Ordered the same way as UDEF.

K(NSIZE) The element stiffness in local element coordinates. For C^1 elements, should be the output of subroutine CRKC1. Arranged in *lower triangular* order. NSIZE is equal to NDOF×NEN×(NDOF×NEN+1)/2. (May be zero if $\hat{\mathbf{K}}(\mathbf{f}^{int})$ is the only desired output.)

Output Arguments

KPROJ(NSIZE) Projected stiffness matrix. Arranged in the same way as argument K. (May even be stored in the same array as K.)

4.5.11 Subroutine CRTGE: Construct Element Triad.

Subroutine CRTGE computes the element corotational frame (or triad) **E** for plate, shell, and solid elements.

Calling Sequence

call CRTGE (ndof, xg, ix, node, ixy, ug, ek)

Input Arguments

NDOF — Number of degrees of freedom per node. If NDOF = 0, then the element displacements, UG, are assumed to be zero. (NOTE: If NDOF > 0, then it must currently be at least three, such that the first three degrees of freedom corresponding to the translational degrees of freedom: u, v, w.)

XG(3,NEN) — Table of initial global coordinates at element nodes; XG(i, a) defines the ith global coordinate of element node a, where $i = 1, 2, 3$ corresponds to x, y, z, and $a = 1$—NEN, where NEN is the number of element nodes. (Mathematical notation: x_i^a)

IX(4) — Element node number list index array giving *counterclockwise* order of nodes, corner nodes listed *first*. IX(a) is the number of the node that is the ath corner node in the standard counterclockwise ordering. Only the corner nodes need to be listed for subroutine CRTGE. For triangles, IX(3) = IX(4). This array is used to establish the correspondence between the developer's arbitrary numbering scheme and the required ordering.

NODE — Index in IX of corner node to which the local element x or y axis points towards. For example, if NODE = 2, the local x or y axis points to node IX(NODE) projected onto the local element frame.

IXY — Axis indicator. If IXY$\leq$1, the local x axis points to IX(NODE); otherwise, the local y axis points to IX(NODE). The two most common examples of the use of NODE and IXY is the combination [2,1] and [4,2]. In the first case, the local x axis points to node IX(2) (or the corner node to the right of the reference node, which is *always* node IX(1)). In the second case, the local y axis points to node IX(4), or the corner node above the reference node. For rectangular elements, this would mean that the x axis points along the 1—2 edge, the y axis along the 1—4 edge, with the z axis pointing *toward* the viewer for *both* of the above definitions.

UG(NDOF,NEN) — Current global displacements at element nodes; UG(i,a) = $\{\mathbf{d}_i^a\}_e$; the index i ranges from 1 to NDOF and, if NDOF=6, corresponds to $u, v, w, \theta_x, \theta_y, \theta_z$. Only the displacements corresponding to corner nodes (as indicated by IX) are employed by this routine. (Irrelevant if NDOF=0.)

Output Arguments

EK — Current (or initial if NDOF = 0) element triad (3 $\times$ 3). (Mathematical notation **E**)

4.5.12 Subroutine CRTGEB: Construct Element Triad for Beams

Subroutine CRTGEB computes the element corotational frame (or triad) **E** for beam elements.

Calling Sequence

call CRTGEB (ndof, xg, ixy, ug, e0, s0, psk, ek)

Input Arguments

NDOF — Number of degrees of freedom per node. NDOF = 0 implies that this is the initial pass.

XG(3,NEN) — Table of initial global coordinates of nodes at the endpoints of the beam; XG(i,a) defines the ith global coordinate of element node a, where $i = 1,2,3$ corresponds to x,y,z, and a = 1—NEN, where NEN is the number of element nodes used in this routine. If this is the initial pass, then the coordinates of a third (reference) node not lying along the line joining the endpoint nodes is expected. (Mathematical notation: x_i^a)

IXY — Controls *initial* selection of the beam y or z axis; has no effect on the update option of subroutine CRTGEB. If IXY$\leq$2, the initial trial y axis lies along the line directed from node 1 to the reference node (see description under argument XG). The z axis is constructed to be perpendicular to the plane containing the three nodes. The x axis always points from node 1 to node 2. The trial y axis is then orthogonalized to form a right-hand system. If IXY>2, the initial trial z axis points to the reference node, with the y axis constructed to be perpendicular to the plane containing the three nodes. A right-hand system is then similarly constructed by orthogonalization.

UG(NDOF,NEN) Current global displacements at element nodes; UG(i,a) = $\{\mathbf{d}_i^a\}_e^k$; the index i ranges from 1 to NDOF and, if NDOF=6, corresponds to $u, v, w, \theta_x, \theta_y, \theta_z$. (Irrelevant if NDOF=0.)

E0(3,3) Initial element triad, $\mathbf{E}^0$. (Irrelevant if NDOF = 0)

S0(3,3) Initial surface triad, $[\mathbf{S}^0]^1$, at element node 1.

PSK(3) Current total-rotation pseudo-vector, $\boldsymbol{\theta}$, at element node 1; corresponds to rotation matrix, $\mathbf{R}_{tot} = \mathbf{S}\,[\mathbf{S}^0]^T$.

Output Arguments

EK(3,3) Current (or initial if NDOF = 0) element triad, $\mathbf{E}$.

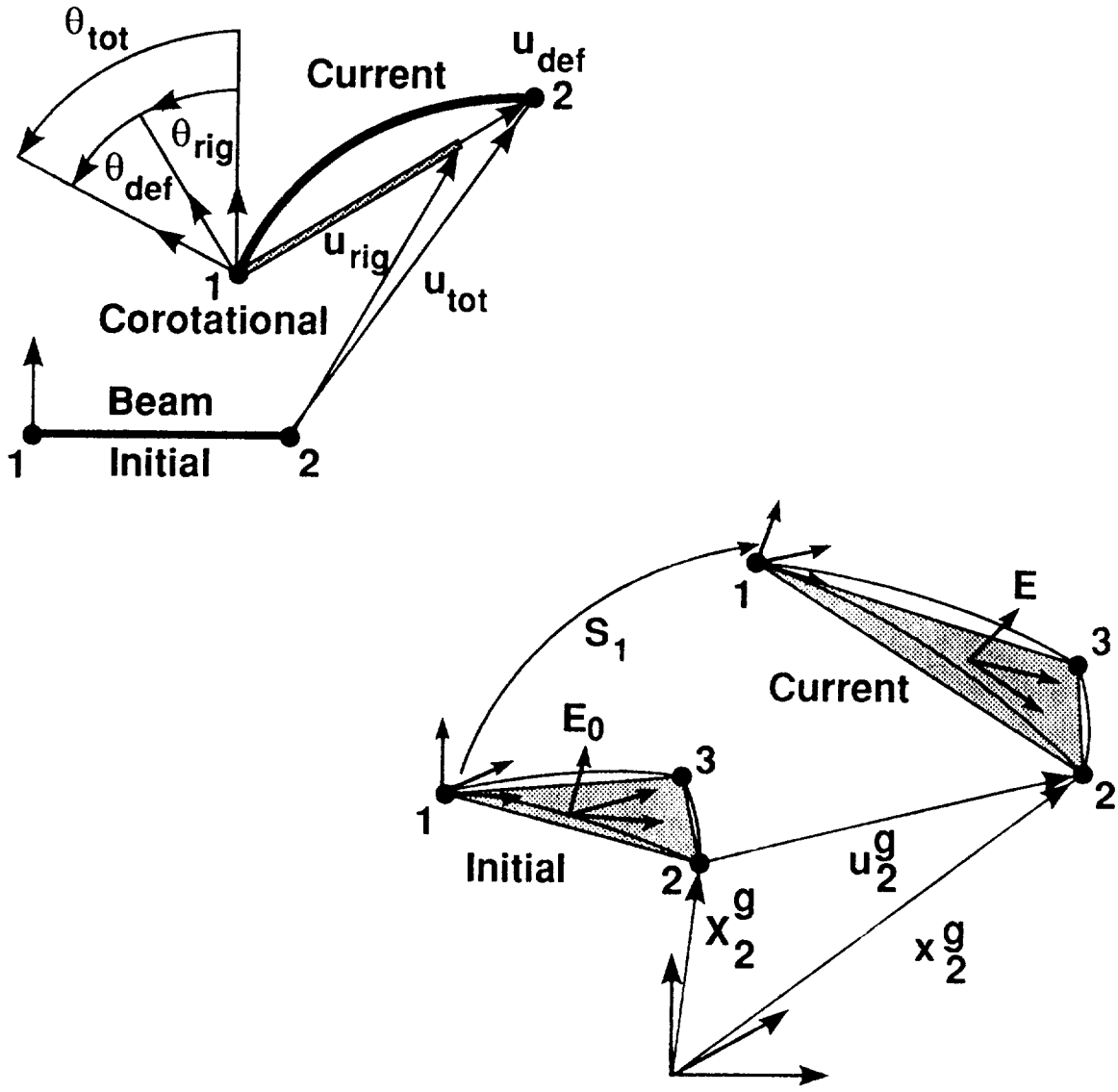


Figure 4.1 Corotational Description of Motion for 1-D and 2-D Elements.

