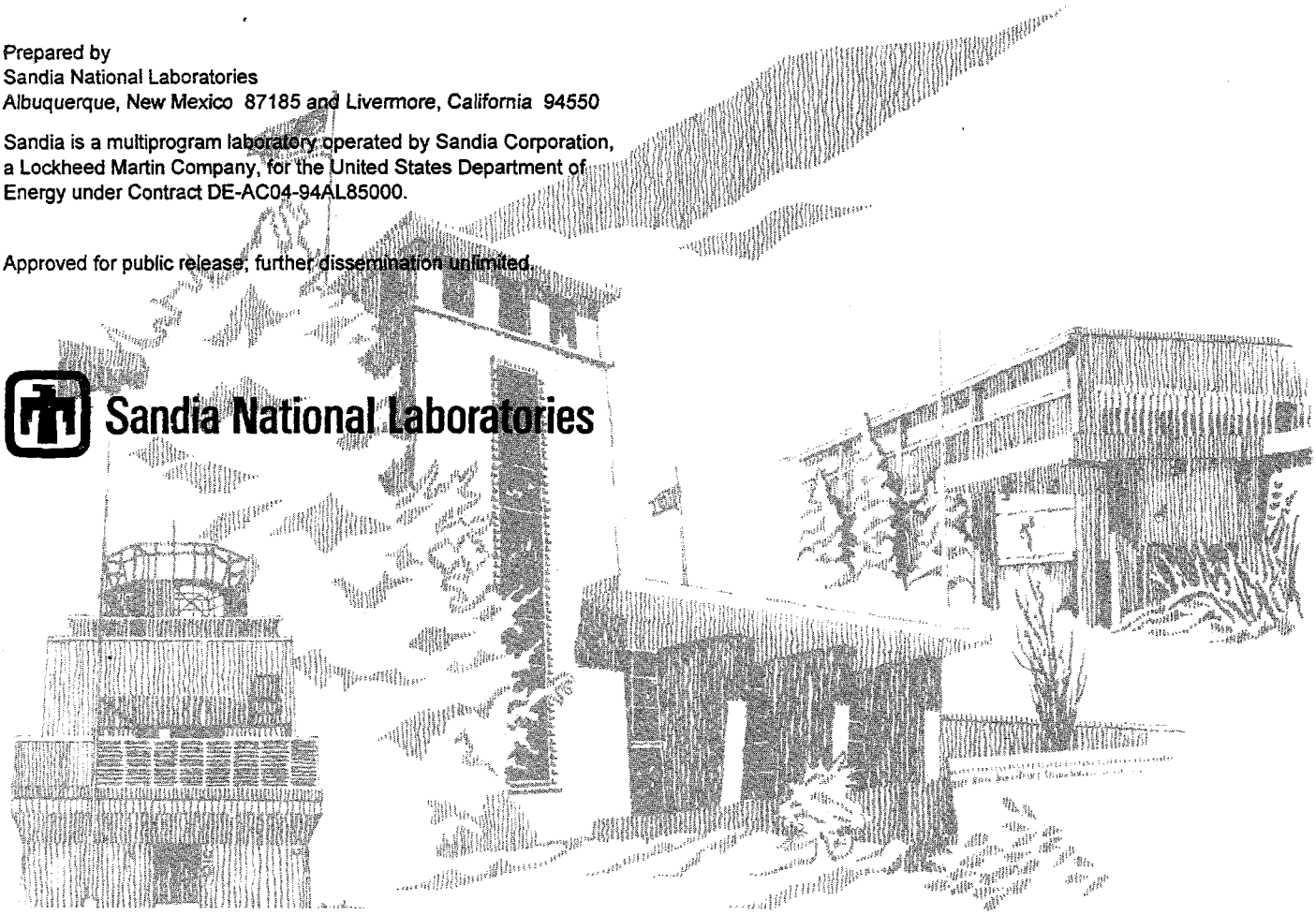
SANDIA REPORT
SAND99–8229
Unlimited Release
Printed April 1999

An Annotated Reference Guide to the Finite-Element Interface Specification Version 1.0

Robert L. Clay, Kyran D. Mish, Ivan J. Otero, Lee M. Taylor, Alan B. Williams

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.



Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Prices available from (615) 576-8401, FTS 626-8401

Available to the public from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd
Springfield, VA 22161

NTIS price codes
Printed copy: A04
Microfiche copy: A01

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

SAND99-8229
Unlimited Release
Printed April 1999

An Annotated Reference Guide to the Finite-Element Interface Specification

Robert L. Clay, Lee M. Taylor, Alan B. Williams
Sandia National Laboratories

Ivan J. Otero
Lawrence Livermore National Laboratories

and

Kyran D. Mish
California State University, Chico

Abstract

The Finite-Element Interface (FEI) specification provides a layered abstraction that permits finite-element analysis codes to utilize various linear-algebra solution packages with minimal concern for the internal details of the solver modules. Alternatively, this interface can be viewed as a way for solver developers to provide solution services to finite-element clients without having to embed finite-element abstractions within their solver libraries. The purpose of this document is to provide some level of documentation between the bare interface specification itself, which consists only of C/C++ header files, and the full documentation suite that supports the interface definition by providing considerable detail as to its design and implementation. This document primarily provides the "how" of calling the interface member functions, so that programmers can readily learn how to utilize the interface implementation without having to consider all the details contained in the interface's definition, design, and motivation.

The interface specification is presented three times in this document, each time with an increasing level of detail. The first presentation provides a general overview of the calling sequence, in order to acquaint the programmer with a basic introduction to how the interface is used to "train" the underlying solver software on the particular finite-element problem that is to be solved. The second pass through the interface definition provides considerable detail on each method, including specific considerations as to the structure of the underlying data, and an exposition of potential pitfalls that may occur as a byproduct of either the finite-element modeling process, or of the use of the associated interface implementation. Finally, a third description of the interface is given implicitly via the discussion of sample problems that provide concrete examples of the use of the finite-element interface.

Table of Contents

1. Introduction

The Finite-Element Interface (FEI) specification provides a layered abstraction that permits finite-element analysis codes to utilize linear-algebra solution packages without worrying about the internal details of the solver modules. Alternatively, this interface can be viewed as a way for solver developers to provide solution services to finite-element clients without having to embed finite-element physics abstractions within their solver libraries[1].

1.1. The Purpose of this Document

The purpose of this document is to provide some level of documentation between the bare interface specification itself, which consists of C/C++ header files[2], and the full documentation set [Clay, et. al, 1999, as well as various URLs listed in the references] that supports the interface definition by providing considerable detail as to its design and implementation. This document primarily provides the "how" of calling the interface member functions, so that programmers can readily learn how to utilize the interface implementation without having to consider all the details contained in the interface's definition, design, and motivation.

In order to be used with the full range of finite-element clients and solution servers, the interface's calling sequence and data structures must be sufficiently general and extensible so as to support the full variety of finite-element data types.

This document implicitly assumes some knowledge of the fundamental abstractions used in the finite-element interface specification. These definitions can be found in the associated Sandia Technical Report on the interface [Clay, et. al, 1999a], but simplified versions are given in an appendix to this document as well.

1.2. How to Use This Document

The interface specification is presented three times in this document, each time with an increasing level of detail. The first presentation provides a general overview of the calling sequence, in order to acquaint the programmer with a basic introduction to how the interface is used to "train" the underlying solver software on the particular finite-element problem that is to be solved. The second pass through the interface definition provides more detail regarding each method, including specific considerations as to the structure of the underlying data. It also includes an exposition of pitfalls that may occur as a byproduct of either the finite-element modeling process, or of the use of the associated interface implementation. The third description of the interface is given

---

[1] This version of the specification does not address matrix-free implementations or multi-level methods. Since a primary function of the interface is to load explicit representations of linear algebraic components (e.g., matrices and vectors), matrix-free implementations are, strictly speaking, not needed. While the FEI could be used to provide common interaction with the solvers, the underlying linear component interfaces can be efficiently used directly in that case. Multi-level extensions to this specification are currently being developed.

[2] Note that for purposes of supporting procedural codes there is a C header file (fei_proc.h) which is functionally equivalent to the C++ header file (fei.h).

implicitly via the exposition of various sample problems that provide concrete examples of the use of the finite-element interface. In each of these example problems, a particular simple finite-element analysis is performed to demonstrate many of the more subtle concepts embedded in the finite-element interface specification.

1.3. Some Fundamental Assumptions and Conventions

Throughout this document, a color mono-spaced font is used to highlight source-code listings. These source examples are taken either from the C++ interface specification header file (fei.h), or from the various example problems included with the FEI implementation package that accompanies the full interface specification distribution.

The reader is assumed to have some familiarity with the C++ programming language, as that language is used in all concrete expositions of procedures and data. The general term "identifier", abbreviated "ID" is taken to represent a global (i.e., uniquely valid over all processors) numbering scheme. Finally, a more detailed exposition of fundamental assumptions and design principles can be found in the references [Clay, et. al, 1999a].

2. Interface Specification

This section provides two views of the interface specification:

- an overview of the interface calling sequence, and
- a more detailed exposition of the individual procedures that collectively define the interface specification.

2.1. Overview of Calling Sequence

The interface consists of four main steps, namely:

(1) initialization,

(2) loading,

(3) solution, and

(4) solution parameter return.

These individual processes will be discussed in order below. In addition to these four fundamental constituents, various utility functions are provided to aid the finite-element developer in using the interface.

2.1.1. Initialization

The first step in the calling process is to pass the structure of the finite-element data, so that this physical structure can be translated into an algebraic sparse matrix.

The data passed during the initialization step includes:

- control data defining the underlying element types and solution fields used, as well as data indicating how many aggregate finite-element data types will be utilized,

- element data, including element connectivity information that can be used to translate finite-element nodal equations to systems of sparse algebraic equations,
- control data for nodes that must be handled via special logic, for example, nodes shared among processors, and
- data to aid in the definition of any constraint relations local to a given processor.

At the end of the initialization step, the interface implementation can determine the underlying matrix structure, and allocate memory for matrix storage in preparation for the load step.

2.1.2. Load

Once the matrix structure memory has been allocated, that structure can be populated with finite-element data according to standard finite-element matrix assembly procedures. This is the general task of the load step in the interface calling sequence.

Examples of data passed during the load step include:

- boundary-condition data for implementing essential, natural, or mixed boundary conditions for nodes associated with each processor,
- element arrays, both element stiffness and loads, passed as aggregate element set abstractions, and
- constraint relations, defined in terms of nodal algebraic weights and tables of associated nodes.

At the end of the load step, the sparse matrix structure is fully populated and has all appropriate boundary and algebraic constraint relations implemented.

2.1.3. Solution

Once the load step is complete, the solver module's internal solution procedures may be invoked. This process necessarily involves passing control data (e.g., convergence tolerances and control codes to indicate the type of preconditioner and solver to be used) through the interface to the solver. This step in the calling sequence therefore typically requires some solver-specific parameters, although reasonable defaults are provided. In addition to providing methods to invoke the solver, the specification also permits passing initial solution estimates to the solution module, which is especially useful in the case of iterative solvers being used in nonlinear and/or transient finite-element analyses.

2.1.4. Solution Return

Upon completion of the solution process, the finite-element solution data must be passed from the solver back to the finite-element client. During this step, the interface implementation transfers the algebraic solution data into arrays corresponding to finite-element data. These aggregate data structures support the following processes:

- return nodal solution parameters in containers associated with each block of elements,
- return elemental solution parameters using the same blocked containers, and

- return constraint data on either an individual or collective basis.

In each case, utility functions (e.g., query functions to determine the size of the associated containers) are provided to support these return processes, as the finite-element client program may need to allocate memory in support of the solution return step.

2.2. Detailed Exposition of Individual Calls

The following subsections provide more detail on the various methods used to implement the interface, and on associated programming support. In each case, C++ conventions are used to demonstrate the various procedures and data that define the interface, even though the interface is also intended for use with procedural languages like ANSI C and FORTRAN. The C++ language is utilized here because it provides the most precise and concise specification for the interface's procedures and data, and because the original implementation of the specification was written in C++.

2.2.1. Associated Header Files

The file "basicTypes.h" contains definitions required to define some of the derived data types used in the interface specification. In particular, the *GlobalID* data type is defined to permit long integer identifiers to be associated with nodes and elements used by large-scale finite-element programs. This type is provided because a conventional 32-bit integer may not be sufficient for either large (e.g., multi-billion node) finite-element meshes, or to permit the finite-element developer to utilize part of the nodeID/elementID representation for alternative purposes (e.g., adaptive mesh refinement, load balancing, etc.). In any case, the use of the GlobalID data type insulates the interface specification from unwarranted machine-dependencies caused by reliance on a particular integer format used to represent nodes and elements.

The header files "fei.h" and "cfei.h" define the FEI specification precisely, in C++ and ANSI C format, respectively. The ANSI C header is recommended for developers who wish to work in procedural languages like FORTRAN.

2.2.2. Object-Oriented Design Philosophy

Because the fundamental motivation for the interface specification is the abstraction of equation-solver services needed by finite-element programs, each method used in the interface is associated with a specific linear equation system object. In particular, the various procedures that define the C++ interface specification are implemented as class methods associated with a *SparseLinearEquations* object. The procedural (C) interface specification is functionally equivalent. In order to support multiple instances of linear systems, the procedural functions contain an extra parameter for specifying a particular system of equations.

The *SparseLinearEquations* object is given a "communicator" at construction time to establish a context for inter-processor communications. Currently, the FEI implementation utilizes MPI (Message Passing Interface) for communications, so the constructor method takes the following form:

```
SparseLinearEquations(MPI_Comm FEI_COMM_WORLD);
```

2.2.3. Initialization

The various initialization methods are presented below. In each case, the method returns an integer-valued warning status indicator. In general, a zero return value implies that the method has completed without problems, and nonzero values imply some form of warning that is appropriate to the particular method's function. In the initial v1.0 release of the implementation, developers are advised to look up nonzero warnings directly in the source code distribution found in the *fei-isis/src/* distribution directory.

2.2.3.1. Solve-Step Initialization

Each solution step corresponds to the formation of a single system of linear equations, and an initialization call is made to indicate the start of a new solve step. This code to **initSolveStep** takes two parameters:

- *numElemBlocks,* the total number of element blocks[3] passed in this solve step,
- *solvType,* a parameter to identify the solution type (currently taken as zero, which corresponds to simply solving a linear system, but which will eventually include nonzero values for eigensolution and other services – if a nonzero value is passed for version 1.0 of the specification, it is simply ignored).