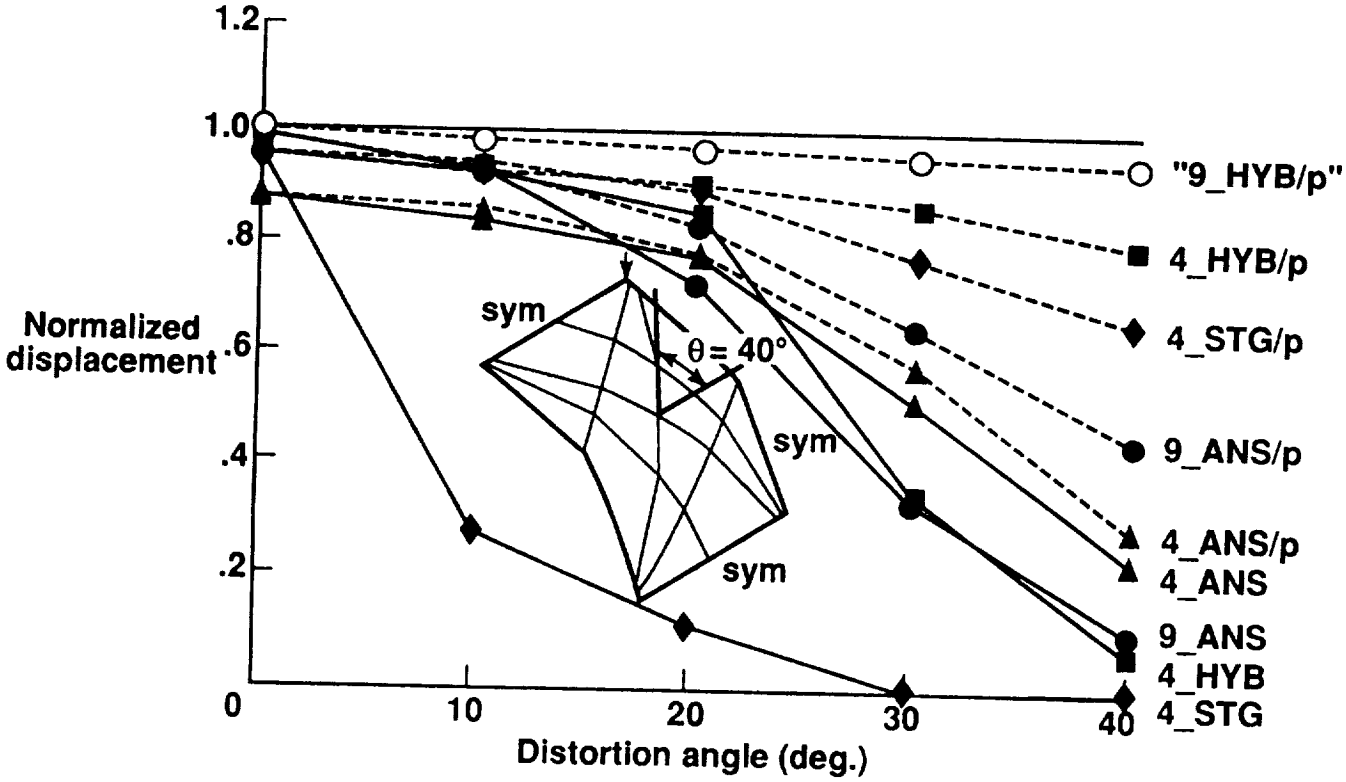


Figure 4.2 Effect of the Projection Operator for Pinched Cylinder with Distorted Mesh.

5. CONSTITUTIVE INTERFACE

CHAPTER OUTLINE

Section	*Title*	*Description*
5.1	**Overview**	Introduces users and developers to the various ways in which constitutive functions (*i.e.*, material definition, stress and tangent-modulus calculations required for element force and stiffness formation) may be performed using ES processors.
5.2	**User Interface**	Describes how user selects material fabrications and constitutive models, associates them with pertinent element types, and interprets stress output from ES processors.
5.3	**Developer Interface**	Describes various built-in ES options for performing constitutive (stress and tangent-modulus) calculations; as well as options for developers to provide their own.
5.4	**Proposed Interface: Generic Constitutive Processor**	Outlines general-purpose scheme (in-progress) for upgrading the generic ES processor to handle material nonlinearity and a growing library of constitutive (and failure) models; by interfacing the ES shell to a generic *constitutive* library — for "standard" structural element types only (see Section 3.4).

5.1 Overview

Constitutive functions for structural elements include calculation of element stresses (for the internal-force vector, geometric stiffness matrix and result output), tangent modulii (for the tangent stiffness matrix), and failure criteria (for result output and material degradation). Currently, the Generic Structural-Element (ES) processor performs only some of these functions automatically; the element developer must provide others. In particular, only *linear* constitutive models are currently implemented within the ES shell, and users and developers must perform their own failure analysis (see also processor LAU in the CSM Testbed User's Manual, ref. 4).

The constitutive functions that are currently performed by the ES processor shell rely on pre-processing of material/fabrication properties by processors LAU (for shell and solid elements) and LAUB (for beam elements). Thus Sections 5.2 and 5.3 explain the database connection between ES processors and processors LAU and LAUB, and indicate the various options that element developer have to exploit (or ignore) this built-in constitutive capability. Section 5.4 describes plans for interfacing with a generic constitutive processor, which will replace the current element/constitutive interface, and give all ES processors access to a growing array of advanced linear and nonlinear constitutive models.

5.2 User Interface

5.2.1 Materially Linear Analysis

To perform analysis with linear materials, the user has two options:

(1) Employ processor LAU (for shell and solid elements), or processor LAUB (for beam elements), to define material and fabrication properties; these properties are automatically converted into integrated constitutive matrices, and deposited in datasets PROP.BTAB.*; or

(2) Define your own PROP.BTAB.* datasets.

Option (1) is valid only for "standard" element types, as defined in Section 3.4 or in Chapter 5 of the CSM Testbed User's Manual documentation for specific ES processors (ref. 4). Option (2) is valid for both standard and "wild" element types, *i.e.*, for all elements implemented within ES processors. However, if option (2) is used for a standard element type, the user must be careful to define the PROP.BTAB.* dataset in a manner corresponding to the output of processors LAU or LAUB. This dataset is element-type dependent, and is described both under processors LAU and LAUB in the Testbed User's Manual (ref. 4), and in the Testbed Data Library Description (ref. 5).

In either case, the connection between the PROP.BTAB.* dataset and a particular set of elements is made through the element connectivity definition processor ELD (see Chapter 2). First, the EXPE command of processor ELD is used to define the *type* of PROP.BTAB.* dataset to be generated; and second, the NSECT command of processor ELD is used to point to the appropriate column of the PROP.BTAB.*, in case there is more than one set of section properties defined.

5.2.2 Materially Nonlinear Analysis

CURRENTLY NOT IMPLEMENTED

5.3 Developer Interface

5.3.1 Materially Linear Analysis

5.3.1.1 Standard Elements

Standard elements, as defined in Section 3.4 (*e.g.*, beams, shells and solids), can employ the built-in constitutive functionality of the ES processor shell to generate stresses and tangent-modulii — including all transformations between material and element stress coordinate systems. Thus, in order to compute stress, the element developer need only provide an element strain routine (ES0E) in most cases. There are exceptions, as described below. The other prerequisite for exploiting the built-in ES constitutive routines (for standard elements) is that an appropriate PROP.BTAB.* dataset is generated for each section property type (see the CSM Testbed Data Library Description, ref. 5, for details). As mentioned in Section 5.2, processors LAU (for shell and solid elements) and processor LAUB (for beam elements) usually provide the most convenient way to generate these datasets.

To select a built-in ES constitutive option, the element developer must set the output argument DEFS(pdCNS) to the appropriate value, from within subroutine ES0D (see Sections 3.2 and 3.3). Meaningful values of DEFS(pdCNS) are as follows:

DEFS(pdCNS) = 0: All stress and tangent-moduli calculations are performed by the ES shell; the developer must supply only a strain routine (ES0E).

DEFS(pdCNS) = 1: Tangent-moduli calculations are performed by the ES shell; the developer must supply a combined stress/strain routine, which uses the tangent-modulus (constitutive) matrix as input. This option is intended primarily for assumed stress type elements, for which stresses are computed directly, and strains are computed by inverting the constitutive matrix.

5.3.1.2 Wild Elements

Wild elements, as defined in Section 3.4, must perform their own constitutive calculations. However, the developer has two ways in which to perform this, which are selected using argument DEFS(pdCNS) from within subroutine ES0D:

DEFS(pdCNS) = 2: The developer provides a tangent-modulus (*i.e.*, constitutive matrix) routine, ES0C, and a combined stress/strain routine, ES0S; or:

DEFS(pdCNS) = 3: The developer provides 3 routines: a tangent-modulus (*i.e.*, constitutive matrix) routine, ES0C, a strain routine, ES0E, and a stress routine, ES0S.

The above 2 options are strictly up to the developer and are usually dictated by convenience for a particular element formulation.

In either case, all data required to construct the constitutive matrix for "wild" elements must be provided by the user in the PROP.BTAB.* dataset (see Section 5.2); and it is the developer's responsibility to explain to the user what is expected in that dataset. The best vehicle for that explanation is in the individual element processor's section of the CSM Testbed User's Manual (ref. 4).

5.3.2 Materially Nonlinear Analysis

CURRENTLY NOT IMPLEMENTED

5.4 Proposed Interface: Generic Constitutive Processor

The Generic Constitutive Processor (GCP) is a set of software modules which are designed to perform all constitutive functions for the Testbed. The purpose behind the design of the GCP is to create a flexible, easy to use framework for the testing and incorporation of new constitutive modeling capability into the Testbed. The GCP replaces and enhances the current linear elastic constitutive capability within the Testbed, as implemented within the Generic Element Processor (GEP).

Two functions will be served by the GCP; a stand-alone processor for use in testing of new constitutive models, and a FORTRAN callable constitutive library directly accessed by Testbed element processors. To enable this duality in function, the GCP is designed to perform constitutive calculations using input received through either an interface to the GEP during a finite element analysis, or from the GCP processor shell when operating in stand-alone mode.

From a method developers standpoint, the GCP will allow ease of access to constitutive functions by all element developers, and allow constitutive models incorporated by materials developers to be available to all elements implemented in the GEP framework. These capabilities have been included through standard developer interfaces. Through a standardized interface to the GCP, the constitutive model developer will have access to all of the higher level constitutive functions. Similarly the element developer is provided with an interface to the GCP allowing complete access to all computational constitutive functions.

The GCP design incorporates the current functional capability of the Testbed processors LAU and LAUB, *i.e.*, through the thickness integration for shells, calculation of elastic constitutive matrix coefficients, and evaluation of mass matrix coefficients. In addition, the GCP replaces the material property input currently performed using the Testbed processor TAB.

The intrinsic functions which the GCP performs are summarized below:

- User input of data, *i.e.*, material properties, laminate fabrications, and analysis parameters.
- Historical data initialization.
- Non-linear analysis.
- Laminate analysis.
- Material stiffness/flexibility calculation.
- Mass and damping property calculation.
- Stress/strain calculation.
- Point-stress/strain failure predictions.
- Historical data update.
- Post-processing of constitutive data.

Each of these functions may be performed at the material point or laminate (*e.g.*, thickness-integrated) levels.

The data created by the GCP functions mentioned above will be manipulated through a set of database utilities providing the user/developer transparent access to the data without knowledge of the database formats. Of primary concern in the design of the GCP is that of computational efficiency and flexibility of the system to be adapted to new requirements. Throughout the design these factors, although mutually exclusive at times, have been considered, and when possible a balance has been achieved between them which is thought to serve the best interests of the Testbed.

The GCP is designed with the flexibility to incorporate the functional capability currently in ADVLAM, Lockheed's primary composite laminate code. ADVLAM embodies linear/nonlinear constitutive capability and is applicable to large displacement/infinitesimal strain problems, where additive strain decomposition is appropriate. A summary of the

desirable capabilities to be included in the GCP is shown in Table 5.4-1. Note that each of the material constitutive and failure models shown exists as an independent module (*i.e.*, a subroutine package) in ADVLAM ensuring ease of incorporating new features within the GCP.

Table 5.4-1 – Summary of Potential GCP Capabilities

- Analysis:
  - — Laminate Properties
  - — Stress Analysis
  - — Strength Analysis
  - — Hygrothermal Analysis
- Constitutive Models (Temperature/Moisture-Dependent):
  - — Orthotropic Elastic
  - — Nonlinear Orthotropic Elastic
  - — Orthotropic Linear Viscoelastic
  - — Isotropic Elastic-Plastic
    - — Mechanical Sublayer
    - — Isothermal Kinematic/Isotropic Hardening with Creep
    - — Nonisothermal Kinematic/Isotropic Hardening Viscoplasticity
  - — Micro-Mechanics Lamina Model(s) (with above material options)
- Lamina Failure Models (Temperature/Moisture-Dependent):
  - — Tsai-Wu
  - — Hoffman
  - — Tsai-Hill
  - — Maximum Stress
  - — Maximum Strain
  - — Hashin
- Transient Hygrothermal Diffusion Options:
  - — Uncoupled Linear
  - — Coupled Nonlinear
- Material and Post-Processing Database Access (GAL)

5.4.1 Overview

The Generic Constitutive Processor (GCP) contains five major functional components, an outer processor shell, an inner processor shell, an interface between the inner shell and both the outer shell and the Generic Element Processor (GEP), a standard interface between the inner shell and the constitutive kernel routines, and a set of constitutive kernel routines. Figure 5.4-1 gives a graphical depiction of the structure of the GCP and the interface to the GEP.

The outer processor shell provides a common user interface to the GCP, and enables it to be used as a stand alone processor for single point constitutive model testing. This shell processes user commands for input of material/fabrication data, interacts with the database, and directs the flow of computations. The outer shell incorporates a non-linear solution algorithm for stand-alone testing of constitutive models, and pointwise material laminate and material failure analyses. Command interpretation and database transactions are accomplished using the Testbed architecture. The outer shell organization is outlined in Section 5.4-2.

The inner processor shell performs through-thickness integration for composite laminates, interpolates state dependent material properties, performs transformations from element to fabrication coordinate systems, calls the constitutive kernel routines, and performs database management of the constitutive historical data, point stress/strain quantities, and material tangent stiffnesses. The functional organization of the inner shell is detailed in Section 5.4-3.

The unique functionality of the GCP providing both the capability for performing stand-alone constitutive analyses and analyses integrated with the GEP requires a flexible interface between the inner shell and the driving routine, either the GCP outer shell or the GEP. This interface consists of a common set of subroutine entry points collectively termed the generic constitutive interface. To maintain computational efficiency this interface will be linked with each individual element processor giving the latter access to the entire library of constitutive kernels, eliminating overhead associated with starting and stopping separate processors within an element loop.

A standardized interface between the inner shell and the constitutive kernel routines, termed the constitutive developer interface, provides the developer of new kernels insulation from the tasks of database management and element dependent calculations. This interface consists of a set of standard cover routines with fixed argument lists. The developer need only provide a set of pointwise constitutive kernels, which are plugged into the GCP using this interface.

The combination of the generic constitutive interface, inner processor shell, constitutive developer interface, and the constitutive kernels is referred to as the constitutive utility library. The constitutive utility library is linked with each element processor to provide efficient element constitutive functions as shown in Figure 5.4-1.

5.4.2 Outer Processor Shell

As previously discussed the outer processor shell of the GCP performs user input, directs the flow of computations in stand-alone mode, and provides post-processing capability for constitutive results. These functions are supplied by three modules in the outer shell, an input module, a solution module and a set of post processing utilities.

The input module of the GCP outer shell performs all user input of material and solution data relevant to the analysis. User commands will be available for the input of shell laminate fabrications, beam cross-section descriptions, material properties, and analysis parameters. The input module perform consistency checks of all user input and upon successful completion archives the data to the computational GAL database for subsequent use.

The solution module of the GCP outer shell provides the mechanism for performing stand-alone analysis of a material point. The types of analyses provided by the solution module are, linear and non-linear stress analysis, material failure analysis, laminate analysis, and linear thermal and diffusion analyses. For the solution of non-linear problems a modified Newton-Raphson algorithm is employed.

A set of post-processing utilities is provided to access constitutive data from the archival database and transform it to forms useful to the user. In addition to these utilities external

programs are available for graphical representation of constitutive results in the form of x-y plots.

Common to these modules of the outer shell are a suite of database utilities which serve as the interface between the GCP and the database managers GAL, and DMGASP. These utilities are shared with the inner processor shell.

5.4.3 Inner Processor Shell

The inner processor shell, which is accessible from either the GCP outer processor shell or the GEP shell through the generic constitutive interface, consists of a set of modules designed to act as the data interface between the element or material point and the pointwise constitutive model. On queues received from the element/point the inner shell directs the database access and flow of computations required to provide the pointwise constitutive kernel the data necessary for the requested function, and returns the results in a form compatible with the requesting element/point. The modules provided to perform these functions can be categorized into, control routines, shell laminate and beam cross-section integration routines, state variable manipulation routines, transformation routines, and database access utilities. The function of each of these modules is detailed below.

The control routines serve as the directors of the flow of data and computations for an element/point. The main function is to determine the sequence in which the other modules in the inner shell are accessed, and when the data is in the appropriate format, call the pointwise constitutive kernel (through the constitutive developer interface), or return control to the element/point (through the generic constitutive interface).

The shell laminate and beam cross-section integration routines perform the through-thickness integration for laminated shells, and the area integration for beams made of arbitrary subelements. These routines transform the stress-strain resultants into pointwise stresses/strains for use by the constitutive kernels, and subsequently integrate the resulting pointwise quantities to form the appropriate resultant quantities to be returned to the element/point.

The state variable manipulation routines interpret the state data passed from the element/point into the format expected by the constitutive kernel routines. Included are a

set of routines to perform interpolation for state dependent material calculations. These routines interpolate the material constants for a constitutive model based on the current state at the point and the input state dependent material properties.

A set of transformation routines performs the required transformations from the element local to the fabrication coordinate systems. These routines will transform the stresses, material stiffnesses, mass and damping matrices based on the orientation between the two coordinate systems and perform the transformations associated with shell and beam eccentricities. The theory associated with these transformations is detailed in Section 6.4.

An integrated set of database access utilities are provided for the manipulation of the relevant data entities used by the GCP. These utilities will use both the GAL and DMGASP data managers for access to data. The database utilities are shared with the outer shell.