The C++ **initSolveStep** function takes the following form:

```
int initSolveStep(int numElemBlocks,
                  int solvType);
```

2.2.3.2. Field Initialization

Because the finite-element method is a general approximation scheme capable of dealing with multiple interacting physical processes, it is commonly desirable to be able to model multi-physics effects during a finite-element simulation. In order to facilitate modeling separate approximations for the various mathematical solution fields present in a problem, the interface requires the finite-element client program to identify all the fields present in the simulation. This identification is handled via the **initFields** method, shown here using its C++ calling convention:

```
int initFields(int numFields,
               const int *cardFields,
               const int *fieldIDs);
```

The following parameters are passed to the **initFields** method:

- *numFields,* the total number of solution fields utilized in this analysis,
- *cardFields,* the list of solution cardinalities[4] associated with each field, and

---

[3] An element block is a collection of elements lying on a single processor and satisfying specific criteria concerning generic element representations. The reader is referred to Appendix A (the glossary) for details.

- *fieldIDs,* a list of integer field identifiers used to identify the various solution fields with their unique element-based approximations

The **initFields** method must be called on all processors, with the same arguments on each processor. All fields present in the entire simulation (i.e., not just local to a processor) need to be provided. A sample field initialization is given below, for the case of a coupled stress/thermal analysis using one element block, where the element displacement field is approximated with a biquadratic interpolant, and the element temperatures are approximated using a bilinear interpolant.

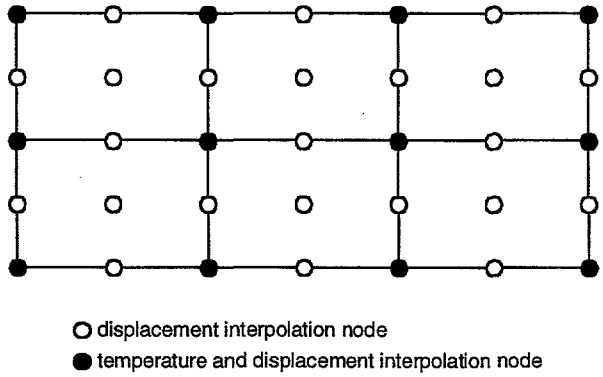


Figure 1: Multiple-field example

Under the assumption that the displacement field is identified with the integer ID 5 and the temperature field with the integer ID 10, the data structures passed to **initFields** take the following form:

numFields: *2*

cardFields: [2 1]

fieldIDs: [5 10]

2.2.3.3. Blocked-Element Initialization

The initialization data is passed in on a block-by-block basis, beginning with functions that permit element-block connectivities to be converted to algebraic sparsity structure. There are three methods used to handle these blocked-element initialization tasks.

---

[4] The nodal solution cardinality is the total number of individual scalar solution components interpolated at a node. For a displacement vector in 3D, the field solution cardinality is 3, and for a scalar field, the solution cardinality is 1. In a multiphysics setting, the nodal solution cardinality is the sum of all the field solution cardinalities, over each field defined at a given node. In general, the reader should establish by context whether the term "cardinality" refers to a given field defined at a node, or to the accumulation of all the fields defined at a node.

The **beginInitElemBlock** method marks the start of the initialization process for a particular block, and this initialization routine is called exactly once for each block. The C++ **beginInitElemBlock** function takes the following form:

```
int beginInitElemBlock(GlobalID elemBlockID,
                       int numNodesPerElement,
                       const int *numElemFields,
                       const int *const *elemFieldIDs,
                       int interleaveStrategy,
                       int numElemDOF,
                       int numElemSets,
                       int numElemTotal);
```

The parameters passed to **beginInitElemBlock** include:

- *elemBlockID,* the element block ID for this block of generic element data,
- *numNodesPerElement,* the generic number of nodes for each element in this block,
- *numElemFields,* the list of number of fields for each node of each generic element,
- *elemFieldIDs,* the table of field identifiers, with each row corresponding to the list of fieldIDs associated with a given node in the generic element,
- *interleaveStrategy,* a parameter that indicates the packing strategy used for passing the nodal field data for each element (this parameter indicates whether element matrix storage will be oriented by a field-first or node-first arrangement. Currently, there are two options, FIELD_MAJOR and NODE_MAJOR, corresponding to the element matrix storage schemes diagrammed in Figure 2 below),

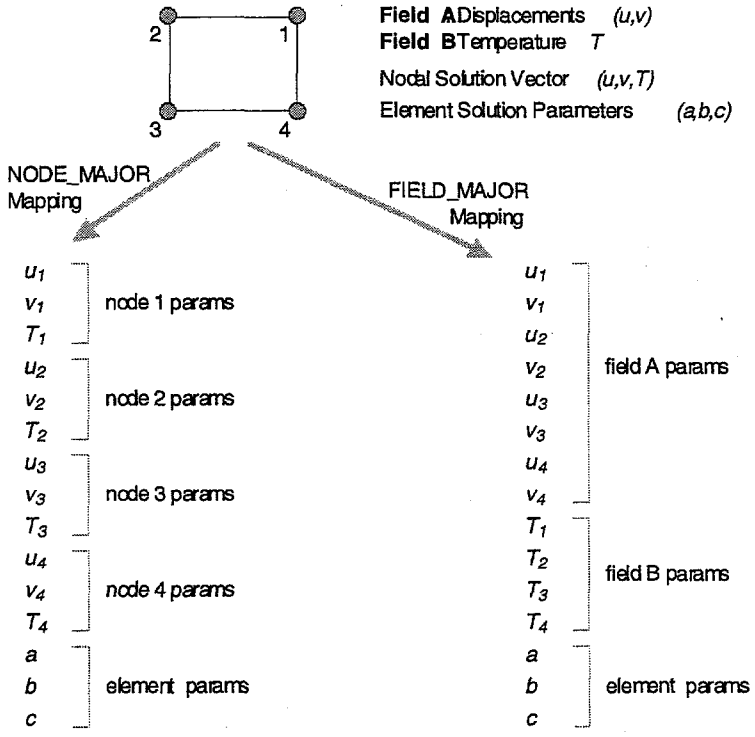


Figure 2: Element storage interleave strategies

- *numElemDOF,* the number of element solution parameters for elements in this block,
- *numElemSets,* the total number of elemSets[5] in this block, and
- *numElemTotal,* the total number of elements in this block.

The **initElemSet** method is invoked once for every element set aggregation required to define a particular block. The C++ **initElemSet** function takes the following form:

```
int initElemSet(int numElems,
                const GlobalID *elemIDs,
                const GlobalID *const *elemConn);
```

The parameters passed to *initElemSet* include:

- *numElems,* the number of elements in this element set,
- *elemIDs,* the list of element IDs for this block, and
- *elemConn,* the matrix of connectivity data for these elements.

The **endInitElemBlock** method marks the end of the initialization process for a particular block, and this initialization routine is called exactly once for each block. This terminal initialization method requires no passed parameters.

```
int endInitElemBlock();
```

2.2.3.4. Node Set Initialization

Nodal data passed through the interface implementation can be decomposed into three classes, and the last two of these must be identified during node set initialization:

- nodes where boundary conditions are specified[6],
- nodes that are shared among processors, and
- external nodes, not present in the active node list[7] and requiring inter-processor communications operations to resolve data needs.

Fundamental information for these last two node classes is provided to the interface implementation in the initialization step via four method calls. The first and last methods

---

[5] A collection of elements grouped to achieve a computational economy of scale: in general, the reader should refer to the glossary for definitions of terms used in this exposition.

[6] It should be noted that nodes where boundary condition information is specified need not be identified or processed during the initialization process. On the other hand, nodes requiring interprocessor communications must be identified during the initialization step.

[7] The set of all nodes associated with a block of elements located on a given processor. More information on this term (and others) is given in the glossary.

define the start/end structure of the node set initialization process, and these two methods are called exactly once per solve step. The other two methods are called once per associated node set, and represent the two classes of node sets that require interprocessor communications.

The **beginInitNodeSets** method marks the start of the nodal-data initialization process for a particular solve step. This initialization routine is called exactly once for each solve step. The C++ **beginInitNodeSets** function takes the following form:

```
int beginInitNodeSets(int numSharedNodeSets,
                      int numExtNodeSets);
```

The parameters passed to **beginInitNodeSets** include the two basic nodal cases that require interprocessor communications:

- *numSharedNodeSets,* the number of shared node sets to be passed, and
- *numExtNodeSets,* the number of external node sets to be passed.

The **initSharedNodeSet** method is called to identify the set of local nodes that are shared with one or more other processors. The C++ **initSharedNodeSet** function takes the following form:

```
int initSharedNodeSet(const GlobalID *sharedNodeIDs,
                      int lenSharedNodeIDs,
                      const int *const *sharedProcIDs,
                      const int *lenSharedProcIDs);
```

The parameters passed to **initSharedNodeSet** include the following:

- *sharedNodeIDs,* the list of shared node IDs in this node set,
- *lenSharedNodeIDs,* the number of shared nodes in this list,
- *sharedProcIDs,* the table of processor IDs for the shared nodes in this nodal list, with the number of rows equal to the number of shared nodes, and the length of each row given by the number of processors which share that node, and
- *lenSharedProcIDs,* the length of each list of processor IDs for all of the shared nodes in *sharedNodeIDs* (note: *lenSharedProcIDs*, like *sharedNodeIDs*, has a length of *lenSharedNodeIDs*).

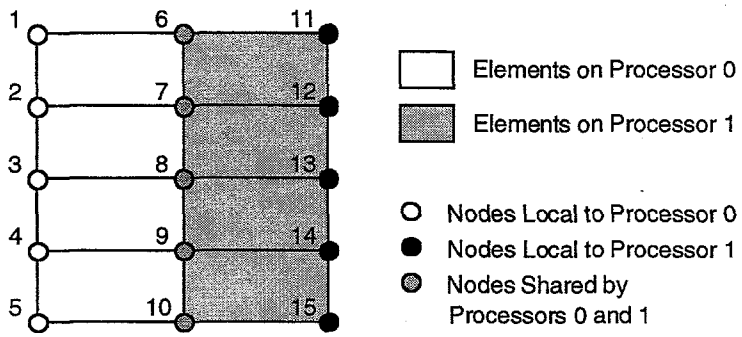


Figure 3: Shared topology sample geometry

In the figure above, the passed data structures take the following form:

lenSharedNodeIDs: *5*

sharedNodeIDs: [6 7 8 9 10]

sharedProcIDs:
0 1
0 1
0 1
0 1
0 1

lenSharedProcIDs: [2 2 2 2 2]

Because of the symmetry of the problem (with respect to sharing of nodes between the two processors), the data passed by each processor is identical. Thus the shared-node example above is relatively simple, and the resulting table of shared processor IDs takes a simple form. A more complex (and realistic) example is given in Figure 4 below:

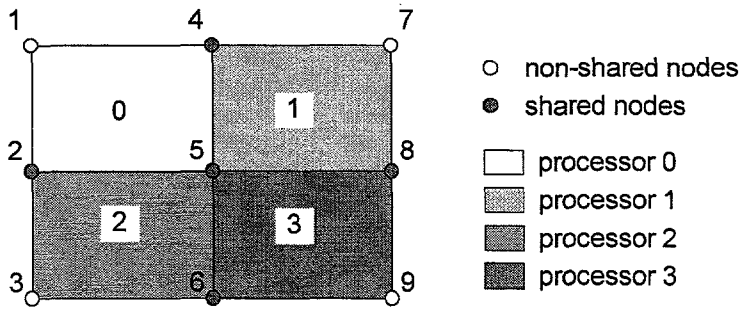


Figure 4: Complex shared topology sample geometry

In this more complicated example, the passed data structures take the following form.

Processor 0:

lenSharedNodeIDs: *3*

sharedNodeIDs: [2 4 5]

	0	2	...	...
sharedProcIDs:	0	1	...	...
	0	1	2	3

lenSharedProcIDs: [2 2 4]

Processor 1:

lenSharedNodeIDs: *3*

sharedNodeIDs: [4 5 8]

	0	1	...	...
sharedProcIDs:	0	1	2	3
	1	3	...	...

lenSharedProcIDs: [2 4 2]

Processor 2:

lenSharedNodeIDs: *3*

sharedNodeIDs: [2 5 6]

	0	2	...	...
sharedProcIDs:	0	1	2	3
	2	3	...	...

lenSharedProcIDs: [2 4 2]

Processor 3:

lenSharedNodeIDs: *3*

sharedNodeIDs: [5 6 8]

	0	1	2	3
sharedProcIDs:	2	3	...	...
	1	3	...	...

lenSharedProcIDs: [4 2 2]

In these shared-node examples, the passed data is presented as sorted by ID, but this convention is merely to aid in understanding. Unless otherwise stated explicitly in this document, no presumption is made in general as to whether the data may arrive in a sorted or random order.

External nodes are nodes which either:

(a) are involved in local calculations (e.g., appear in local constraint relations) but are not found in the active node list, or

(b) are in the local active node list and are involved in another processor's calculations, but are not in that processor's active node list.

The **initExtNodeSet** method must be used to identify external nodes on the owning processor, as wel as all processors which use them in calculations. The C++ **initExtNodeSet** function takes the following form:

```
int initExtNodeSet(const GlobalID *extNodeIDs,
                   int lenExtNodeIDs,
                   const int *const *extProcIDs,
                   const int *lenExtProcIDs);
```

The parameters passed to **initExtNodeSet** include the following:

- *extNodeIDs,* the list of external global node IDs in this node set,
- *lenExtNodeIDs,* the number of external nodes in this list,
- *extProcIDs,* the table of processor IDs for the external nodes in this nodal list, with number of rows given by the number of external nodes, and the length of each row being the number of processors involved in communications for that node, and
- *lenExtProcIDs,* the length of each list of processor IDs for all of the external nodes in *extNodeIDs* list (note: *lenExtProcIDs*, like *extNodeIDs*, has a length of *lenExtNodeIDs*).

A simple external-node example is shown in Figure 5 below. This example is taken from the driver program *fei-isis/fei-drivers/distExtBeamDriver.cc*, and is developed further in Chapter 3 of this document. In the figure below, the three solution field parameters (two components of displacement, one of rotation) at nodes 2 and 3 are to be made equal across the boundary of processors 0 and 1, and the development of these constraints requires specification of external nodes for each processor.

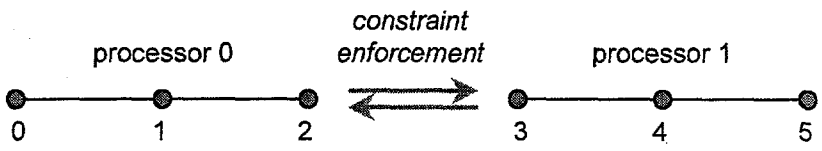


Figure 5: Sample external node geometry

In this example, processor 0 must inform processor 1 of the need for communications from node 2. In addition, processor 1 must inform processor 0 of the need to communicate information relevant to node 3. Thus each processor must inform the other of communications that will be required to satisfy the across-processor constraint. The form of the required data is presented below, for the case of the particular constraint tying nodes 2 and 3 together (other constraints may exist, such as one at the right edge of processor 1, and this additional data is not presented here).

On processor 0, the passed data takes the form:

lenExtNodeIDs: *1*

extNodeIDs: [2]

extProcIDs: [1]

lenExtProcIDs: [1]

On processor 1, the passed external node data is given by:

lenExtNodeIDs: *1*

extNodeIDs: [3]

extProcIDs: [0]

lenExtProcIDs: [1]

More details on passing external node information can be found by examining the drivers *fei-isis/fei-drivers/distExtBeamDriver.cc* and *fei-isis/fei-drivers/distExtPenDriver.cc*.

The **endInitNodeSet** method marks the end of the nodal initialization process for a particular solve step, and this initialization routine is called exactly once for each solve step. This terminal initialization method requires no passed parameters, and takes the following form in C++:

```
int endInitNodeSets();
```

2.2.3.5. Constraint Set Initialization

Constraint data passed through the interface can be decomposed into two distinct types:

- Lagrange multiplier constraints, where additional solution unknowns (namely, Lagrange multipliers) are appended to the system of equations as a by-product of the implementation of the individual constraints, and
- penalty constraints, where the underlying finite-element energy functional is penalized for any lack of constraint satisfaction by adding fictitious energy terms that do not produce any new solution parameters.

The data that defines these two exclusive constraint types is provided to the interface in the initialization step via four method calls. The first and last methods define the start/end structure of the constraint set initialization process, and are called exactly once per solve step. The other two methods are called once per constraint set, and represent the two types of constraints enumerated above.

The **beginInitCREqns** method marks the start of the constraint relation (CR) initialization process for a particular solve step, and this initialization routine is called exactly once for each solve step. The C++ form of this function is given by:

```
int beginInitCREqns(int numCRMultSets, int numCRPenSets);
```

The parameters passed to **beginInitCREqns** include:

- *numCRMultSets,* the number of Lagrange multiplier constraint sets to be passed, and
- *numCRPenSets,* the number of penalty constraint sets to be passed.

The **initCRMult** method is called for each local constraint set containing data for algebraic constraints implemented using Lagrange multipliers. The C++ version of **initCRMult** takes the form below:

```
int initCRMult(const GlobalID *const *CRNodeTable,
               int *CRFieldList,
               int numMultCRs,
               int lenCRNodeList,
               int& CRMultID);
```

The parameters passed to **initCRMult** include the following:

- *CRNodeTable,* the table of node IDs associated with this constraint, with *numCRs* rows and *lenCRNodeList* columns (each row of this table identifies a list of nodes that defines a specific constraint),
- *CRFieldList,* the list of field identifiers associated with each nodal contribution to the constraint relations (note that nodes can be specified more than once in any given constraint, so that distinct nodal solution fields can be coerced into satisfaction of algebraic constraint relations defined by conditions of geometric compatibility).
- *numMultCRs,* the number of Lagrange multiplier constraints to process in this particular constraint set,
- *lenCRNodeList,* the number of columns in *CRNodeTable,* and

- *CRMultID,* a returned constraint set reference identifier to aid in distinguishing the various constraint sets during subsequent load and solution return phases

More precise definitions of the data that defines the *CRNodeTable* and *CRFieldList* data structures can be found below in the annotation of the associated constraint set load funtion **loadCRMult**.

The **initCRPen** method is called for each local constraint set containing data for algebraic constraints implemented using a penalty approach. The C++ version of **initCRPen** takes the form below:

```
int initCRPen(const GlobalID *const *CRNodeTable,
              int *CRFieldList,
              int numPenCRs,
              int lenCRNodeList,
              int& CRPenID);
```

The parameters passed to **initCRPen** include the following:

- *CRNodeTable,* the table of node IDs associated with this constraint, with *numCRs* rows and *lenCRNodeList* columns (each row of this table identifies a list of nodes that defines a specific constraint),
- *CRFieldList,* the list of field identifiers associated with each nodal contribution to the constraint relations (note that nodes can be specified more than once in any given constraint, so that distinct nodal solution fields can be coerced into satisfaction of algebraic constraint relations defined by conditions of geometric compatibility).
- *numPenCRs,* the number of penalty constraints to process in this particular constraint set,
- *lenCRNodeList,* the number of columns in *CRNodeTable,* and
- *CRPenID,* a returned constraint set reference identifier to aid in distinguishing the various constraint sets during subsequent load and solution return phases.

The **endInitCREqns** method marks the end of the constraint initialization process for a particular solve step. This initialization routine is called exactly once for each solve step. This terminal initialization method requires no passed parameters. The C++ form of this function is given by:

```
int endInitCREqns();
```

2.2.3.6. Initialization Completion

The **initComplete** method marks the end of the entire initialization process for a particular solve step, and this initialization routine is called exactly once for each solve step. There are no passed parameters associated with this method, and upon calling this

terminal initialization step, the associated interface implementation is charged with determining the underlying sparse matrix structure, and then allocating memory for a sparse matrix to be used in subsequent steps. The C++ form of this function is given by:

```
int initComplete();
```

2.2.4. Load Sequence

Where the initialization process permits determination of the matrix structure of the underlying finite-element equation set, it does not provide for passing any of the values that populate that matrix. The task of populating the finite-element system stiffness matrix and load vector is the responsibility of the load step, and the various load methods are presented below. In each case, the method returns an integer-valued error status indicator, so that each method is of type int.

2.2.4.1. Matrix Reuse Function

The member function **resetSystem** permits reuse of an existing linear system's matrix structure. This function can be called to permit performing new solves without invoking the interface implementation's initialization phase. In order to utilize this function, the matrix structure cannot change from the last solve step, so in particular, there must be the exact same mesh topology, boundary condition format, and number (and type) of constraint relations. The last restriction implies that any slide-surface constraint implementation is unchanged from the last solve step, which places substantial restrictions on the use of contact/impact/penetration algorithms. It is the responsibility of the finite-element developer to insure that all these restrictions are satisfied in practice before this particular utility function is invoked.

The **resetSystem** method call uses one passed parameter (by default, set to zero) to represent the scalar used to fill each matrix entry in the underlying sparse matrix that is to be reused. Its C++ form is given by:

```
int resetSystem(double s=0.0);
```

2.2.4.2. Node Set Loading

The first stage in performing the load step for a new finite-element equation system involves passing all the associated boundary-condition information to the interface implementation. The boundary-condition information must be passed before the raw element matrices, so this node set load step precedes the blocked element load process. This convention arises because the element data does not necessarily contain any embedded essential boundary-condition data, and hence must be modified in order to implement essential boundary condition information into the assembled sparse system stiffness matrix.

There are three general methods used to handle these node set loading tasks:

- the method used to mark the beginning of the node set loading process,
- the iterated method that loads the boundary condition data, and

- the method used to mark the end of the node set loading process.

The **beginLoadNodeSets** method marks the start of the nodal data load process for a particular solve step. This routine is called exactly once for each solve step, and takes the following form in C++:

```
int beginLoadNodeSets(int numBCNodeSets);
```

The single parameter passed to **beginLoadNodeSets** is defined as:

- *numBCNodeSets,* the number of boundary-condition node sets to process for this solve step on this processor.

The **loadBCSet** method is called for each node set containing data for nodes associated with boundary conditions on this processor.

```
int loadBCSet(const GlobalID *BCNodeSet,
              int lenBCNodeSet,
              int BCFieldID,
              const double *const *alphaBCDataTable,
              const double *const *betaBCDataTable,
              const double *const *gammaBCDataTable);
```

At this point, it is helpful to include the fundamental description of the generic boundary-condition representation used in the interface specification. If the primary field solution unknown (taken to be a scalar[8]) is denoted by u, the dual of the solution (e.g., force as opposed to displacement, heat source as opposed to temperature, etc.) is denoted by q, and the nodal values of these solution parameters are indicated by a subscript j, then a generic boundary specification can be given by:

$$\alpha_j u_j + \beta_j q_j = \gamma_j$$

where α_j, β_j, and γ_j are specified constants.

The table below specifies various values for the constants required to produce the three fundamental types of boundary conditions.

essential	$\alpha_j \neq 0$	$\beta_j = 0$	γ_j arbitrary
natural	$\alpha_j = 0$	$\beta_j \neq 0$	γ_j arbitrary
mixed	$\alpha_j \neq 0$	$\beta_j \neq 0$	γ_j arbitrary

Appropriate values for these defining constants are passed for each solution component in the node list through the *alphaBCDataTable*, *betaBCDataTable*, and *gammaBCDataTable* arrays.

---

[8] If the solution is vector- or tensor-valued, this parameter should be taken as one component of the vector nodal solution.

The parameters passed to **loadBCSet** include the following:

- *BCNodeSet,* the list of node IDs for this boundary-condition node set,
- *lenBCNodeSet,* the length of the *BCNodeSet* list,
- *BCFieldID,* the field identifier for the solution component that is being specified,
- *alphaBCDataTable,* the table of boundary-condition coefficient data for the primary solution term, with *lenBCNodeSet* rows, and number of columns given by the field cardinality of the solution field being specified,
- *betaBCDataTable,* the table of boundary-condition dual-solution data for the secondary solution term, with *lenBCNodeSet* rows, and number of columns given by the field cardinality of the solution field being specified, and
- *gammaBCDataTable,* the table of boundary-condition constant data (i.e., non-homogenous terms that define the boundary condition), with *lenBCNodeSet* rows, and number of columns given by the field cardinality of the solution field being specified.

As a simple concrete example, consider the following boundary condition node set, containing nodes 1, 2, and 3, where each of these nodes has exactly two solution fields:

(a) a vector solution field, representing displacements in the x- and y-directions, symbolized by u and v, respectively, and

(b) a temperature field defined at the same nodes, symbolized by T.

The general geometric interpretation is diagrammed in Figure 6 below.

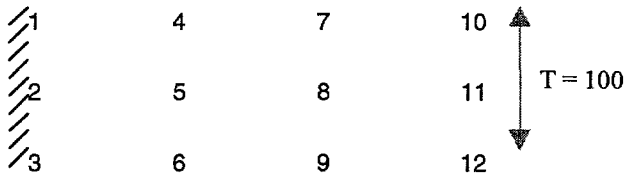


Figure 6: Sample boundary-condition geometry

The first field boundary condition to be considered here is that of a fixed edge along nodes 1, 2, and 3. This implies that the x- and y-components of displacement both vanish at each node, and this vector-valued displacement boundary condition can be implemented using two separate BC sets, namely:

- one set to specify that there is no x-component of displacement at any of the three nodes, and
- another to specify that there is no y-component of displacement for the node set.

In the subsequent presentation of the required data structures, recall the field identification data given in an earlier example from this document, where the displacment field had an ID of 5, and the temperature field had an ID of 10:

numFields: 2

cardFields: [2 1]

fieldIDs: [5 10]

For the first specification (i.e. the specification of u, the x-component of displacement), the data passed takes the following form:

BCFieldID = 5

lenBCNodeSet = 3

BCNodeSet: [1 2 3]

alphaBCDataTable:
1 0
1 0
1 0

betaBCDataTable:
0 0
0 0
0 0

gammaBCDataTable:
0 0
0 0
0 0

Note how each table is three rows long by two columns wide. This table size arises because there are three nodes grouped into the single nodeSet, and each nodal field being specified has two scalar solution components.

The second specification is identical to the first, save for the different values of the entries in the *alphaBCDataTable*, which represent the vanishing of the second (i.e., y) component of displacement instead of the first component:

alphaBCDataTable:
0 1
0 1
0 1

The remaining boundary condition specification involves setting of the non-homogenous temperature condition along nodes 10, 11, and 12. The required data structures for implementing this case are given below:

BCFieldID = 10

lenBCNodeSet = 3

BCNodeSet: [10 11 12]

alphaBCDataTable:
1
1
1

betaBCDataTable:
0
0
0

gammaBCDataTable:
100
100
100

Note in this thermal specification how each table is three rows long by one column wide, because the temperature field is a scalar, and hence only possesses one component.

The **endLoadNodeSet** method marks the end of the nodal load process for a particular solve step. This terminal routine is called exactly once for each solve step, and requires no passed parameters. It takes the following C++ form:

```
int endLoadNodeSets();
```

2.2.4.3. Blocked-Element Loading

The interface implementation is loaded on a block-by-block basis, beginning with the **beginLoadElemBlock** method that initiates the blocked-data loading. There are three underlying methods used to handle these blocked-element initialization tasks:

- marking the beginning of a blocked-load,
- performing the load of element data, and
- marking the end of a block load.

The **beginLoadElemBlock** method marks the start of the load process for a particular block, and this load routine is called only once for each block. Its C++ form is given by:

```
int beginLoadElemBlock(GlobalID elemBlockID,
                       int numElemSets,
                       int numElemTotal);
```

The parameters passed to **beginLoadElemBlock** include:

- *elemBlockID,* the identifier for this particular block. This GlobalID identifier was also passed via the blocked-element initialization routine *beginInitElemBlock,* and is used here to identify the particular element block,
- *numElemSets,* the total number of elemSets[9] in this block, and
- *numElemTotal,* the total number of elements in this block.

The **loadElemSet** method is called for each element set containing data for this particular block. Its C++ form is given by:

```
int loadElemSet(int elemSetID,
                int numElems,
                const GlobalID *elemIDs,
                const GlobalID *const *elemConn,
                const double *const *const *elemStiffness,
                const double *const *elemLoad,
                int elemFormat);
```

The parameters passed to the **loadElemSet** method include:

- *elemSetID,* the elemSet identifier for this group of elements,
- *numElems,* the number of elements in this element set,
- *elemIDs,* the list of element IDs for this block,
- *elemConn,* the mesh connectivity (repeated from **initElemSet**) for this element set,
- *elemStiffness,* the list of element stiffness matrices for this element set, here stored as a 3D array whose structure depends upon the *elemFormat* parameter described below,
- *elemLoad,* the list of element load vectors, here stored as a matrix, with each row of the matrix representing a single element's load vector, and
- *elemFormat,* a flag to indicate the layout of element stiffness data, including full square storage of element stiffnesses, or various row-wise packing of symmetric stiffnesses (this format is intended to be extensible, but the following formats are implemented in the current version of the specification).

The *elemFormat* field presently includes the following standard types:

---

[9] Note that the number of element sets for the load step does not necessarily have to be the same number used in the initialization step. In general, these sets may differ in size between the two cases in order to reflect optimal use of the memory hierarchy by considering the different amounts of data passed during the initialization and the load steps.

elemFormat = 0	Dense, non-symmetric storage, where the element matrix is stored in a row-contiguous manner without any regard for symmetry.
elemFormat = 1	Packed, upper-symmetric storage, where the upper triangle of the element stiffness matrix is stored in a row-contiguous manner.
elemFormat = 2	Packed, lower-symmetric storage, where the lower-triangle of the element stiffness is stored in a row-contiguous manner.
elemFormat = 3	Dense, non-symmetric storage, where the element matrix is stored in a column-contiguous manner without any regard for symmetry
elemFormat = 4	Packed, upper-symmetric storage, where the upper-triangle of the element stiffness is stored in a column-contiguous manner.
elemFormat = 5	Packed, lower-symmetric storage, where the lower-triangle of the element stiffness is stored in a column-contiguous manner.