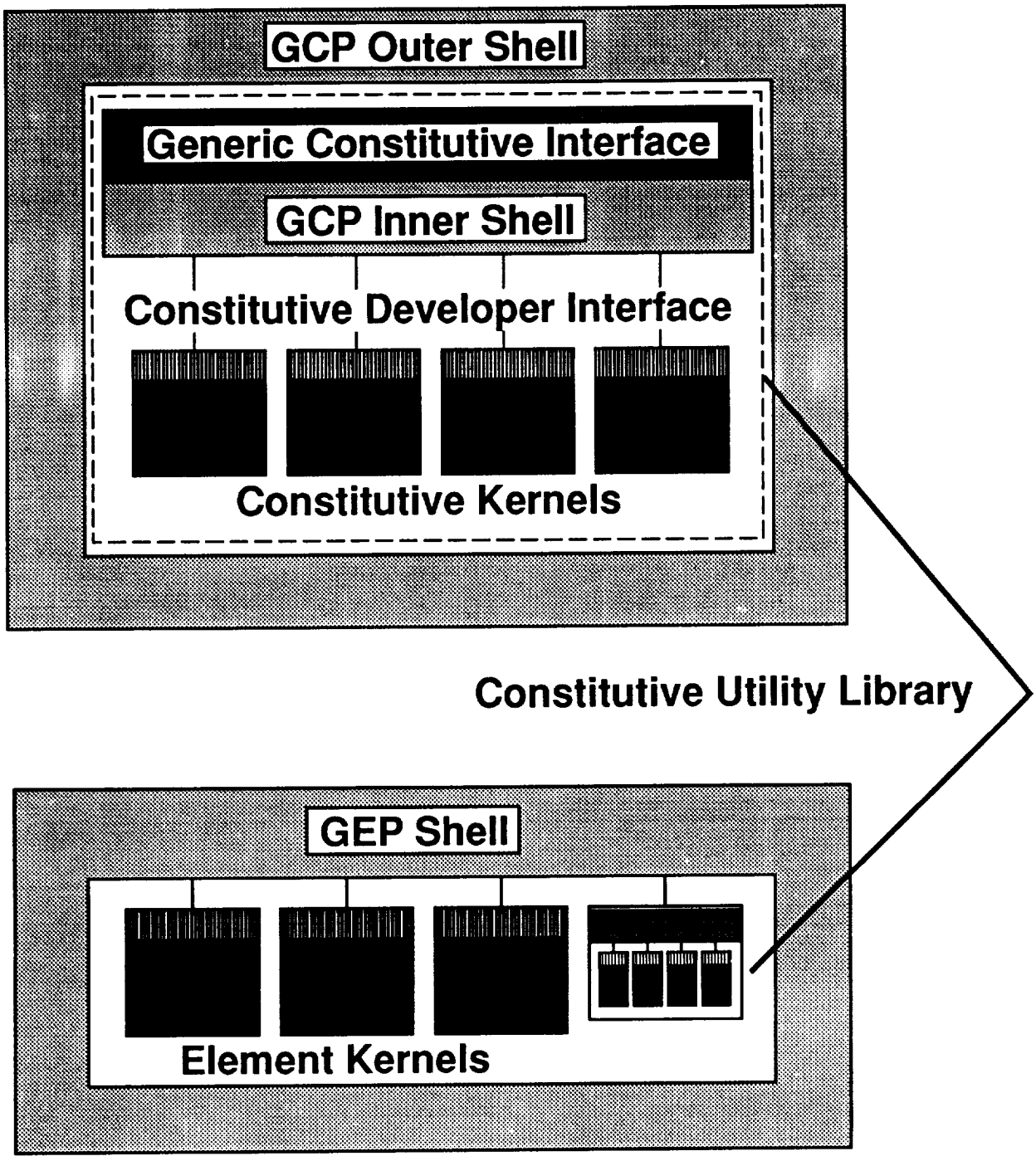


Figure 5.4-1 Generic Constitutive Processor Overview.

THIS PAGE LEFT BLANK INTENTIONALLY.

6. DATABASE INTERFACE

CHAPTER OUTLINE

Section	*Title*	*Description*
6.1	**Overview**	Summarizes interaction of ES processors with the global database.
6.2	**Dataset Descriptions**	Describes all datasets either input or output by ES processors (using the generic ES shell) from or to the global database. Includes tables listing datasets and showing relationship to ES processor commands, as well as detailed data structure descriptions for most of the datasets mentioned.
6.3	**Database Access Utilities**	Provides usage documentation for several database access utilities employed by the generic ES processor shell. These utilities may be called by other processors to conveniently access (input/output) many of the datasets described in Section 6.2. In particular, one routine facilitates access of the EFIL dataset, which is a rather complex data structure containing element stiffness matrices, coordinate information, connectivity, etc. — all combined within each individual element data record. Another utility, EL*, manipulates element load datasets.

6.1 Overview

In this chapter, we focus on the global *database* as it pertains to the generic structural-element (ES) processor, describing datasets that are either input or output by all ES processors. While Chapter 2 (User Interface) listed the various datasets associated with each ES processor command, this chapter provides detailed descriptions of the contents of those datasets. This serves two purposes: First, it enables the user to better anticipate the operation of ES processors, and to examine some of the results produced by ES processors (*e.g.*, stresses and strains) by simply using the CSM Testbed *PRINT directive. Second, it enables developers to build new processors that interface with ES processors strictly by communicating through the database, rather than requiring detailed knowledge of the actual software logic involved in the ES processors.

In addition to dataset descriptions, special-purpose database access utilities currently employed by the ES processor shell are also described in this chapter (Section 6.3). These software utilities may be used by processor developers to facilitate access to complex element datasets (such as the EFIL dataset). However, they will probably not be needed by ES processor developers, since the ES shell performs this function for element developers automatically.*

Finally, note that most of the dataset documentation presented in this chapter can also be found in the CSM Testbed Data Library Description (ref. 5). Such documentation is replicated here for the reader's convenience.

* It is also conceivable that ES processor developers may wish to access other datasets — beyond those that are automatically accessed by the ES shell and described herein. While this is permissible, it is currently recommended that those developers use the standard (generic) database management utilities described in reference 2.

6.2 Dataset Descriptions

Table 6.1 contains a summary of all datasets currently accessed by the generic ES processor – using the standard processor *shell*. Following this summary, the contents and data structures for many of these datasets is described in detail. Descriptions of those datasets not described here may be found in the CSM Testbed Data Library Description (ref. 5).

Table 6.1 SUMMARY OF DATASETS ACCESSED BY ES PROCESSORS

Dataset	*Contents*	*Input*	*Output*
DEF.<ES_NAME>.*	Element Defn. (Connectivity)	√	
DIR.<ES_NAME>.*	Element EFIL Directory	√	
ES.SUMMARY.*	ES Processor Status	√	√
<ES_NAME>.EFIL.*	Element Computational Data	√	√
JLOC.BTAB.*	Nodal Coordinates	√	
LOADS.<ES_NAME>.*	Element Loads	√	√
PROP.BTAB.*	Material/Section Properties	√	
QJJT.BTAB.*	Nodal Transformations	√	
STRN.<ES_NAME>.*	Element Strains		√
STRS.<ES_NAME>.*	Element Stresses		√
.DISP.	System Displacement Vector	√	
.FORC.	System Force Vector	√	√
.ROTA.	Nodal Rotation Pseudo-vectors	√	

6.2.1 Dataset ES.SUMMARY (Structural Element Summary)

6.2.1.1 Contents Summary

This dataset contains a comprehensive set of parameters which collectively describe each element type involved in the current model definition. The dataset is useful both for user queries during pre-processing and post-processing, as well as for driving the standard ES procedure (see Chapter 2), which cycles through all pertinent element to perform analysis functions.

6.2.1.2 Record Descriptions

Table 6.2 describes the various record groups stored in dataset ES.SUMMARY. The reader is also referred to the kernel argument glossary in Section 3.3, under argument array DEFS. The records stored in this dataset correspond to the individual elements of that argument array; *e.g.*, argument DEFS(pdNEN) corresponds to record ES_NEN.

The number of structural-element (ES) processors active in the current model is denoted *nesp*. The sequence of element processors/types represented in this dataset corresponds to the sequence in which the elements were defined using the DEFINE ELEMENTS command. Note that the ith element-processor and element-type defined in the model would be stored in records ES_PROC.i and ES_NAME.i, respectively.

Table 6.2 Records in Dataset ES.SUMMARY			
Record	*Type*	*Length*	*Description*
ES_C.1:*nesp*	Integ	(1)	Element displacement continuity (*e.g.*, $1 \Rightarrow C^1$).
ES_CLAS.1:*nesp*	Char	(4)	Element class, *e.g.*, BEAM | SHELL | SOLID.
ES_CNS.1:*nesp*	Integ	(1)	Constitutive option (see DEFS(pdCNS) in §3.3).
ES_DIM.1:*nesp*	Integ	(1)	Number of element intrinsic dimensions (1,2,3).
ES_NAME.1:*nesp*	Char	(4)	Element type name within processor (*e.g.*, EX97).
ES_NDOF.1:*nesp*	Integ	(1)	Number of freedoms per element node (1:6).
ES_NEE.1:*nesp*	Integ	(1)	Number of element equations (NDOF*NEN).
ES_NEN.1:*nesp*	Integ	(1)	Number of element nodes.
ES_NIP.1:*nesp*	Integ	(1)	Number of element integration (stress) points.
ES_NORO.1:*nesp*	Integ	(1)	Nodal drilling-rotation parameter (see §3.3).
ES_NSTR.1:*nesp*	Integ	(1)	Number of stress components per integ. point.
ES_OPT.1:*nesp*	Integ	(1)	Element developer's option number (internal).
ES_NPAR.1:*nesp*	Integ	(1)	Number of parameters stored in ES_PARS.
ES_PARS.1:*nesp*	Real	(npar)	Element research parameters.
ES_PROC.1:*nesp*	Char	(4)	Element processor name (*e.g.*, ES1, ES2, ...).
ES_PROJ.1:*nesp*	Integ	(1)	Corotational projection option (see §4.3).
ES_SHAP.1:*nesp*	Char	(4)	Element planform shape: TRIA | QUAD.
ES_STOR.1:*nesp*	Integ	(1)	Number of entries in Segment 6 of EFIL dataset.
ES_TGE.1:*nesp*	Integ	(1)	Element intrinsic frame option (1 or 2).