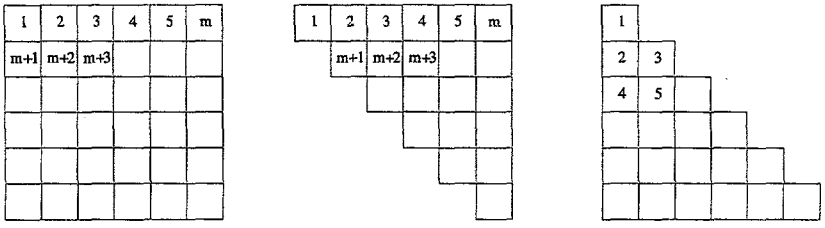


Figure 7: Row-contiguous base formats for element matrix storage

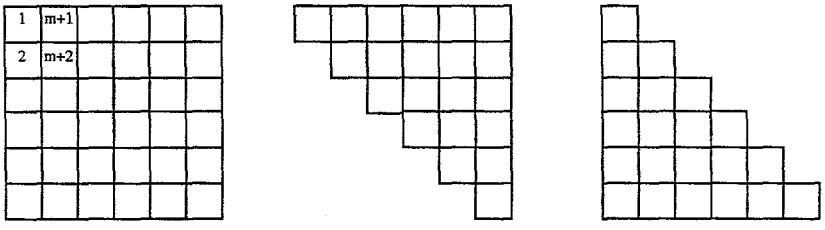


Figure 8: Column-contiguous base formats for element matrix storage

Finally, the **endLoadElemBlock** method marks the end of the element-block loading process for a particular block. This routine is called exactly once for each block. This method requires no passed parameters, and takes the following C++ form:

```
int endLoadElemBlock();
```

2.2.4.4. Constraint Set Loading

Constraint data is loaded via the two underlying types of constraint relations: (a) constraints implemented via Lagrange multipliers, and (b) those implemented using a penalty formulation. These two classes of constraint enforcement schemes are bracketed with marking routines to delineate the beginning and end of the constraint loading process.

The **beginLoadCREqns** method marks the start of the constraint loading process for a particular solve step. This load routine is called exactly once for each solve step, and takes the following C++ form:

```
int beginLoadCREqns(int numCRMultSets,
                    int numCRPenSets);
```

The parameters passed to **beginLoadCREqns** echo the same parameters passed during the **beginInitCREqns**:

- *numCRMultSets,* the number of Lagrange multiplier constraint sets to be passed, and
- *numCRPenSets,* the number of penalty constraint sets to be passed.

The **loadCRMult** method is called for each local constraint set containing data for algebraic constraints implemented using Lagrange multipliers. Its C++ form is given below:

```
int loadCRMult(int CRMultID,
               int numMultCRs,
               const GlobalID *const *CRNodeTable,
               const int *CRFieldList,
               const double *const *CRWeightTable,
               const double *CRValueList,
               int lenCRNodeList);
```

The parameters passed to **loadCRMult** include the following:

- *CRMultID,* the constraint set reference identifier returned for this constraint set by the *initCRMult* initialization method,
- *numMultCRs,* the number of Lagrange multiplier constraints to process in this constraint set,
- *CRNodeTable,* the table of node IDs associated with this constraint, with *numCRs* rows and *lenCRNodeList* columns (each row of this table identifies a list of nodes that defines the specific constraints),

- *CRFieldList,* the list of field identifiers associated with each generic constraint, of length *lenCRNodeList* (note that field identifiers can be repeated within this list, if distinct field components at the same node are to be constrained),
- *CRWeightTable,* the table of weights associated with this constraint, with *lenCRNodeList* rows, and with the number of columns given by the field solution cardinality at each node in the node table,
- *CRValueList,* the list of non-homogenous values associated with each constraint, stored as a vector of length *numMultCRs,* and
- *lenCRNodeList,* the number of columns in *CRNodeTable.*

The following example problem demonstrates loading a simple algebraic constraint. The problem modeled is the junction of a beam and a continuum, where the beam elements possess two solution fields per node (a planar displacement and a scalar out-of-plane rotation) and the continuum elements have one solution field per node (only the displacement vector). The geometry of the problem is shown in Figure 9 below.

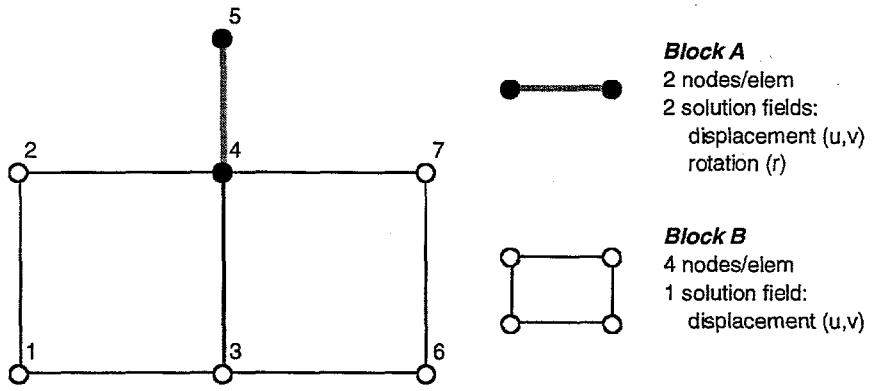


Figure 9: Example constraint geometry

The constraint embedded in this example is that the rotation field at the lower node of the beam must be slaved to a linear combination of the normal (vertical) displacements along the upper edge of the continuum. The general form of the constraint at the common node (node 4 in the figure above) is given by:

$$r_4 = w_2 v_2 + w_4 v_4 + w_7 v_7$$

where the weights w_i depend upon the geometry of the continuum elements. This constraint can be thus represented in the canonical form given by:

$$r_4 - w_2 v_2 - w_4 v_4 - w_7 v_7 = 0$$

If the field ID associated with the rotation field is taken as 10 and the field ID associated with the displacement is given by 5, then the relevant data structures passed to the **loadCRMult** method are given by the following:

$$NumMultCRs = 1$$

$$LenCRNodeList = 4$$

$$CRNodeTable = \begin{bmatrix} 4 & 2 & 4 & 7 \end{bmatrix}$$

$$CRFieldList = \begin{bmatrix} 10 & 5 & 5 & 5 \end{bmatrix}$$

$$CRWeightTable^{10} = \begin{matrix} 1 & \ldots \\ 0 & -w_2 \\ 0 & -w_4 \\ 0 & -w_7 \end{matrix}$$

$$CRValueList = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}$$

The **loadCRPen** method is called for each local constraint set containing data for algebraic constraints implemented using a penalty formulation. The C++ version of the **loadCRPen** method is given by:

```
int loadCRPen(int CRPenID,
              int numPenCRs,
              const GlobalID *const *CRNodeTable,
              const int *CRFieldList,
              const double *const *CRWeightTable,
              const double *CRValueList,
              const double *penValues,
              int lenCRNodeList);
```

The parameters passed to **loadCRPen** include the following:

- *CRPenID,* the constraint set reference identifier returned for this constraint set by the *initCRPen* initialization method,
- *numPenCRs,* the number of penalty constraints to process in this constraint set,
- *CRNodeTable,* the table of node IDs associated with this constraint, with *numCRs* rows and *lenCRNodeList* columns (each row of this table identifies a list of nodes that defines the specific constraints),
- *CRFieldList,* the list of field identifiers associated with each generic constraint, of length *lenCRNodeList* (note that field identifiers can be repeated within this list, if distinct field components at the same node are to be constrained),
- *CRWeightTable,* the table of weights associated with this constraint, with *lenCRNodeList* rows, and with the number of columns given by the field solution cardinality at each node in the node table,

---

[10] The ellipsis (...) in the table represents an entry that is not defined. Because the first row has only one column and the remaining rows have two columns, the ellipsis indicates that there is no second column in the first row. This device thus represents an attempt to cast these non-constant-length tabular data structures into a matrix form more suitable for typesetting.

- *CRValueList,* the list of non-homogenous values associated with each constraint, stored as a vector of length *numPenCRs*,
- *penValues,* the list of penalty values to be used for each constraint in the set, and
- *lenCRNodeList,* the number of columns in *CRNodeTable.*

The **endLoadCREqns** method marks the end of the constraint loading process for a particular solve step. This routine is called exactly once for each solve step. This terminal constraint loading method requires no passed parameters, and takes the following C++ form:

```
int endLoadCREqns();
```

2.2.4.5. Load Step Completion

The **loadComplete** method marks the end of the entire loading process for a particular solve step, and is called exactly once for each load phase. The **loadComplete** method takes no parameters, and takes the following C++ form:

```
int loadComplete();
```

2.2.5. Solution

The solution phase is where the linear equation solver is invoked. This step in the overall calling sequence therefore generally requires solver-specific parameters in order to solve the assembled finite-element equations. The **parameters** method permits setting these solver-specific terms, and takes the following C++ form:

```
void parameters(int numParams,
                char **paramStrings);
```

The arguments passed to the **parameters** method include the following:

- *numParams,* the number of parameters to be passed, and
- *paramStrings,* the list of *numParams* strings to be decoded by the solver for determining appropriate solution control.

An example of a call to the **parameters** method is given below, taken from the example problems presented later in this document.

```
numParams = 6;
char **paramStrings = new char*[ numParams];
for(int i = 0; i < numParams; i++)
    paramStrings[ i] = new char[ 64];

strcpy(paramStrings[ 0],"solver qmr");
strcpy(paramStrings[ 1],"preconditioner diagonal");
strcpy(paramStrings[ 2],"maxIterations 500");
strcpy(paramStrings[ 3],"tolerance 1.e-10");
```

```
strcpy(paramStrings[ 4],"rowScale false");
strcpy(paramStrings[ 5],"colScale false");

linearSystem.parameters(numParams, paramStrings);
```

Here, the QMR solver is utilized, which is appropriate for unsymmetric or indefinite systems. A diagonal preconditioner is invoked, with no additional row or column scaling. The ISIS++ package is utilized, so the strings passed represent standard ISIS++ solution commands, and a different solution module would require different numbers and types of strings.

The **iterateToSolve** method invokes the underlying solution modules, which then solve the system defined during the initialization and load phases, under the guidance of the solution control data passed via the **parameters** method. The **iterateToSolve** method requires no arguments, and takes the following C++ form:

```
int iterateToSolve();
```

Utility functions provided by the finite-element interface specification in order to support the solution process include various methods to pass initial solution vector estimates to the solver module. These methods are discussed below, under the general topic of utility functions.

2.2.6. Solution Return

Upon return from the **iterateToSolve** method, the solution components need to be returned to the client finite-element program. The interface specification provides a number of schemes for returning these solution parameters.

2.2.6.1. Blocked Solution Return Functions

The element block structure represents a fundamental subdivision of the finite-element mesh on a given processor. Therefore, this block structure provides a natural means to return the solution to the client finite-element program on a block-by-block basis. There are two classes of blocked-solution return functions: (a) one to return the nodal solution parameters for all nodes in the given block's active node list, and (b) one to return the elemental solution parameters for a given block. The first class of blocked-solution return function has two further subdivisions: (i) methods that return all the solution parameters associated with each node in the active node list, and (ii) methods that return only the solution parameters associated with a given field identifier. For the first class of solution return, the collection of nodal solution parameters is provided in the field order dictated by the finite-element client in the corresponding call to **beginInitElemBlock**.

The **getBlockNodeSolution** method returns all the nodal solution parameters associated with a given block (i.e., the one identified with *elemBlockID*). Because there is no unique way to construct the active node list for a block, this method returns a copy of the active node list so as to facilitate associating the returned solution parameters with their corresponding nodes. The C++ version of **getBlockNodeSolution** is of the following form:

```
int getBlockNodeSolution(GlobalID elemBlockID,
                         GlobalID *nodeIDList,
                         int &lenNodeIDList,
                         int *offset,
                         double *results);
```

The parameters associated with **getBlockNodeSolution** include the following:

- *elemBlockID*, the element block identifier for this solution list,
- *nodeIDList*, the returned list of nodes associated with this block (no particular order should be presumed here),
- *lenNodeIDList*, the length of *nodeIDList* (returned, but also available via a call to the utility function **getNumBlockActNodes**),
- *offset*, the list of index offsets to the first answer at each node (this list is of length *lenNodeIDList*+*1*, so that the block view of nodal solution cardinality can be recovered for every node by subtracting the nodes' offset from the next consecutive larger offset value), and
- *results*, the list of raw solution parameters at appropriate offsets. This list has length equal to the sum of all the solution cardinalities over the *lenNodeIDList* nodes that comprise the active node list.

The **getBlockFieldNodeSolution** method is similar to **getBlockNodeSolution**, except that the former returns only the solution parameters associated with a given field identifier *fieldID*.

The C++ version of **getBlockFieldNodeSolution** takes the following form:

```
int getBlockFieldNodeSolution(GlobalID elemBlockID,
                              int fieldID,
                              GlobalID *nodeIDList,
                              int& lenNodeIDList,
                              int *offset,
                              double *results);
```

The data structures required for **getBlockFieldNodeSolution** are not enumerated here, as they are otherwise the same as those needed by **getBlockNodeSolution**.

It should be noted that the calling finite-element program is charged with the responsibility of allocating memory for each of the four lists used in the **getBlockNodeSolution** and **getBlockFieldNodeSolution** calls, including providing the additional storage entry needed for the *offset* vector. In order to assist the finite-element developer in determining these storage needs, the utility functions **getNumBlockActNodes** and **getNumBlockActEqns** are provided in the interface specification. These two utility functions take as an argument the element block identifier, and return (respectively) the number of active nodes and equations associated with that element block.

The **getBlockElemSolution** method returns all the elemental solution parameters associated with a given block (i.e., the one identified with *elemBlockID*). Its C++ form is given by:

```
int getBlockElemSolution(GlobalID elemBlockID,
                         GlobalID *elemIDList,
                         int& lenElemIDList,
                         int *offset,
                         double *results,
                         int& numElemDOF);
```

The parameters associated with **getBlockElemSolution** are the following:

- *elemBlockID,* the block identifier for this solution list,
- *elemIDList,* the returned list of elements associated with this block,
- *lenElemIDList,* the returned length of *elemIDList*,
- *offset*, the returned list of index offsets to the first answer in each element,
- *results*, the returned list of raw elemental solution parameters (of length *lenElemIDList*numElemDOF*), and
- *numElemDOF*, the returned number of solution unknowns per element in this block.

In order to simplify allocation of memory for the arrays associated with this call, the utility routine **getNumBlockElements** is provided to return the number of elements associated with a given *elemBlockID*.

2.2.6.2. Constraint Parameter Return Functions

Constraint relations may or may not cause additional solution parameters to be appended to the underlying equation set, as the Lagrange multipliers represent new equations while the penalty numbers do not. The Lagrange multipliers associated with a given constraint block are returned via the **getCRMultParam** method. Its C++ form is given by:

```
int getCRMultParam(int CRMultID,
                   int numMultCRs,
                   double *multValues);
```

The parameters passed to **getCRMultParam** include the following:

- *CRMultID,* the constraint set reference identifier returned for this constraint set by the *initCRMult* initialization method,
- *numMultCRs,* the number of Lagrange multiplier constraints in this particular constraint set, and
- *multValues*, the returned list of Lagrange multiplier values for this constraint set that were computed during the solve step. This list has length *numMultCRs.*

A similar method provides all the Lagrange multipliers associated with a given processor, and this **getCRMultSolution** method represents a collective version of **getCRMultParam**. The C++ form of **getCRMultSolution** is given by:

```
int getCRMultSolution(int& numCRMultSets,
                      int *CRMultIDs,
                      int *offset,
                      double *results);
```

The parameters associated with **getCRMultSolution** include:

- *numCRMultSets,* the total number of Lagrange multiplier constraint sets on this processor (this is a return value),
- *CRMultIDs,* the returned list of constraint set reference identifiers associated with the collection of all the Lagrange multiplier constraint sets on this processor,
- *offset*, the returned list of index offsets to the first Lagrange multiplier in each constraint block (the offset list has length *numCRMultSets* and is allocated by the user's code), and
- *results*, the returned list of raw Lagrange multiplier solution parameters (this list has length equal to the sum of the number of constraints *numMultCRs* taken over all *numCRMultSets* lying on this processor, and is allocated by the user's code)

Allocating memory for these arrays is simplified via the use of the **getCRMultSizes** utility function, which returns the number of Lagrange multiplier constraint sets (*numCRMultIDs*), as well as the required length of the results vector (*lenResults*).

```
int getCRMultSizes(int& numCRMultIDs,
                   int& lenResults);
```

2.2.7. Utility Functions

Utility functions are included in the interface specification to aid in querying the interface layer for data that may be useful at various stages of the solution process. In addition to these query functions, utilities are provided to permit passing initial solution vector estimates from the finite-element client application to the solver module.

2.2.7.1. Interface Internal Query Functions

In order to facilitate the use of the interface implementation, a number of "read-only" utility functions are provided to aid the finite-element developer in determining key control parameters that can be used for allocation of memory or for program execution. These utility functions, which generally take some form of identifier, and return an internal parameter associated with that identifier, are catalogued below. In case of error, each method returns a value of –1.

The utility method **getNumSolnParams** returns the nodal solution cardinality (i.e., the total number of solution parameters defined at a node) associated with the node *globalNodeID*.

```
int getNumSolnParams(GlobalID globalNodeID);
```

The utility method **getNumElemBlocks** returns the number of element blocks on the local processor.

```
int getNumElemBlocks();
```

The utility method **getNumBlockActNodes** returns the number of active nodes associated with a given element block.

```
int getNumBlockActNodes(GlobalID blockID);
```

The utility method **getNumBlockActEqns** returns the number of active equations associated with a given element block.

```
int getNumBlockActEqns(GlobalID blockID);
```

The utility method **getNumNodesPerElement** returns the number of nodes associated with each element of a given block.

```
int getNumNodesPerElement(GlobalID blockID);
```

The utility method **getNumEqnsPerElement** returns the number of equations (including both nodal and elemental solution parameters) associated with each element of a given block.

```
int getNumEqnsPerElement(GlobalID blockID);
```

The utility method **getNumBlockElements** returns the number of elements associated with a given element block.

```
int getNumBlockElements(GlobalID blockID);
```

The utility method **getNumBlockElemEqns** returns the number of element equations associated with each element in a given element block.

```
int getNumBlockElemEqns(GlobalID blockID);
```

The utility method **getBlockNodeIDList** returns the list of active nodes associated a given element block (see the related solution return function **getBlockNodeSolution** for an explanation of the parameters passed).

```
int getBlockNodeIDList(GlobalID elemBlockID,
                       GlobalID *nodeIDList,
                       int& lenNodeIDList);
```

The utility method **getBlockElemIDList** returns the list of elements associated a given element block (see the related solution return function **putBlockElemSolution** for an explanation of the parameters passed).

```
int getBlockElemIDList(GlobalID elemBlockID,
                       GlobalID *elemIDList,
                       int& lenElemIDList);
```

2.2.7.2. Solution Initial Estimate Functions

The following functions are provided to aid the developer in passing initial solution vector estimates to the solver. These three functions can be used to pass any or all of the following components of a solution vector:

- the nodal solution parameters associated with a given block of elements,
- the element solution parameters associated with a given block of elements, and
- the Lagrange multipliers associated with a given constraint set.

In each case, the arguments are similar to those found in the associated "get" functions outlined in the solution return section above.

The initial estimate method **putBlockNodeSolution** provides a means for the finite-element program to pass initial estimates for nodal solution parameters on a block-by-block basis. The passed parameters are similar to the **getBlockNodeSolution** call, and the various arrays required can be allocated using information obtained from utility method calls, such as **getBlockNodeIDList**. The C++ form of **getBlockNodeSolution** is given by the following form:

```
int putBlockNodeSolution(GlobalID elemBlockID,
                         const GlobalID *nodeIDList,
                         int lenNodeIDList,
                         const int *offset,
                         const double *estimates);
```

The related method **putBlockFieldNodeSolution** is similar to **putBlockNodeSolution**, except that it restricts the initial estimate only to one field, given by the *fieldID* field identification parameter provided in its argument list. Its C++ form is given by:

```
int putBlockFieldNodeSolution(GlobalID elemBlockID,
                              int fieldID,
                              const GlobalID *nodeIDList,
                              int lenNodeIDList,
                              const int *offset,
                              const double *estimates);
```

The initial estimate method **putBlockElemSolution** provides a means for the finite-element program to pass initial estimates for elemental solution parameters on a block-by-block basis. The passed parameters are similar to the **getBlockElemSolution** call,

and the various arrays required can be allocated using information obtained from utility method calls, such as **getBlockElemIDList**. Its C++ form is given by:

```
int putBlockElemSolution(GlobalID elemBlockID,
                         const GlobalID *elemIDList,
                         int lenElemIDList,
                         const int *offset,
                         const double *estimates,
                         int numElemDOF);
```

The initial estimate method **putCRMultParam** provides a means for the finite-element program to pass initial estimates for Lagrange multipliers for each constraint set. The C++ form of **putCRMultParam** is given by:

```
int putCRMultParam(int CRMultID,
                   int numMultCRs,
                   const double *multEstimates);
```

2.2.8. Other Functions

In order to facilitate extending this interface specification into new venues of finite-element modeling or solver technology, various new functions may be added, or existing functions generalized via overloading. In particular, there are currently underway efforts to add support for multilevel solution techniques and to generalize some of the existing functions to provide support for eigensolution and multiple load vectors

One example is given by the **iterations** function, which returns the number of iterations required by the solver to converge to a solution (or to fail to converge, if a solution was not found). The C++ form of the **iterations** function is given below:

```
int iterations();
```

The **iterations** function takes no arguments, and returns the number of iterations, provided this function is supported by the underlying solver package.

3. Example Problems

Two classes of example problems are presented in this section:

- sample problems that demonstrate how the interface is utilized, using a simple motivating physical problem, and replete with plenty of sample C++ code, and
- more complex problem fragments that do not include sample code, but do provide insight on the structure of the data used to implement more complicated multiphysics simulations.

The first class of problems serves as a necessary introduction to the effective use of the interface, but the sample problems are simple enough so that the resulting finite-element client code will fit within this document. Hence this first class of example problems is simpler than most practical finite-element analysis codes. The second class of problems provides more complex examples that better represent high-performance finite-element simulations commonly encountered in engineering practice.

3.1. Sample Problems Demonstrating the Interface Calling Architecture

Three sample problems are presented in this section to aid developers in learning how to utilize the finite-element solution interface. The problems are designed for simplicity in each aspect that is not tightly coupled to the particulars of the process of solving finite-element equations, in order to simplify learning how the solver interface works. These restrictions include:

- one-dimensional geometry, so that the generally complicated (and highly problem- and program-dependent) relationship between node and element numbering can be avoided, and replaced with an explicit closed-form relationship between the node identifiers and the element identifiers,
- a constant solution cardinality for each node, which simplifies the sample programs by permitting considerable generic coding practice,
- a simplified version of constraint implementation, where the general linear constraint relations are utilized only to slave individual nodes to others in a simple single-master/single-slave setting, and
- no elemental solution parameters, so that all the solution unknowns are either nodal quantities, or (in the case of the distributed problem where the various domains are welded together with Lagrange multiplier constraints) Lagrange multipliers arising from a master/slave nodal constraint.

Even with these simplifying principles, these three sample problems illustrate a wide variety of finite-element development concepts that are implemented using the finite-element/solver interface specification. Thus these simple programs form a useful tool for understanding the design and philosophy of the interface specification.

3.1.1. Uniprocessor Beam Example

This simplest example problem is presented to serve as an introduction to the more complicated parallel versions that follow. Therefore, the generic aspects of the sample

problems are presented in the uniprocessor case, so that the parallel version can be utilized to focus on the particular details of multiprocessor use of the interface.

The uniprocessor problem models the flexure of a simple cantilever beam under a uniform self-weight transverse load. The relevant problem geometry is shown in Figure 10 below, where the particular mesh used has nine nodes and eight elements. The general relationship between nodes and elements in this case is that there is one more node than the number of elements. This mesh, or its obvious generalizations, will be used for all of the sample problems presented here.

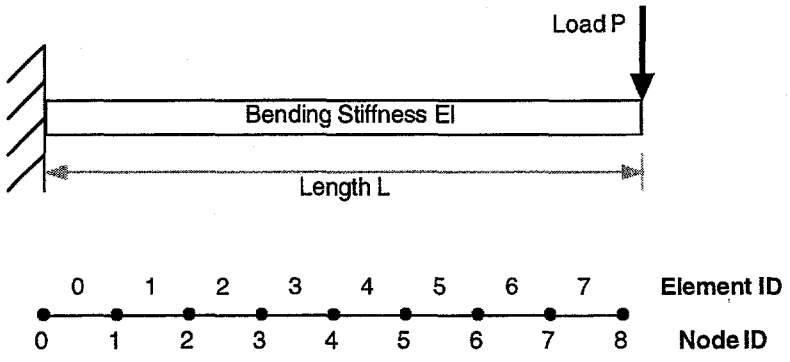


Figure 10: Cantilever beam example with associated uniprocessor mesh

After reading the parameters defining the mesh, the uniprocessor program determines a strategy to divide up the elements equitably across the range of element blocks, and then calculates element and node information for each block. The task of subdividing the mesh among the processors is performed by the **block_strategy** procedure, which allocates elements to processors so that the number of elements per block is relatively constant over all the elements. The details of this program can be found in the driver source file *fei-isis/fei-drivers/distBeamDriver.cc*, which provides both uniprocessor and shared-node multiprocessor testing for the FEI implementation.

```
block_strategy(myNumElements, myNumElemBlocks, elemsPerBlock);
int sumElem = 0;
for (i = 0; i < myNumElemBlocks; i++) {
    startElem[ i] = sumElem;
    endElem[ i] = startElem[ i] + elemsPerBlock[ i] - 1;
    startNode[ i] = startElem[ i] ;
    endNode[ i] = endElem[ i] + 1;
    nodesPerBlock[ i] = elemsPerBlock[ i] + 1;
    sumElem += elemsPerBlock[ i] ;
}
```

Once these base parameters for each element block have been computed, node *(FE_Node)* and element *(FE_Elem)* objects can be initialized, as can be seen in the source file *fei-isis/fei-drivers/distBeamDriver.cc*. In this driver program, a beam is modeled of length 10.0 with bending stiffness *EI*, axial stiffness *EA*, and applied transverse/longitudinal loads *qLoad* and *pLoad*, respectively. The element type utilized

is the usual 3-DOF/2-Node Hermitian cubic beam approximation. The various node and element class methods perform such tasks as assigning node and element ID's, setting the nodal solution cardinality and node location, assigning element beam and load properties, and other essential tasks.

Once these defining parameters have been set, the finite-element interface can be called for initialization purposes:

```
ISIS_SLE linearSystem(0);

myErrorCode = linearSystem.initSolveStep(myNumElemBlocks,
                                         mySolvType);
myErrorCode = linearSystem.initFields(myNumFields,
                                      myCardFields,
                                      myFieldIDs);
```

Note that error handling code is not presented here, in the interest of brevity.

Following the overall initialization step, the interface node and element initialization steps are called. In the code sample below, the element initialization process is shown, and a similar iteration is used to initialize the nodal data.

```
myNumElems = localElemsPerBlock;
myNumElemTotal = myNumElems;
myErrorCode = linearSystem.beginInitElemBlock(myElemBlockID,
                                              myNumNodesPerElem,
                                              myNumElemFields,
                                              myElemFieldIDs,
                                              myStrategy,
                                              myNumElemDOF,
                                              myNumElemSets,
                                              myNumElemTotal);

//  for now, to keep things simple, just use one element set per block...

myElemIDs = new GlobalID[ myNumElems] ;
myElemConn = new GlobalID* [ myNumElems] ;

int ntest;
for (k = 0; k < myNumElems; k++) {
    myElemIDs[ k]  = localStartElem + (GlobalID)k;
    myElemConn[ k]  = new GlobalID[ myNumNodesPerElem] ;
    myLocalElements[ k] .returnNodeList(ntest, myElemConn[ k] );
}
myErrorCode = linearSystem.initElemSet(myNumElems,
                                       myElemIDs,
                                       myElemConn);

myErrorCode = linearSystem.endInitElemBlock();
```

Upon completion of all initialization calls, the **initComplete** method is invoked to advise the solver of the structure of the matrix implicitly defined by the initialization calls.

```
myErrorCode = linearSystem.initComplete();
```

The load process proceeds in a manner similar to that of the initialization process, except that the particular arrays defining the boundary conditions and the element matrices are

passed. The boundary condition data is only required at the left endpoint of the beam, and the appropriate parameters for this beam problem are defined via the following initializations, which are applied only at node zero, representing the left end of the beam. Since the 3-DOF beam approximation can be modeled either as a single field consisting of 2 displacements and a rotation, or as two solution fields consisting of a plane displacement field and an out-of-plane scalar rotation parameter, the number of fields present in the sample problem is represented by the integer *myNumFields*.

```
for (k = 0; k < myNumFields; k++) {
    myAlphaTable= new double* [ myLenBCNodeSet[ k] ] ;
    myBetaTable = new double* [ myLenBCNodeSet[ k] ] ;
    myGammaTable = new double* [ myLenBCNodeSet[ k] ] ;
    for (j = 0; j < myLenBCNodeSet[ k] ; j++) {
        myAlphaTable[ j] = new double[ myNumFieldParams[ k] ] ;
        myBetaTable[ j] = new double[ myNumFieldParams[ k] ] ;
        myGammaTable[ j] = new double[ myNumFieldParams[ k] ] ;
    }
    for (i = 0; i < myLenBCNodeSet[ k] ; i++) {
        for (j = 0; j < myNumFieldParams[ k] ; j++) {
            myAlphaTable[ i][ j] = 1.0;
            myBetaTable[ i][ j] = 0.0;
            myGammaTable[ i][ j] = 0.0;
        }
    }
    myErrorCode = linearSystem.loadBCSet(myBCNodeSet,
                                         myLenBCNodeSet[ k] ,
                                         myBCFieldID[ k] ,
                                         myAlphaTable,
                                         myBetaTable,
                                         myGammaTable);
}
```