6.2.2 Dataset <ES_NAME>.EFIL.* (Element Computational Data)

6.2.2.1 Contents Summary

This dataset contains computational (intermediate) data for all elements of type <ES_NAME>; stored as a single nominal record group, DATA.1:*nel*, where *nel* is the total number of <ES_NAME> elements defined.

Each record (*i.e.*, element) contains mixed-type data, arranged in segments, whose offsets (*i.e.*, relative addresses) from the beginning of the record may be determined from dataset DIR.<ES_NAME>.

6.2.2.2 Record Descriptions

Table 6.3 Records in Dataset <ES_NAME>.EFIL.*

Record	*Type*	*Length*	*Description*
DATA.1:nel	Mixed	(See Table 6.4)	Primary record group.

In Table 6.3, *nel* is the number of elements of type <ES_NAME>.

6.2.2.3 Record DATA.*i* Segments:

The computational data for element *i* is stored in record DATA.*i*, and is partitioned into nine segments, as described in Table 6.4.

Table 6.4 Segments in Record DATA.*i* of Dataset <ES_NAME>.EFIL

Segment		*Item*	*Length*	*Type*	*Description*
1	Definition			Integ	Same as dataset DEF.<ES_NAME>.
2	Material			Real	(currently unused)
3	Geometry			Real	Element geometric parameters.
		XE0	(3,*nen*)		Initial element nodal coordinates in element basis.
		TEG	(3,3)		Transf. from global to current element basis (same as $\mathbf{E}^T$ in Chapter 4).
		TEC	(3,3,*nen*)		Transfs. from computational to element basis at element nodes.
		⋮	⋮		⋮
		XG0	(3,*nen*)		Initial elt. nodal coords in global basis.
		TEG0	(3,3)		Transf. from global to initial element basis (same as $[\mathbf{E}^0]^T$ in Chapter 4).
		DE	(*nee*)		Deformational displacements ($\mathbf{d}_e^{def}$).
4	Property			Real	(currently unused)
5	Matrix	KM	*nmt*	Real	Element matrix (stiffness/mass); only upper triangle of nodal blocks.
6	Aux. Storage			Real	Auxiliary storage for element developer.
7	Stress			Real	(currently unused)
8	Therm. Force			Real	(currently unused)
9	Therm. Stress			Real	(currently unused)

NOTES on Table 6.4:

1) Nomenclature: *nen* is the number of element nodes; *ndof* is the number of degrees of freedom per element node; *nee* is the number of element equations (usually $nen \times ndof$); and *nmt* is the number of matrix terms in the upper triangle of the element stiffness/mass matrix, which is stored in $ndof \times ndof$ nodal blocks. The number of matrix terms, *nmt*, is computed as follows:

$$nmt = ndof \times ndof \times nen \times (nen + 1)/2$$

2) The locations of segments 1-9 relative to the beginning of the record are given in dataset: DIR.<ES_NAME>.

3) The vertical dots after item TEC in segment 3 indicate that the following items are end-justified within the segment.

4) The element stiffness/mass matrix, item KM in segment 5, may be stored in either single or double precision, as specificed in dataset DIR.<ES_NAME> (entry 15). However, all of the other REAL data in the <ES_NAME>.EFIL dataset are stored exclusively in single precision.

6.2.3 Dataset LOADS.<ES_NAME>.<ES_LOAD_SET> (Element Loads)

6.2.3.1 Contents Summary

This dataset contains element loads for all elements of type <ES_NAME>, and load set number <ES_LOAD_SET>. A variety of records and record-groups may be present in this dataset, depending on the types of element loads that have been defined by the user. This dataset is usually created by the ES processor's DEFINE LOADS command, and is later used by ES processors in response to the FORM FORCE command, wherein element distributed loads are converted to consistent nodal forces.

6.2.3.2 Record Descriptions

Table 6.5 describes the various record groups potentially stored in dataset LOADS.<ES_NAME>.<ES_LOAD_SET>. A glossary of dimension parameters appearing in Table 6.5 is provided in the next section.

Table 6.5 Records in Dataset LOADS.<ES_NAME>.*

Record	*Type*	*Length*	*Description*
CONTENTS	Char	14	'ELEMENT_LOADS '
LINE_LDS.1:nllr	Real	(ndof,nnlt)	Line load vectors at element nodes.
LINE_TOC.1:nel	Integ	(1)	Element pointers to LINE_LDS records.
LINE_BND	Integ	(1)	Number of lines per element (nle).
LINE_NOD.1:nle	Integ	(1)	Number of nodes per element line.
LINE_DOF	Integ	(1)	Number of line-load components per node.
LINE_SYS	Char	12	Line-load vector coordinate system*
(continued ...)			

* Valid coordinate system names are GLOBAL, NODAL and ELEMENT.

Table 6.5 (cont.) Records in Dataset LOADS.<ES_NAME>.*			
Record	*Type*	*Length*	*Description*
SURF_LDS.1:nslr	Real	(ndof,nnst)	Surface-load vectors at element nodes.
SURF_TOC.1:nel	Integ	(1)	Element pointers to SURF_LDS records.
SURF_BND	Integ	(1)	Number of surfaces per element (nse).
SURF_NOD.1:nse	Integ	(1)	Number of nodes per element surface.
SURF_DOF	Integ	(1)	Number of surface load components per node.
SURF_SYS	Char	12	Surface-load vector coordinate system*
PRES_LDS.1:nllr	Real	(ndof,nnlt)	Line load vectors at element nodes.
PRES_TOC.1:nel	Integ	(1)	Element pointers to PRES_LDS records.
PRES_BND	Integ	(1)	Number of lines per element (nle).
PRES_NOD.1:nle	Integ	(1)	Number of nodes per element line.
PRES_DOF	Integ	(1)	Number of line-load components per node.
BODY_LDS.1:nslr	Real	(ndof,nnst)	Surface-load vectors at element nodes.
BODY_TOC.1:nel	Integ	(1)	Element pointers to BODY_LDS records.
BODY_BND	Integ	(1)	Number of surfaces per element (nse).
BODY_NOD.1:nse	Integ	(1)	Number of nodes per element surface.
BODY_DOF	Integ	(1)	Number of surface load components per node.
BODY_SYS	Char	12	Surface-load vector coordinate system*
LIV_*_LDS.1:nr	Real	(ndof,nnst)	Live-load vectors at element nodes.
LIV_*_TOC.1:nel	Integ	(1)	Element pointers to LIV_*_LDS records.
LIV_*_BND	Integ	(1)	Number of boundaries per element (nbe).
LIV_*_NOD.1:nbe	Integ	(1)	Number of nodes per element boundary.
LIV_*_DOF	Integ	(1)	Number of live-load components per node.
LIV_*_SYS	Char	12	Live-load vector coordinate system*

6.2.3.3 Glossary of LOADS.<ES_NAME>.* Dataset Parameters

Variable	Description
nbe	Number of element boundaries (generic for lines or surfaces, depending on actual load type).
nblr	Number of body-load records; one body-load record represents a body-load pattern for an entire element; many elements may point to the same body-load record if they have the same load pattern.
nel	Number of elements of type <ES_NAME>.
nle	Number of lines per element. For example, quadrilateral shell elements have 4 lines, and solid brick elements have 12 lines or edges.
nllr	Number of line-load records; one line-load record represents a line-load pattern for an entire element; many elements may point to the same line-load record if they have the same load pattern.
nlr	Number of load records; generic for line, surface, pressure or body load records — as appropriate.
nnlt	Total number of nodes on all lines of each element. For example, for a 4-node quadrilateral element, $nnlt = 8$; while for an 8-node solid element, $nnlt = 24$.
nnst	Total number of nodes on all surfaces of each element. For example, for a 4-node quadrilateral element, $nnst = 4$; while for an 8-node solid element, $nnst = 6$.
nplr	Number of pressure-load records; one pressure-load record represents a pressure-load pattern for an entire element; many elements may point to the same pressure-load record if they have the same load pattern.
nse	Number of surfaces per element. For example, quadrilateral shell elements have 1 surface, and solid brick elements have 6 surfaces or faces.

nslr Number of surface-load records; one surface-load record represents a surface-load pattern for an entire element; many elements may point to the same surface-load record if they have the same load pattern.

6.2.4 Dataset STRN.<ES_NAME>.* (Element Strains)

6.2.4.1 Contents Summary

This dataset contains element strains for all elements of type <ES_NAME>. A variety of record groups may be present in this dataset, representing different types of strain-evaluation locations within the element, and/or different coordinate systems in which the strain components may be resolved. All record groups contain data for one element per record, and the number and meaning of the strain components will depend on the element type (element type parameters are stored in, *e.g.*, dataset ES.SUMMARY).