A series of **loadElemSet** calls passes the element arrays to the solver, but as this iteration is similar to the element initialization process shown above, it is not presented here. When the load step is complete, the interface method **loadComplete** is invoked, to advise the interface implementation that the finite-element equation set is now completely defined and may be prepared for solution.

```
myErrorCode = linearSystem.loadComplete();
```

Once the load step is complete, the solution process can be invoked. This step requires passing solver-dependent parameters through the interface to the solution services module via the **parameters** method. The solution process is then initiated using the **iterateToSolve** method. The **parameters** interface call shown below represents the choice of parameters appropriate for use with the ISIS++ [Clay, Mish, and Williams 1997] solution module. Other implementations of the interface, such as versions required for use with AZTEC [Hutchinson, et.al., 1995] can be constructed and/or implemented using the appropriate solver module documentation set. In the source fragment below, the QMR solver is utilized, as this solution algorithm can be applied over the entire range of example problems, including the indefinite equation set underlying the external-node Lagrange-multiplier parallel driver *fei-isis/fei-drivers/distExtBeamDriver.cc*.

```
int ii, numParams = 6;
```

```
char **paramStrings = new char*[ numParams] ;
for(ii = 0; ii < numParams; ii++) paramStrings[ ii] = new char[ 64] ;

strcpy(paramStrings[ 0] ,"solver qmr");
strcpy(paramStrings[ 1] ,"preconditioner diagonal");
strcpy(paramStrings[ 2] ,"maxIterations 50000");
strcpy(paramStrings[ 3] ,"tolerance 1.e-10");
strcpy(paramStrings[ 4] ,"rowScale false");
strcpy(paramStrings[ 5] ,"colScale false");

linearSystem.parameters(numParams, paramStrings);

linearSystem.iterateToSolve();
```

Once the solution process has been completed, the solution-return methods can be invoked to return the various classes of solution parameters (e.g., nodal, elemental, Lagrange constraint terms) desired. The following code fragment demonstrates the allocation and return (**getBlockNodeSolution**) of the nodal solution parameters for each block used in the sample mesh.

```
double *mySolnValues;
int *mySolnOffsets;
GlobalID *myNodeList;
int myLenList;
int j = myElemBlockID;

int myNumBlkActNodes = linearSystem.getNumBlockActNodes(j);
int myNumBlkActEqns = linearSystem.getNumBlockActEqns(j);
myNodeList = new GlobalID[ myNumBlkActNodes] ;
mySolnOffsets = new int[ myNumBlkActNodes] ;
mySolnValues = new double[ myNumBlkActEqns] ;

linearSystem.getBlockNodeSolution(j, myNodeList, myLenList,
                                  mySolnOffsets, mySolnValues);
```

The solution-return calls form the end of the finite-element interface logic in this uniprocessor driver program. The finite-element developer interested in using this interface layer is strongly urged to consider this uniprocessor driver (and its parallel shared-node implementation) *fei-isis/fei-drivers/distBeamDriver.cc* in detail, including browsing the associated driver source code for details not presented in the discussion above. In the following parallel examples, only the relevant differences associated with the underlying parallel program architecture will be shown. This more concise presentation provides another good reason for the finite-element developer to study the uniprocessor problem in appropriate detail.

3.1.2. Shared-Node Parallel Example (8 Elements)

The shared-node parallel sample problem geometry for the case of four processors, eight elements and nine nodes is shown in Figure 11 below. Since there is a unique shared node located at each processor boundary, the finite-element approximation is already shared between adjacent processors, so no discrete constraint relations need to be enforced at interprocessor boundaries. The generation of constraint relations to match adjacent finite-element approximations for non-shared nodes is the subject of the external-node parallel example that follows.

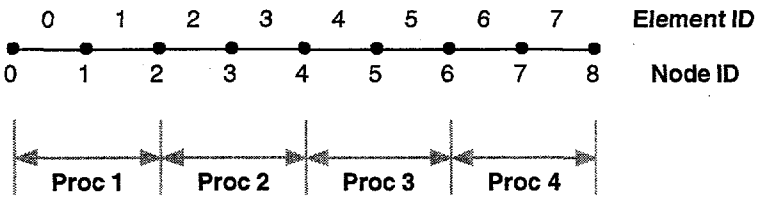


Figure 11: Mesh and processor geometry for shared-node parallel example

The most important difference between the uniprocessor example and this shared-node parallel case is that the parallel SPMD computation is implemented by having the "master" processor divide the beam into elements and element blocks, and then hand off each block to a separate processor. This necessitates some initial communications to pass basic element topology information (such as the number of nodes and elements to be found on each processor, and their start/end ID's), as shown in the following code fragments. It should be noted that in a production finite-element setting, a separate domain-decomposition application is generally used *a priori* to partition the mesh, and this approach obviates the need for the "master processor" required in the driver.

```
//  send all this overhead data to the various processors and then
//  parallel computation can get done...

for (i = 1; i < myNumElemBlocks; i++) {
     MPI_Send(&myNumElements, 1, MPI_INT, i, s_tag1, MPI_COMM_WORLD);
     MPI_Send(&myNumElemBlocks, 1, MPI_INT, i, s_tag1, MPI_COMM_WORLD);
     MPI_Send(&startElem[ i] , 1, MPI_GLOBALID, i, s_tag1, MPI_COMM_WORLD);
     MPI_Send(&endElem[ i] , 1, MPI_GLOBALID, i, s_tag1, MPI_COMM_WORLD);
     MPI_Send(&elemsPerBlock[ i] , 1, MPI_INT, i, s_tag1, MPI_COMM_WORLD);
     MPI_Send(&startNode[ i] , 1, MPI_GLOBALID, i, s_tag1, MPI_COMM_WORLD);
     MPI_Send(&endNode[ i] , 1, MPI_GLOBALID, i, s_tag1, MPI_COMM_WORLD);
     MPI_Send(&nodesPerBlock[ i] , 1, MPI_INT, i, s_tag1, MPI_COMM_WORLD);
}

//  if I'm not the master, receive the data sent by the master CPU...

MPI_Recv(&myNumElements, 1, MPI_INT, masterRank, s_tag1,
          MPI_COMM_WORLD, &status);
MPI_Recv(&numBlocks, 1, MPI_INT, masterRank, s_tag1,
          MPI_COMM_WORLD, &status);
MPI_Recv(&localStartElem, 1, MPI_GLOBALID, masterRank, s_tag1,
          MPI_COMM_WORLD, &status);
MPI_Recv(&localEndElem, 1, MPI_GLOBALID, masterRank, s_tag1,
          MPI_COMM_WORLD, &status);
MPI_Recv(&localElemsPerBlock, 1, MPI_INT, masterRank, s_tag1,
          MPI_COMM_WORLD, &status);
MPI_Recv(&localStartNode, 1, MPI_GLOBALID, masterRank, s_tag1,
          MPI_COMM_WORLD, &status);
MPI_Recv(&localEndNode, 1, MPI_GLOBALID, masterRank, s_tag1,
          MPI_COMM_WORLD, &status);
MPI_Recv(&localNodesPerBlock, 1, MPI_INT, masterRank, s_tag1,
          MPI_COMM_WORLD, &status);
```

The next relevant difference between the uniprocessor and parallel/shared-node case arises from the fact that the shared nodes must be identified via appropriate interface calls. The shared-node parallel driver aids in this step by considering the various ways that shared nodes arise from blocks at either end of the problem (which have only one set of shared nodes), or from internal blocks (which have two sets, one at each end). The requisite logic is shown in the code sample below, where the *localRank* parameter defines the processor ID.

```
if (numProcessors == 1) {
    numSharedNodes = 0;
}
else if (localRank == 0) {
    numSharedNodes = 1;
    listSharedNodes = new GlobalID[ numSharedNodes] ;
    listSharedProcs = new int* [ numSharedNodes] ;
    listSharedProcs[ 0] = new int [ 2] ;
    listSharedNodes[ 0] = localEndNode;
    listSharedProcs[ 0][ 0] = localRank;
    listSharedProcs[ 0][ 1] = localRank + 1;
}
else if (localRank == (numProcessors - 1)) {
    numSharedNodes = 1;
    listSharedNodes = new GlobalID[ numSharedNodes] ;
    listSharedProcs = new int* [ numSharedNodes] ;
    listSharedProcs[ 0] = new int [ 2] ;
    listSharedNodes[ 0] = localStartNode;
    listSharedProcs[ 0][ 0] = localRank - 1;
    listSharedProcs[ 0][ 1] = localRank;
}
else {
    numSharedNodes = 2;
    listSharedNodes = new GlobalID[ numSharedNodes] ;
    listSharedProcs = new int* [ numSharedNodes] ;
    listSharedProcs[ 0] = new int [ 2] ;
    listSharedProcs[ 1] = new int [ 2] ;
    listSharedNodes[ 0] = localStartNode;
    listSharedNodes[ 1] = localEndNode;
    listSharedProcs[ 0][ 0] = localRank - 1;
    listSharedProcs[ 0][ 1] = localRank;
    listSharedProcs[ 1][ 0] = localRank;
    listSharedProcs[ 1][ 1] = localRank + 1;
}
```

The arrays defined in this code fragment aid the driver in determining which nodes are shared, and these arrays are converted to the form required by the finite-element interface implementation via the following logic.

```
//  pass the shared node data (each shared node is shared between two CPU's)

for (i = 0; i < myNumSharedNodeSets; i++) {
     myLenSharedNodeSet = numSharedNodes;
     mySharedNodes = new GlobalID[ myLenSharedNodeSet] ;
     myLenSharedProcIDs = new int[ myLenSharedNodeSet] ;
     mySharedProcIDs = new int* [ myLenSharedNodeSet] ;

     for (j = 0; j < myLenSharedNodeSet; j++) {
          mySharedNodes[ j] = listSharedNodes[ j] ;
          myLenSharedProcIDs[ j] = 2;
          mySharedProcIDs[ j] = new int[ myLenSharedProcIDs[ j]] ;
```

```
            for (k = 0; k < myLenSharedProcIDs[ j] ; k++) {
                mySharedProcIDs[ j][ k] = listSharedProcs[ j][ k] ;
            }
        }

//  FE interface call ---------------------------------------------

        myErrorCode = linearSystem.initSharedNodeSet(mySharedNodes,
                                                     myLenSharedNodeSet,
                                                     mySharedProcIDs,
                                                     myLenSharedProcIDs);

        delete [] mySharedNodes;
        for (j = 0; j < myLenSharedNodeSet; j++) {
            delete [] mySharedProcIDs[ j] ;
        }
        delete [] mySharedProcIDs;
}
```

The rest of the shared-node/parallel driver program is similar to the uniprocessor example case, including the solution invocation and solution return steps. Therefore, these redundant code components are not shown here.

The shared-node parallel problem demonstrates overall SPMD utilization of the interface implementation, but it misses some important areas of the interface code. In particular, the shared-node problem does not exercise any portions of the interface layer that implement constraint equations. These methods are presented in the next example.

3.1.3. External-Node Parallel Example (8 Elements)

In order to examine the interface layer's handling of distributed-memory constraint relations, a similar beam problem can be modeled, but this more complex example does not utilize shared-node logic to enforce solution compatibility across processor boundaries. Instead, each processor has a completely separate component of the finite-element mesh, and the solution is constrained across processor boundaries by discrete constraint relations (using either the Lagrange multiplier or penalty constraint handling facilities of the interface, in the source files *fei-isis/fei-drivers/distExtBeamDriver.cc* and *fei-isis/fei-drivers/distExtPenDriver.cc*, respectively) on the displacement and slope at each inter-processor junction. The general topology of a four-processor/eight-element/twelve-node mesh is shown in Figure 12 below.

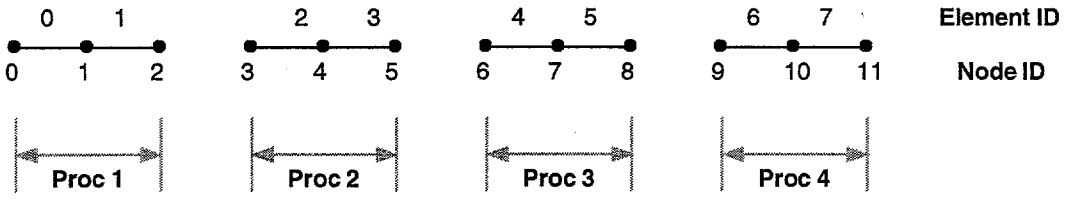


Figure 12: Mesh and processor geometry for external-node, parallel example

The main differences between this example and the previous cases are the following:

- this external-node example involves a slightly more complex node-numbering scheme, as the simple "off-by-one" relationship between nodes and elements present in the shared-node and uniprocessor cases does not apply,
- logic is required to determine the data required to identify and to initialize the external nodes found at the boundaries of each block, and
- there are two field constraints at each block junction, one enforcing displacement continuity and the other enforcing slope continuity.

The first difference is easy to develop, so it will not be treated further here. The second and third cases will be briefly outlined below with sample code from the driver. The nodal data initialization pass requires specification of the fixed-end boundary conditions at node 0, as well as the identification of external node sets for each block of elements. There are two sets of external nodes arising from the constraints (one at each end) for each internal element, and one set for blocks at either endpoint, as can be seen from the following code fragment.

```
if (numProcessors == 1) {
    myNumExtSendNodeSets = 0;
    myNumExtRecvNodeSets = 0;
}
else if (localRank == 0) {
    myNumExtSendNodeSets = 0;
    myNumExtRecvNodeSets = 1;
}
else if (localRank == (numProcessors - 1)) {
    myNumExtSendNodeSets = 1;
    myNumExtRecvNodeSets = 0;
}
else {
    myNumExtSendNodeSets = 1;
    myNumExtRecvNodeSets = 1;
}
int myNumExtNodeSets = myNumExtSendNodeSets + myNumExtRecvNodeSets;
myErrorCode = linearSystem.beginInitNodeSets(myNumSharedNodeSets,
                                             myNumExtNodeSets);
```

Identification of the external nodes involves logic similar to that for the shared-node case, so this process is not presented here. The actual constraint logic (for the case of a single solution field representing all three solution parameters, with each component of the solution tied across processor boundaries with a separate constraint) is outlined below.

```
if (myNumCRMultSets > 0) {
    myNumCRs = 1;          // 1 eqn (disp or slope) in each constraint set
    myLenCRList = 2;       // 2 nodes (one per block) in each constraint
    myCRMultIDList = new int[ myNumCRMultSets] ;
    myCRMultFieldIDList = new int [ myLenCRList] ;
    myCRMultFieldIDList[ 0] = myFieldIDs[ 0] ;   // here, tie the same field
    myCRMultFieldIDList[ 1] = myFieldIDs[ 0] ;   // at each end...
    for (j = 0; j < myNumCRMultSets; j++) {
        myCRNodeTable = new GlobalID* [ myNumCRs] ;
        for (k = 0; k < myNumCRs; k++) {
            myCRNodeTable[ k] = new GlobalID[ myLenCRList] ;
            myCRNodeTable[ k][ 0] = localEndNode;
            myCRNodeTable[ k][ 1] = localEndNode;
            myCRNodeTable[ k][ 1] ++;
```

```
            }
            myErrorCode = linearSystem.initCRMult(myCRMultIDList[ j],
                                                  myCRNodeTable,
                                                  myCRMultFieldIDList,
                                                  myNumCRs,
                                                  myLenCRList);

            for (i = 0; i < myNumCRs; i++) {
                delete [] myCRNodeTable[ i];
            }
            delete [] myCRNodeTable;
        }
    }
```

Similar logic is used in the load step to handle the weights required to implement the constraint, each of which is of the general form given by:

$$u_{left} = u_{right} \qquad \text{or} \qquad u_{left} - u_{right} = 0$$

where the subscripts *left* and *right* indicate the nodes located on the left and right sides of the interprocessor node-numbering gaps, and the solution parameter u is taken either as the transverse displacement, axial displacement, or the out-of-plane rotation.