6.2.4.2 Record Descriptions

The following table describes the various record groups stored (potentially) in dataset STRN.<ES_NAME>.*.

Table 6.6 Records in Dataset STRN.<ES_NAME>.*

Record	*Type*	*Length*	*Description*
INTEG_PTS_S*sys*.1:nel	Real	(*nstr*, *nip*)	Strains at element integration points.
CENTROIDS_S*sys*.1:nel	Real	(*nstr*)	Strains at element centroids.
NODES_S*sys*.1:nel	Real	(*nstr*, *nen*)	Strains at element nodes.

where *nel* is the number of elements of type <ES_NAME>; *nstr* is the number of strain components at each element evaluation point (*e.g.*, 6 for a solid continuum element); *nip* is the number of element integration points; *nen* is the number of element nodes; and *sys* is an index denoting the coordinate system in which the strain components are resolved. The index, *sys*, normally ranges between 0 and 3, where $sys = 0$ denotes the element's local stress coordinate system, and $sys > 0$ implies that material axis *sys* is the strain x axis, and the y and z axes follow by cyclic permutation.

6.2.5 Dataset STRS.<ES_NAME>.* (Element Stresses)

6.2.5.1 Contents Summary

This dataset contains element stresses for all elements of type <ES_NAME>. A variety of record groups may be present in this dataset, representing different types of stress-evaluation locations within the element, and/or different coordinate systems in which the stress components may be resolved. All record groups contain data for one element per record, and the number and meaning of the stress components will depend on the element type (element type parameters are stored in, *e.g.*, dataset ES.SUMMARY).

6.2.5.2 Record Descriptions

The following table describes the various record groups stored (potentially) in dataset STRS.<ES_NAME>.*.

Table 6.7 Records in Dataset STRS.<ES_NAME>.*

Record	*Type*	*Length*	*Description*
INTEG_PTS_S*sys*.1:nel	Real	(*nstr*, *nip*)	Stresses at element integration points.
CENTROIDS_S*sys*.1:nel	Real	(*nstr*)	Stresses at element centroids.
NODES_S*sys*.1:nel	Real	(*nstr*, *nen*)	Stresses at element nodes.

where *nel* is the number of elements of type <ES_NAME>; *nstr* is the number of stress components at each element evaluation point (*e.g.*, 6 for a solid continuum element); *nip* is the number of element integration points; *nen* is the number of element nodes; and *sys* is an index denoting the coordinate system in which the stress components are resolved. The index, *sys*, normally ranges between 0 and 3, where $sys = 0$ denotes the element's local stress coordinate system, and $sys > 0$ implies that material axis *sys* is the stress x axis, and the y and z axes follow by cyclic permutation.

6.2.6 Dataset *.DISP.* (System Displacements)

6.2.6.1 Contents Summary

This dataset contains nodal displacements for the assembled structure. Depending on the first part of the dataset name, the displacements may be total (TOT.DISP.*), incremental (INC.DISP.*), etc. The dataset name STAT.DISP.* is also used for the displacement solution in certain linear analysis procedures (*e.g.*, L_STATIC). The last (numeric) part of the dataset name is used either to denote the load and constraint cases — for linear static analyses, or to denote the load or time step for nonlinear static analyses, or transient analyses. For example, TOT.DISP.100 would be the total displacement solution at load step 100 in a nonlinear static analysis performed by procedure NL_STATIC_1.

Note that for geometrically (large-rotation) nonlinear analysis, the rotational components of TOT.DISP.* type datasets may not be physically meaningful. In that case, the current orientation of the nodal (surface) triads is used to represent the rotational part of the motion, and this is stored — in pseudo-vector form — in dataset TOT.ROTA.* (see description for dataset *.ROTA.*).

6.2.6.2 Record Descriptions

Only one record is stored in this dataset, as described in Table 6.8.

Table 6.8 Records in Dataset *.DISP.*

Record	*Type*	*Length*	*Description*
DATA.1	Real	$(ndof, nnodes)$	Nodal displacements in computational basis.

where $ndof$ is the number of potential freedoms at each node (see START command under processor TAB in CSM Testbed User's Manual, ref. 4), and $nnodes$ is the total number of nodes defined in the structural model — including nodes with prescribed displacements. Typically, $ndof = 6$, such that the first three displacement components at each node are translations, and the last three components are rotations (which are meaningful only for small/moderate-rotation analysis).

6.2.7 Dataset *.FORC.* (System Forces)

6.2.7.1 Contents Summary

This dataset contains nodal forces for the assembled structure. Depending on the first part of the dataset name, the forces may be internal (INT.FORC.*), external (EXT.FORC.*), residual (RES.FORC.*), etc. The dataset name REAC.FORC.* is also used for the reaction forces generated in certain linear analysis procedures (*e.g.*, L_STATIC_1). The last (numeric) part of the dataset name is used either to denote the load and constraint cases — for linear analysis, or to denote the load or time step — for nonlinear or transient analysis. For example, INT.FORC.100 would be the internal forces at load step 100 in a nonlinear static analysis performed by procedure NL_STATIC_1.

6.2.7.2 Record Descriptions

Only one record is stored in this dataset, as described in Table 6.9.

Table 6.9 Records in Dataset *.FORC.*

Record	*Type*	*Length*	*Description*
DATA.1	Real	(*ndof*, *nnodes*)	Nodal forces in computational basis.

where *ndof* is the number of potential freedoms at each node (see START command under processor TAB in CSM Testbed User's Manual, ref. 4), and *nnodes* is the total number of nodes defined in the structural model — including nodes with prescribed displacements. Typically, *ndof* = 6, such that the first three components at each node are actual forces, and the last three components are moments.

6.2.8 Dataset *.ROTA.* (System Rotation Pseudo-Vectors)

6.2.8.1 Contents Summary

This dataset contains nodal pseudo-vectors representing the rotation of the nodal freedom triad, from the initial configuration to the current configuration. These pseudo-vectors are relevant only for large-rotation geometrically nonlinear analysis in which rotational freedoms are used at some nodes. Otherwise, this dataset should not even appear in the database (note that the rotational components of the *.DISP.* datasets are meaningful for small or moderate rotation analysis).

When this dataset does exist, it is typically called TOT.ROTA.*, where the last (numeric) part of the name denotes the load (or time) step number in a nonlinear static (or dynamic) analysis.

For a definition of the term *pseudo-vector*, consult Chapter 4 and references mentioned therein. For present purposes, suffice it to say that the pseudo-vector at each node points in the direction of the axis of rotation, and the magnitude is simply the angle of rotation (in radians) — where the rotation is measured from the initial configuration to the current configuration, and the components are expressed in the global coordinate system. Note that there is a unique correspondence between pseudo-vectors and rotation (orthogonal) matrices, so that a full 3×3 triad can be obtained at each node from the 3-component pseudo-vectors. The pseudo-vector storage scheme is preferred over the matrix storage scheme for reasons of both size and computational efficiency.

6.2.8.2 Record Descriptions

Only one record is stored in this dataset, as described in Table 6.10.

Table 6.10 Records in Dataset *.ROTA.*

Record	*Type*	*Length*	*Description*
DATA.1	Real	(3, *nnodes*)	Nodal rotation pseudo-vectors expressed in global basis.

where *nnodes* is the total number of nodes defined in the structural model — including nodes with prescribed displacements.

6.3 Database Access Utilities

6.3.1 Subroutine NSXELT (Accessing Element Computational –EFIL– Datasets)

Calling Sequence

```
call NSXELT ( Option, Name, IELT, NELT, NIE, ELTDIR, NEN, XE0,
              TEGK, TEC, XG0, TEG0, DE, KME, STORE, NULL,
              STATUS )
```

Input Arguments

Option	'INITIALIZE' (set core boundaries) 'OPEN/OLD' (open existing dataset) 'OPEN/NEW' (open new dataset) 'GET' (get data for 1 element) 'PUT' (put data for 1 element) 'CLOSE' (flush buffer)
Name	Name of element computational dataset (EFIL.<ES_NAME>).
IELT	Number of element to be accessed within this dataset.

Input/Output Arguments (Input if OPTION='PUT', Output if OPTION='GET')

NELT	Total number of elements. (I)
NIE	Number of items per element. (I)
ELTDIR ()	Element directory array, from dataset DIR.<ES_NAME>. (I)

NEN	Number of element nodes (I)
XE0(3,NEN)	Coordinates of element nodes in initial element basis. (R)
TEGK (3,3)	Transformation from global basis to element basis: $\mathbf{T}^k_{eg} = (\mathbf{T}^k_{ge})^T$. (R)
TEC (3,3,NEN)	Transformation from computational basis to element basis at each element node. (R)
XG0 (3,NEN)	Initial coordinates of element nodes in global basis. (R)
TEG0 (3,3)	Transformation from global basis to initial element basis. (R)
DE (NDOF,NEN)	Current element nodal displacements (translations and rotations) relative to element basis. (R)
KME	(NDOF×NDOF×NEN×(NEN+1)/2) Element material stiffness matrix in element basis. (R)
STORE (NSTOR)	Element auxiliary storage data, created by INITIALIZE command. (R)
NULL (NSTOR/NEE)	Currently unused; previously reserved for embedded element stresses. (R)

Output Arguments

STATUS	Return status ($\geq 0 \Rightarrow$OK).