Other than these differences, this external node sample problem is similar to the shared-node parallel example.

Because each constraint involves a combination of nodes local to and external to any given processor, this example not only demonstrates general constraint implementation, but also the associated external-node, distributed-memory logic to handle off-processor communications. For the Lagrange multiplier version of the constraint relations, the resulting multipliers represent physically important parameters (in this case, the internal forces in the beam at the points where the continuity constraints are applied). So, this external node test problem also demonstrates the solution return functions for Lagrange multiplier constraint sets.

3.2. Sample Problem Results

Results from the sample problems are shown in the figures below, for the case of the external node problem with two constraints per node (transverse displacement and out-of-plane rotation) and eight elements located on four processors. Figure 13 shows the transverse beam displacement, while Figure 14 graphs the beam rotation. Both of these solution parameters agree perfectly with the theoretical results, which are plotted as solid lines in these figures – this agreement is an example of the well-known finite-element characteristic of *superconvergence*, which is a property of the underlying finite-element model, and not of any interface construct. Details on superconvergence, or on other aspects of finite-element approximation can be found in a variety of references, including the textbooks of Hughes [Hughes,1987] and Zienkiewicz [Zienkiewicz and Taylor, 1991].

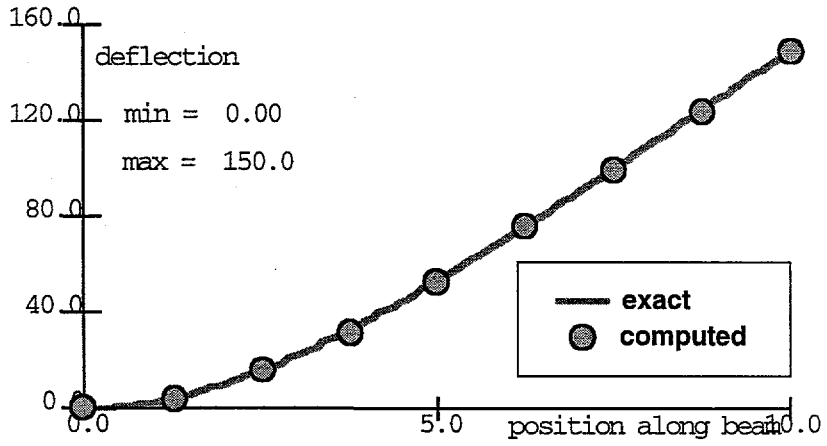


Figure 13: Results for beam transverse displacement at nodes

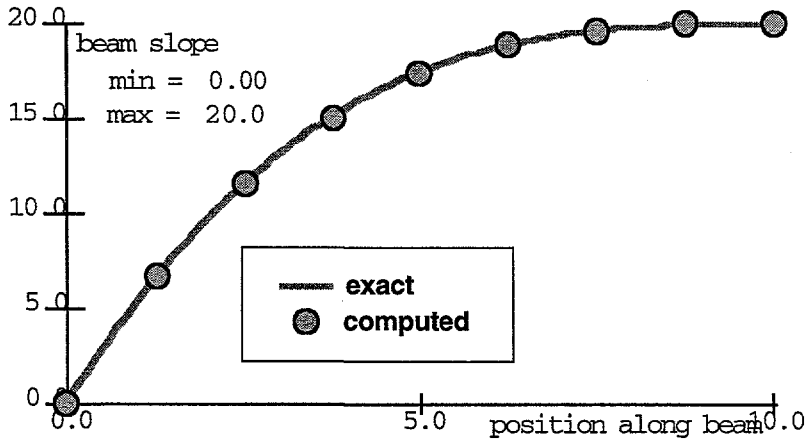


Figure 14: Results for beam slope at nodes

Figures 13 and 14 also provide the same displacement/rotation results as would be found in the uniprocessor and shared-node problems.

Figures 15 and 16 show the beam's internal force state, with the first figure plotting beam bending moment and the second showing transverse beam shear. The three points shown overlaying the curves are the computed Lagrange multipliers for the three junctions among the four processors. These results demonstrate that the Lagrange multipliers calculated during the constraint implementation process have the physical interpretation of internal forces within the beam that are required to maintain the constraints.

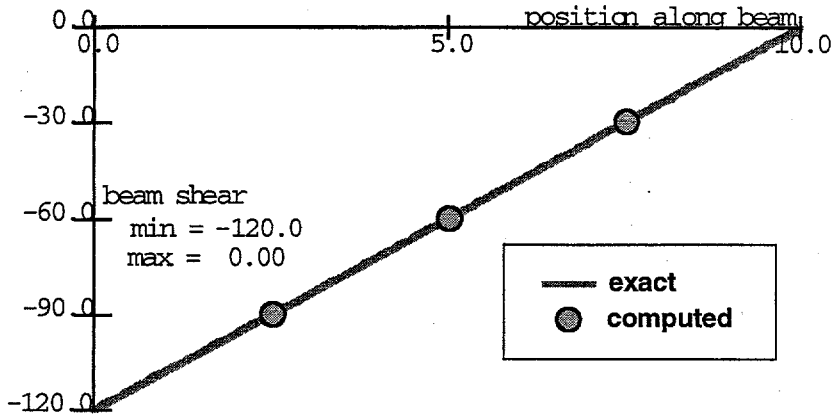


Figure 15: Beam shear (Lagrange multiplier for displacement constraints)

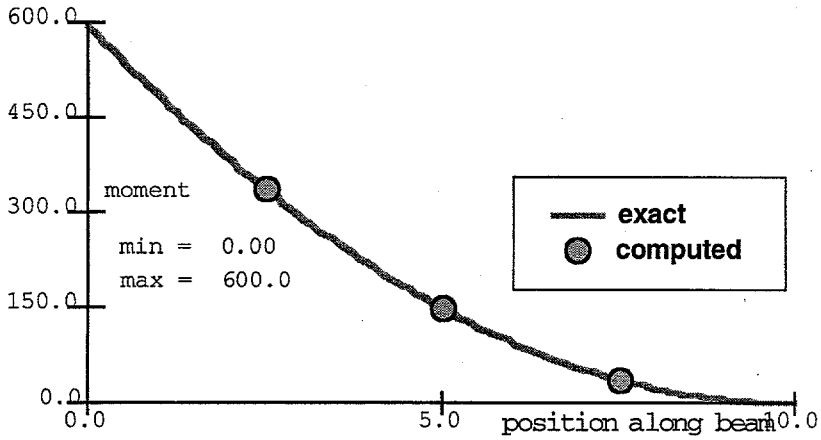


Figure 16: Beam moment (Lagrange multiplier for slope constraints)

3.3. Examples from Multiphysics Simulations and Other Special Cases

The previous examples demonstrate the use of the finite-element interface in considerable detail, but in a setting that is simpler than what is normally found in practice. The following examples present less programming detail, but provide insights on more complicated finite-element analyses, and especially on how multiphysics problems can readily be handled using the abstractions present in the interface specification.

3.3.1. Field and Element Block Example

This simple two-dimensional example problem demonstrates how field information and element block information interact in the setting of a standard multiphysics problem. The sample presented involves the specification for a block of two-dimensional elements to solve thermoelasticity problems in the setting of an incompressibility constraint. The incompressible formulation results in mixed finite-element formulation, so that the

displacement field must be accompanied by a pressure field that serves as a Lagrange multiplier function whose purpose is to enforce the distributed incompressibility constraint. The mixed formulation requires certain consistency conditions to be satisfied between the displacement and pressure fields (see the text by Hughes [Hughes,1987] for details), and the resulting consistent mixed element used in this example is a nine-node biquadratic Lagrange element, with a three-node internal linear pressure expansion. The thermal response of the element is decoupled from the mixed displacement/pressure interpolation, so a simpler four-node bilinear Lagrange element is sufficient for modeling the temperature field.

The resulting twelve-node element approximations are diagrammed in the Figure 17 below. Note that the "nodes" for the pressure degrees of freedom can be located at any location within the element sufficient to define a linear pressure field of the form

$$p(x,y)=a+bx+cy$$

This pressure field can also be idealized as a three-term elemental solution parameter list, in which case the whole notion of "pressure nodes" can be dispensed with in favor of an interpretation of the discontinuous pressure field as a purely elemental concept.

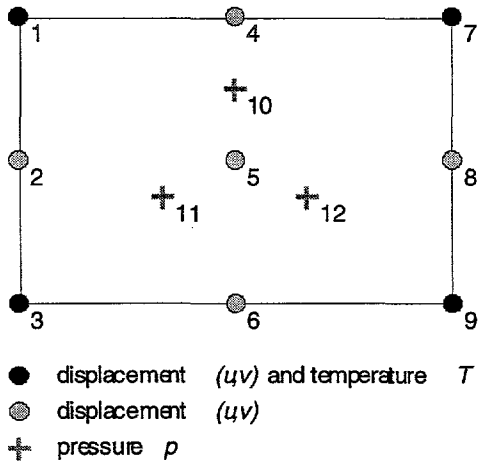


Figure 17: Example 2D multiphysics element

In this example, the field identifiers for displacement, temperature, and pressure are given respectively by the fieldIDs 100, 101, and 102. With this convention, the initialization call to **initFields** will utilize the following data:

numFields: 3

cardFields: [2 1 1]

fieldIDs: [100 101 102]

The data to be passed to the **beginInitElemBlock** function is given by the following:

numNodesPerElement: 12

numElemFields: [2 1 2 1 1 1 2 1 2 1 1 1]

elemFieldIDs:

100 101
100 ...
100 101
100 ...
100 ...
100 ...
100 101
100 ...
100 101
102 ...
102 ...
102 ...

numElemDOF: 0

Note that some of the parameters passed to **beginInitElemBlock** (namely *elemBlockID*, *interleaveStrategy*, *numElemSets*, and *numElemTotal*) are not germane to this example, and hence they are not considered here.

Since the pressure field here is discontinous across element boundaries, the solution parameters that define the pressure over a given element can be eliminated at the element level to produce a simpler element block definition that requires finite-element interpolation of only the displacement and temperature fields. In this alternative formulation, the pressure nodes would be identified as internal element unknowns, in which case the three pressure nodes would be dropped from consideration within elements from this block definition. This approach would also remove the pressure field from explicit description within the finite-element interface calls. The resulting parameters passed to these initialization routines would then take the following form.

numFields: 2

cardFields: [2 1]

fieldIDs: [100 101]

numNodesPerElement: 9

numElemFields: [2 1 2 1 1 1 2 1 2]

	100 101
	100 ...
	100 101
	100 ...
elemFieldIDs:	100 ...
	100 ...
	100 101
	100 ...
	100 101
numElemDOF:	3

3.3.2. Rotated Boundary-Condition Example

The next example demonstrates the manner in which boundary conditions can be applied in directions not aligned with the underlying coordinate system of the analysis. The example considered is that of a cantilever beam propped at its terminal end with an inclined frictionless roller support. The geometry of the sample problem is shown in Figure 18 below, where the element numbered *m* is connected to a rotated roller support at node 13.

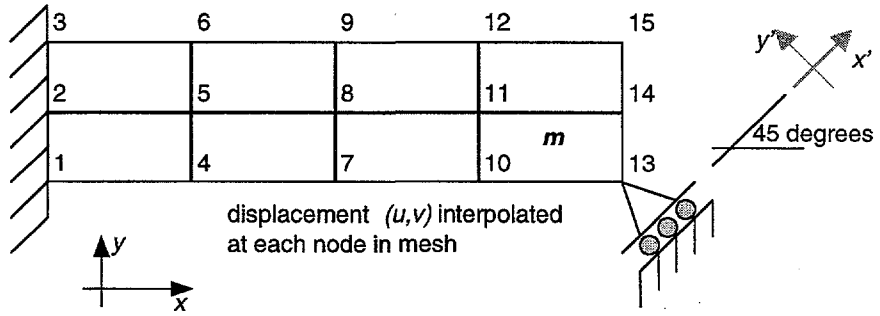


Figure 18: Example 2D boundary-condition handling

Because of the finite-element interface's wide latitude in specifying field-oriented solution data (and because rotations of solution fields clearly belong to the "physics" side of the physics/algebra divide), the finite-element interface specification does not provide for the direct implementation of the rotated boundary condition at node 13. Instead, the finite-element client code bears the responsibility for the following tasks:

- recognizing that the rotated boundary condition does not lie along the natural coordinate directions of the problem,
- synthesizing a new rotated coordinate system for the boundary specification (represented by the *x'*-*y'* system in Figure 18 above),

- passing the appropriate data for the boundary condition in the rotated coordinate system via the **loadBCSet** method, and
- transforming appropriate data structures (in this case, the finite-element stiffness matrix and load vectors) into the rotated coordinate system before calling the **loadElemSet** method.

The last task is beyond the scope of this document, but its implementation details can be found in standard finite-element references such as Zienkiewicz [Zienkiewicz and Taylor, 1991]. The general process involves transforming the underlying finite-element data structures in a manner consistent with their tensorial nature. In this example case, where the solution field is vector-valued, this would require premultiplication of the element load vector for element *m* by the matrix of direction cosines, and a left- and right-hand matrix transformation by the direction cosines for the element stiffness matrix. Since none of the other elements in the mesh are associated with the rotated boundary condition, these would constitute the only transformations required for the load step.

The data structures required for enforcement of the essential boundary conditions at the left end of the beam have already been outlined earlier in this document, in the discussion of the **loadBCSet** method. Therefore, only the enforcement of the inclined roller condition at node 13 will be presented below.