6.3.2 Subroutine NSXTAB (Accessing Table Datasets)

Calling Sequence

```
call NSXTAB ( Option, Name, ID, DATA, ICOL, NITEMS, NROWS,
              NCOLS, ITYPE, STATUS )
```

Input Arguments

Option	'INITIALIZE' 'OPEN/OLD' [/GET/PUT] 'OPEN/NEW' [/PUT] 'GET' 'PUT' 'CLOSE'
Name	Data-set name. (C)

Input/Output Arguments

ID	Internal dataset identifier; output if option involves an 'OPEN', otherwise input. (I)
DATA (NROWS)	One column of table dataset; input if option involves a 'PUT', output if option involves a 'GET', otherwise irrelevant. (R or I)
ICOL	Column number in table dataset; input if option involves a 'PUT', output if option involves a 'GET', otherwise irrelevant. (I)
NITEMS	Total number of items in table dataset; input if option = 'OPEN/NEW', output if option = 'OPEN/OLD'. (I)

NCOLS — Number of columns in table dataset (equal to NJ in the CSM Testbed Data Library Description, ref. 6); input if option = 'OPEN/NEW', output if option = 'OPEN/OLD'. (I)

ITYPE — Datum type; input if option = 'OPEN/NEW', output if option = 'OPEN/OLD'. (I)

Output Arguments

STATUS — Return status ($\geq 0 \Rightarrow$OK). (I)

C-4

6.3.3 Subroutine NSXNOM (Accessing Nominal Datasets)

Calling Sequence

call **NSXNOM** (Option, NamDS, NamRG, ID, DATA, ICOL , NITEMS, NROWS, NCOLS, ITYPE, STATUS)

Input Arguments

Option	'INITIALIZE' 'OPEN/OLD' [/GET/PUT] 'OPEN/NEW' [/PUT] 'GET' 'PUT' 'CLOSE'
NamDS	Dataset name. (C)
NamRG	Record group name. (C)

Input/Output Arguments

ID	Internal dataset identifier; output if option involves an 'OPEN', otherwise input. (I)
DATA (NROWS)	One column of table dataset; input if option involves a 'PUT', output if option involes a 'GET', otherwise irrelevant. (R or I)
ICOL	Column number in table dataset; input if option involves a 'PUT', output if option involes a 'GET', otherwise irrelevant. (I)
NITEMS	Total number of items in table dataset; input if option = 'OPEN/NEW', output if option = 'OPEN/OLD'. (I)

NROWS — Number of rows in table dataset; input if option = 'OPEN/NEW', output if option = 'OPEN/OLD'. (I)

NCOLS — Number of columns in table dataset (equal to NJ in the CSM Testbed Data Library Description, ref. 6); input if option = 'OPEN/NEW', output if option = 'OPEN/OLD'. (I)

ITYPE — Datum type (see CSM Testbed Data Library Description); input if option = 'OPEN/NEW', output if option = 'OPEN/OLD'. (I)

Output Arguments

STATUS — Return status ($\geq 0 \Rightarrow$OK). (I)

6.3.4 Element-Load Data Utilities

The FORTRAN utilities given in Table 6.3-1 should be used to access the element loads dataset. Calling sequences and argument definitions for the above utilities are given in the following sections.

Table 6.3-1 Data Utilities for Element Loads (EL) Data Objects

Subroutine	*Description*
ELinit	Initialize usage of EL utilities
ELopn	Begin access to a particular data object (old/new)
ELget	Get load data for a single element from the database
ELput	Put load data for one or more elmeents into the database
ELcls	Close access to a particular element loads data object
ELinf	Obtain information about a particular element loads data object

6.3.4.1 Subroutine ELinit

Subroutine ELinit initializes access to the element load database utility. It should be called once before making any calls to ELopn, ELget, ELput, etc.

Calling Sequence

```
call ELinit ( status )
```

Input Arguments

Name	*Type*	*Description*
(NONE)		

Output Arguments

Name	*Type*	*Description*
status	I	Return status; $\geq 0 \Rightarrow$OK

6.3.4.2 Subroutine ELopn

Subroutine ELopn is used to begin accessing a particular element loads (EL) data object.

Calling Sequence

```
call ELopn ( Qual, ldi, dsnam, Ldtyp, live, nel, nbndy, nnbndy, ncomp,
             ldsys, begcor, endcor, ELindx, status )
```

Input Arguments

Name	*Type*	*Description*
Qual	C	Open qualifier: 'NEW' ⇒Create new data object; 'OLD' ⇒Access existing data object.
ldi	I	Logical device index of data library.
DSnam	C	Dataset name where data object resides (*e.g.*, LOADS.eltnam.iset).
Ldtyp	C	Load (object) type name; 'LINE' ⇒line loads 'SURF' ⇒surface loads 'PRES' ⇒pressure loads 'BODY' ⇒body loads
live	I	Live load switch: 0 ⇒off 1 ⇒on
ldsys	I	Load coordinate system (only if /NEW)
nel	I	Number of elements in dataset (only if /NEW)
nbndy	I	Number relevant boundaries per element (only if /NEW)
nnbndy(nbndy)	I	Number of nodes per boundary (only if /NEW)
ncomp	I	Number of load components per node (only if /NEW)
begcor	I	Beginning address of workspace in blank common
endcor	I	Ending address of workspace in blank common

Output Arguments

Name	*Type*	*Description*
ELindx	I	Index of current data object (for future transactions with EL utilities)
ldsys	I	Load coordinate system (only if /OLD)
nel	I	Number of elements in dataset (only if /OLD)
nbndy	I	Number relevant boundaries per element (only if /OLD)
nnbndy(nbndy)	I	Number of nodes per boundary (only if /OLD)
ncomp	I	Number of load components per node (only if /OLD)
begcor	I	Beginning address of workspace in blank common
status	I	Return status; $\geq 0 \Rightarrow$OK

6.3.4.3 Subroutine ELget

Subroutine ELget is used to get element load values for a particular load type, and particular element(s). It is assumed that subroutine ELopn has already been called, so that the logical index for the particular data object (*i.e.*, ELindx) is known.

Calling Sequence

```
call ELget (  Qual, ELindx, ldrecd, load, eltnum, numelt, status )
```

Input Arguments

Name	*Type*	*Description*
Qual	C	Name of item to get: 'LOAD' ⇒get load record (default)
ELindx	I	Logical index of data object (from call to ELopn)
eltnum	I	Element number for which load data is requested
numelt	I	Number of elements for which load data is requested (currently restricted to 1)

Output Arguments

Name	*Type*	*Description*
ldrecd	I	Load record number; 0 if no loads are found for requested element
load(ncomp,nnbt)	F	Load vectors at element boundary nodes; *ncomp* = number of load components per node, which depends on the load type (*e.g.*, *ncomp* = 1 for pressure loads); *nnbt* = total number of nodes on all pertinent element boundaries (*e.g.*, surface by surface if pressure loads, line by line if line loads).
status	I	Return status; ≥ 0 ⇒OK

6.3.4.4 Subroutine ELput

Subroutine ELput is used to put element load values into the database for a particular load type, and set of element. It is assumed that subroutine ELopn has already been called, so that the logical index for the particular data object (*i.e.*, ELindx) is known.

Calling Sequence

```
call ELput (  Qual, ELindx, ldrecd, load, eltlis, numelt, status )
```

Input Arguments

Name	*Type*	*Description*
Qual	C	Name of item to get: 'LOAD' ⇒get load record (default)
ELindx	I	Logical index of data object (from call to ELopn)
ldrecd	I	Load record number (get from ELinf)
load(ncomp,nnbt)	F	Load vectors at element boundary nodes; *ncomp* = number of load components per node, which depends on the load type (*e.g.*, *ncomp* = 1 for pressure loads); *nnbt* = total number of nodes on all pertinent element boundaries (*e.g.*, surface by surface if pressure loads, line by line if line loads).
eltlis(numelt)	I	List of element numbers associated with current load values
numelt	I	Number of elements asssociated with current load values

Output Arguments

Name	*Type*	*Description*
status	I	Return status; ≥ 0 ⇒OK

6.3.4.5 Subroutine ELcls

Subroutine ELcls closes transactions with a specific element loads data object (*e.g.*, pressure loads within a given element loads dataset). It should be used after all ELget/ELput calls are made — and particularly if ELput has been used, to ensure that the database is properly updated.

Calling Sequence

```
call ELcls (    Qual,  ELindx,  status )
```

Input Arguments

Name	*Type*	*Description*
Qual	C	Currently unused.
ELindx	I	Logical index of element load data object being closed.

Output Arguments

Name	*Type*	*Description*
status	I	Return status; $\geq 0 \Rightarrow$OK

6.3.4.6 Subroutine ELinf

Subroutine ELinf gets information about the attributes of an element loads data object.

Calling Sequence

```
call ELinf (    Item, ELindx, ivals, fvals, chars, nvals, status )
```

Input Arguments

Name	*Type*	*Description*
Item	C	Name of attribute for which information is desired.
ELindx	I	Logical index of element load data object.

Output Arguments

Name	*Type*	*Description*
ivals(nvals)	I	List of integer values, if attribute is of integer type.
fvals(nvals)	I	List of floating point values, if attribute is of floating point type.
Chars*nvals	C	Character string, if attribute is of type character.
status	I	Return status; $\geq 0 \Rightarrow$OK.

THIS PAGE LEFT BLANK INTENTIONALLY.

7. ARCHITECTURE INTERFACE

7.1 GEP Internal Organization

The internal organization of the Generic Element Processor (GEP) is illustrated in Figure 7.1. As shown the processor shell (ES) is split into two layers, a top layer that is accessed through subroutine ES0, and a bottom layer that is accessed through subroutine ES0CR. The top layer (ES0) handles the user and database interfaces, making calls to the Testbed architectural utilities to parse commands and input/output datasets. The bottom layer (ES0CR) is a general-purpose interface (or cover routine) to all kernel-level functions. ES0CR thus calls each of the individual, specific-function cover routines, such as ES0E (strains), ES0KM (material stiffness), ES0FI (internal force), etc., and coordinates them with calls to constitutive (CS*), corotational (CR*) and matrix-algebra (GS*) utilities to produce the desired output quantity for a given element.

7.2 GEP Use of the NICE Architecture

All GEP calls to the Testbed architecture, *i.e.*, CLIP (command language) and GAL (database) utilities, are made from the top layer of the ES processor shell. This means either directly in subroutine ES0 or through various utilities that are called by ES0.

7.2.1 GEP Calls to CLIP

Procedure ES calls to CLIP for command and macrosymbol processing are made in subroutines ES0, ES0CMD, and ES0LDS. CLIP subroutine CLREAD is used to read command lines; functions ICLVAL and CCLVAL are used to parse command lines, and functions ICLMAC, CCLMAC and DCLMAC are used to fetch macrosymbol values. Additionally, subroutine CLGET is used to initialize macrosymbol definitions.

7.2.2 GEP Calls to GAL

Procedure ES does not call GAL directly for database management in version 1.2 of the Testbed. Instead, it calls higher-level dataset utilities, such as NSX*, which in turn call the SPAR database cover routine complex, DAL/RIO, which finally make the calls to GAL. The extra layer created by DAL/RIO was required at one time for compatibility with the Testbed. However, it is due to be removed in version 1.3 of the Testbed.

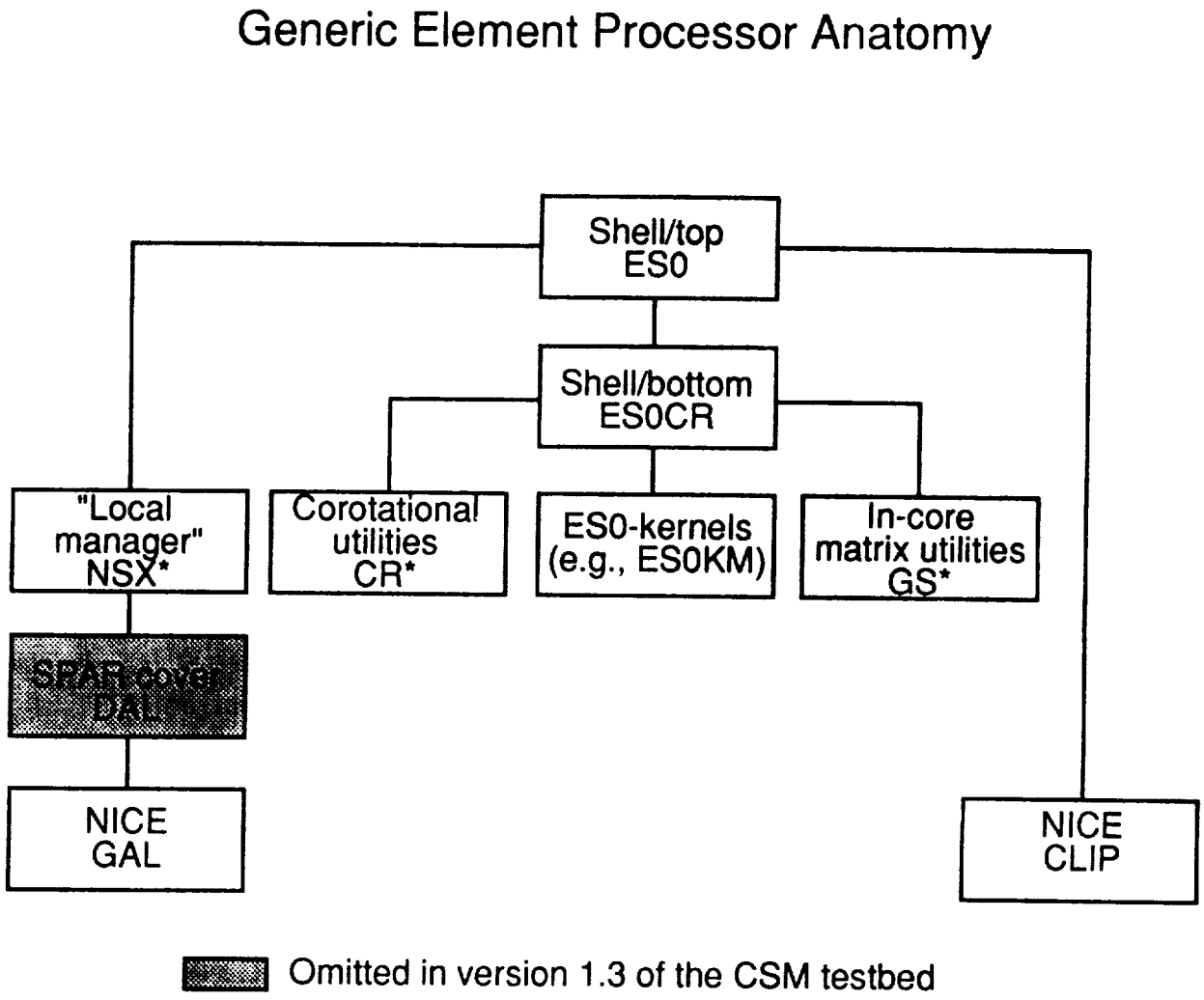


Figure 7.1 Architecture of GEP.

8. REFERENCES

1. Knight, N.F., Jr.; Gillian, R. E.; McCleary, S.L.; Lotts, C.G.; Poole, E.L.; Overman, A.L.; and Macy, S.C.: *CSM Testbed Development and Large-Scale Structural Applications.* NASA TM-4072, 1989.

2. Felippa, Carlos A.: *The Computational Structural Mechanics Testbed Architecture: Volume II - Directives.* NASA CR-178385, 1988.

3. Wright, Mary A.; Regelbrugge, Marc E.; and Felippa, Carlos A.: *The Computational Structural Mechanics Testbed Architecture: Volume IV - The Global-Database Manager GAL-DBM.* NASA CR-178387, 1988.

4. Stewart, Caroline B.: *The Computational Structural Mechanics Testbed User's Manual.* NASA TM-100644, October 1989.

5. Knight, N.F.; McCleary, S.L.; and Stanley, G.M.: *The Computational Structural Mechanics Testbed Procedures Manual.* NASA TM-100646, 1989.

5. Stewart, Caroline B., Compiler: *The Computational Structural Mechanics Testbed Data Library Description.* NASA TM-100645, October 1988.

6. Belytschko, T. and Hsieh, B.J.: Nonlinear Transient Finite Element Analysis with Convected Coordinates, *International Journal of Numerical Methods in Engineering*, vol. 7, 1973, pp. 255-271.

7. Wempner, G.: Finite Elements, Finite Rotations and Small Strains of Flexible Shells, *International Journal of Solids and Structures*, vol. 5, 1969, pp. 117-153.

8. Rankin, C.C. and Brogan, F.A.: An Element-Independent Corotational Procedure for the Treatment of Large Rotations, *ASME Journal of Pressure Vessel Technology*, vol. 108, 1986, pp. 165-174.

9. Rankin, C.C. and Nour-Omid, B.: The Use of Projectors to Improve Finite Element Performance, *Computers and Structures*, vol. 30, 1988, pp. 257-267.

THIS PAGE LEFT BLANK INTENTIONALLY.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-181728		

4. Title and Subtitle	5. Report Date
The Computational Structural Mechanics Testbed Generic Structural-Element Processor Manual	March 1990
	6. Performing Organization Code
7. Author(s)	8. Performing Organization Report No.
Gary M. Stanley and Shahram Nour-Omid	LMSC-D878511
9. Performing Organization Name and Address	10. Work Unit No.
Lockheed Missiles and Space Company, Inc. Research and Development Division 3251 Hanover Street Palo Alto, California 94304	505-63-01-10
	11. Contract or Grant No.
	NAS1-18444
12. Sponsoring Agency Name and Address	13. Type of Report and Period Covered
National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225	Contractor Report
	14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitors: W. Jefferson Stroud and Norman F. Knight

16. Abstract

The purpose of this manual is to document the usage and development of structural finite element processors based on the CSM Testbed's Generic Element Processor (GEP) template. By convention, such processors have names of the form ES*i*, where *i* is an integer.

This manual is therefore intended for both Testbed users who wish to invoke ES processors during the course of a structural analysis, and Testbed developers who wish to construct new element processors (or modify existing ones).

17. Key Words (Suggested by Authors(s))	18. Distribution Statement
Structural analysis software Finite Element Implementation Corotational Formulation CSM Testbed System	Unclassified—Unlimited Subject Category 39

19. Security Classif.(of this report)	20. Security Classif.(of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	293	A13

NASA FORM 1626 OCT 86

For sale by the National Technical Information Service, Springfield, Virginia 22161-2171