In order to satisfy the rotated essential (i.e., displacement) boundary condition at node 13, the displacement must vanish in the *x'* direction. This rotated specification casts the rotated essential boundary condition into the following data structures:

LenBCNodeSet:	1
BCNodeSet:	[13]
alphaBCDataTable:	[1 0]
betaBCDataTable:	[0 0]
gammaBCDataTable:	[0 0]

The roller support provides a natural (i.e., force) boundary condition in the *y'* direction, so that the force (i.e., the dual of the displacement, which in this mechanical case can be identified as the associated quantity whose product with the displacement has the physical interpretation of work) specified along this direction vanishes. This condition can be implemented using the various data structures shown below:

LenBCNodeSet:	1
BCNodeSet:	[13]
alphaBCDataTable:	[0 0]

betaBCDataTable: [0 1]

gammaBCDataTable: [0 0]

A group of nodes containing a generic boundary condition of this type can be aggregated into a nodeSet collection, in which case the *BCNodeSet* list will have multiple entries, and the various tables that define the boundary condition will not have the degenerate form shown in the example above (i.e., they will form tables instead of row vectors).

3.3.3. Shell-Continuum Junction Example

This three-dimensional example demonstrates a fairly complex set of algebraic constraints that join two different mesh fragments with different views of the nodal solution cardinality. The geometry of this problem has been chosen to simplify the presentation of the requisite constraint data, but the problem itself is representative of a wide class of common structural analysis applications. There are some mathematical details that are not relevant to the particulars of this example problem, and these niceties are thus relegated to footnote status.

The sample problem involves the junction of a shell with a three-dimensional continuum. While the continuum elements are each associated with a single solution field (namely the displacement field), the shell elements also interpolate a second solution field, the pseudo-rotations[11] in the three coordinate directions. The process of embedding these rotational solution parameters into appropriate continuum displacements constitutes the focus of this sample problem.

The geometry of this sample problem is shown in Figure 19 below, where a collection of four-node bilinear shell elements is to be joined to a mesh composed of eight-node trilinear continuum elasticity elements.

---

[11] Strictly speaking, the rotation field in a solid is *not* a vector, but instead is naturally represented by an antisymmetric second-rank tensor. For purposes of approximation, however, it is traditional practice to store the components of this tensor as a list (or vector) of scalar rotations about the coordinate axes. While the rotation field is mathematically associated with a so-called *axial vector* which contains the three independent parameters that define the rotation tensor, this axial vector involves rearrangements and sign changes that are not of relevance to this example problem. Furthermore, since the whole issue of a standard solution sign convention for analysis of plates and shells is an open question in mechanics, this example problem avoids *all* of these issues by merely representing the shell's nodal rotation field as a vector collection of three scalar components, denoted by *r, s,* and *t.* For the reader interested in a more careful exposition of this analysis detail, the text by Hughes [Hughes,1987] is particularly recommended.

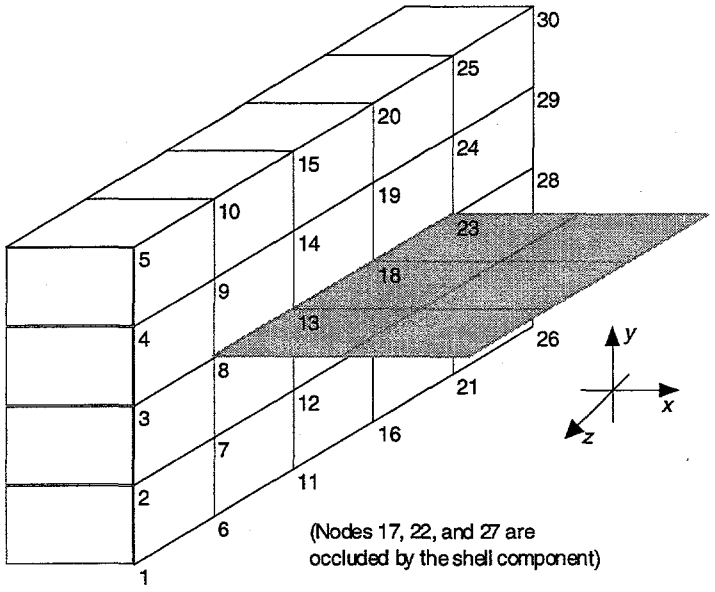


Figure 19: Example 3D constraint handling

Because the shells are analyzed using two solution fields (displacement and rotation vectors), the process of coupling the shell approximation and the continuum model at the common nodes (nodes 8, 13, 18, and 23) consists of two separate steps:

- coupling the displacement fields of the two domains (a process that occurs automatically in this finite-element analysis, because each element approximation is conformable with the other, so that the use of a common node numbering system at the junction provides inter-element approximation compatibility across the continuum-shell junction), and
- coupling the shell's rotation field to the continuum's displacement field (a process that requires satisfaction of a set of discrete algebraic constraints at the common nodes 8, 13, 18, and 23).

The two solution fields present in this example problem are diagrammed in Figure 20 below, where the individual element blocks are associated with their respective solution fields registered via appropriate calls to **beginInitElemBlock**.

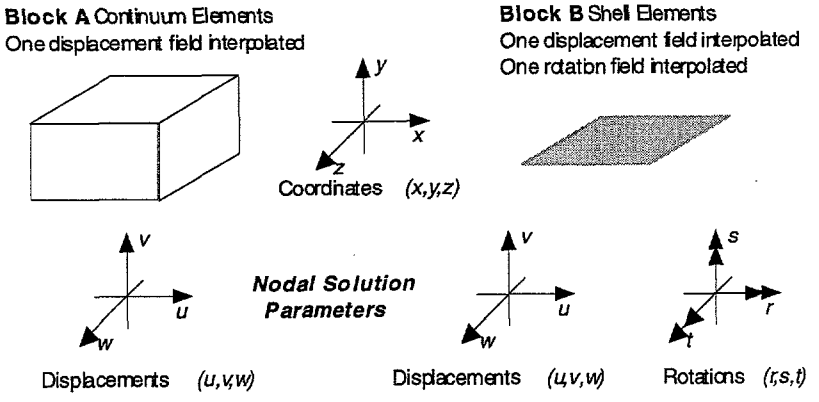


Figure 20: Element-field associations

The discrete algebraic constraints are constructed by combining normal displacement components in directions corresponding to the sense of the rotation components. The various solution components are tabulated below:

This rotation component:	*Depends upon these displacement components:*
r_8	v_3, w_7, v_8, w_8, w_9, *and* v_{13}
s_8	u_3, u_8, *and* u_{13}
t_8	u_7, u_8, *and* u_9
r_{13}	v_8, w_{12}, v_{13}, w_{13}, w_{14}, *and* v_{18}
s_{13}	u_8, u_{13}, *and* u_{18}
t_{13}	u_{12}, u_{13}, *and* u_{14}
r_{18}	v_{13}, w_{17}, v_{18}, w_{18}, w_{19}, *and* v_{23}
s_{18}	u_{13}, u_{18}, *and* u_{23}
t_{18}	u_{17}, u_{18}, *and* u_{19}
r_{23}	v_{18}, w_{22}, v_{23}, w_{23}, w_{24}, *and* v_{28}
s_{23}	u_{18}, u_{23}, *and* u_{28}
t_{23}	u_{22}, u_{23}, *and* u_{24}

If the three sets of scalar weights ξ, η, and ζ are introduced for purposes of representing the r, s, and t shell rotation solution parameters in terms of the common displacement

field components *u, v,* and *w*, then the twelve relations tabulated above can be cast into three groups of generic algebraic constraints[12].

x-component of rotation:

$$r_8 = \xi_a v_3 + \xi_b w_7 + \xi_c v_8 + \xi_d w_8 + \xi_e w_9 + \xi_f v_{13}$$
$$r_{13} = \xi_a v_8 + \xi_b w_{12} + \xi_c v_{13} + \xi_d w_{13} + \xi_e w_{14} + \xi_f v_{18}$$
$$r_{18} = \xi_a v_{13} + \xi_b w_{17} + \xi_c v_{18} + \xi_d w_{18} + \xi_e w_{19} + \xi_f v_{23}$$
$$r_{23} = \xi_a v_{18} + \xi_b w_{22} + \xi_c v_{23} + \xi_d w_{23} + \xi_e w_{24} + \xi_f v_{28}$$

y-component of rotation:

$$s_8 = \eta_a u_3 + \eta_b u_8 + \eta_c u_{13}$$
$$s_{13} = \eta_a u_8 + \eta_b u_{13} + \eta_c u_{18}$$
$$s_{18} = \eta_a u_{13} + \eta_b u_{18} + \eta_c u_{23}$$
$$s_{23} = \eta_a u_{18} + \eta_b u_{23} + \eta_c u_{28}$$

z-component of rotation:

$$t_8 = \zeta_a u_7 + \zeta_b u_8 + \zeta_c u_9$$
$$t_{13} = \zeta_a u_{12} + \zeta_b u_{13} + \zeta_c u_{14}$$
$$t_{18} = \zeta_a u_{17} + \zeta_b u_{18} + \zeta_c u_{19}$$
$$t_{23} = \zeta_a u_{22} + \zeta_b u_{23} + \zeta_c u_{24}$$

Note that in this example, the generic nature of the weights that multiply the various displacement solution components arises from the fact that the mesh is sufficiently regular so that the geometric properties (e.g., the dimensions of the continuum elements in the *yz* plane) that underlie these constraint relations are the same for each constraint. If the mesh dimensions were not regular (e.g., if the mesh were graded in the *y* direction), then these constraints would have to be ungrouped appropriately. The generic nature of these constraints has been chosen specifically to demonstrate here how individual algebraic constraints can be grouped into constraint sets.

The data structures used in **loadCRMult** or **loadCRPen** for these three sets of constraints are given below. In each case, the field identifier for the displacement vector is taken as *5* and the field identifier for the rotation vector is taken as *10*.

---

[12] These constraints only apply to a small-deformation finite-element analysis. A consistent linearization is required to obtain kinematical relations that are appropriate for a large-deformation analysis. The details of linearizing these kinematic relations are beyond the scope of this illustrative example.

x-component of rotation:

NumMultCRs: 4

LenCRNodeList: 5

CRNodeTable:

$$\begin{matrix} 8 & 3 & 7 & 8 & 9 & 13 \\ 13 & 8 & 12 & 13 & 14 & 18 \\ 18 & 13 & 17 & 18 & 19 & 23 \\ 23 & 18 & 22 & 23 & 24 & 28 \end{matrix}$$

CRFieldList: $\begin{bmatrix} 10 & 5 & 5 & 5 & 5 & 5 \end{bmatrix}$

CRWeightTable:

$$\begin{matrix} -1 & 0 & 0 \\ 0 & \xi_a & 0 \\ 0 & 0 & \xi_b \\ 0 & \xi_c & \xi_d \\ 0 & 0 & \xi_e \\ 0 & \xi_f & 0 \end{matrix}$$

CRValueList: $\begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}$

y-component of rotation:

NumMultCRs: 4

LenCRNodeList: 4

CRNodeTable:

$$\begin{matrix} 8 & 3 & 8 & 13 \\ 13 & 8 & 13 & 18 \\ 18 & 13 & 18 & 23 \\ 23 & 18 & 23 & 28 \end{matrix}$$

CRFieldList: $\begin{bmatrix} 10 & 5 & 5 & 5 \end{bmatrix}$

CRWeightTable:

$$\begin{matrix} 0 & -1 & 0 \\ \eta_a & 0 & 0 \\ \eta_b & 0 & 0 \\ \eta_c & 0 & 0 \end{matrix}$$

CRValueList: $\begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}$

z-component of rotation:

NumMultCRs: 4

LenCRNodeList: *4*

CRNodeTable:

$$\begin{matrix} 8 & 7 & 8 & 9 \\ 13 & 12 & 13 & 14 \\ 18 & 17 & 18 & 19 \\ 23 & 22 & 23 & 24 \end{matrix}$$

CRFieldList: $[10 \quad 5 \quad 5 \quad 5]$

CRWeightTable:

$$\begin{matrix} 0 & 0 & -1 \\ \zeta_a & 0 & 0 \\ \zeta_b & 0 & 0 \\ \zeta_c & 0 & 0 \end{matrix}$$

CRValueList: $[0 \quad 0 \quad 0 \quad 0]$

4. Related Documents

This document provides an annotated version of the finite-element interface specification, but it does not provide substantial material on the motivations or design principles behind the finite-element interface, or other issues regarding its use.

In order to provide detail on these (and other) important topics, a few other associated documents are available for use by developers of finite-element clients or solution services modules. These auxiliary documents currently include:

4.1. Header Files

The C++ and ANSI C header files fei.h and cfei.h represent the ultimate realization of the interface specification, as viewed by the C++ compiler, and as such constitutes the authoritative documentation of the interface. It should be utilized to determine the type, size, and calling sequence for all passed parameters, though its internal documentation is sufficiently terse so that programmers will want to read the accompanying interface specification.

4.2. Interface Specification Document

This document **[Clay, et.al., 1999]** provides motivation for the design of the finite-element interface layer, as well as fundamental theory required to understand the design goals of the particular interface implementation. By itself, it introduces the software engineer to the "why" of the finite-element interface, as well as many of the details as to "how" or "when". It does not provide a full suite of test problems, however, in large part because the wide generality of finite-element analyses make the task of providing self-contained documentation for all aspects of finite-element equation solution (including a diverse range of sample problems) prohibitively difficult.

4.3. Example Problem Suite

The interface distribution includes a diverse range of sample problems that can be used to demonstrate and test the interface implementation. These drivers include the following source files, all located in the *fei-isis/fei-drivers/* subdirectory:

distBeamDriver.cc	uniprocessor and shared-node parallel code
distCFEBeamDriver.c	ANSI-C version of *distBeamDriver.cc*
distExtBeamDriver.cc	external-node parallel code with Lagrange constraints
distExtPenDriver.cc	external-node parallel code with penalty constraints

5. Acknowledgements

The development group would like to thank the following people who contributed to this effort.

- Colin Aro and the ALE3D team at LLNL for being such great alpha testers.
- Jim Schutt, Johnny Biffle, Carter Edwards and the Sierra team at SNL for design input, road testing, and general support.

References

5.1. General References

Clay, R.L., Mish, K.D., and Williams, A.B., *ISIS++ Reference Guide (Iterative Scalable Implicit Solver in C++) Version 1.0*, Technical Report SAND97-8535, Sandia National Laboratories, Livermore, CA, 1997

Clay, R.L., Mish, K.D., Otero, I., Taylor, L.M., and Williams, A.B., *A Proposed General Finite-Element/Solver Interface Specification, Version 1.0*, Technical Report SAND99-8230, Sandia National Laboratories, Livermore, CA, 1999

Hughes, T.J.R., *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*, Prentice Hall, Englewood Cliffs, New Jersey, 1987

Hutchinson, S., Shadid, J. and Tuminaro, R., *Aztec Users Guide version 1.1*, Technical Report SAND95-1559, Sandia National Laboratories, Albuquerque NM, 1995

Zienkiewicz, O.C. and Taylor, R.L., *The Finite Element Method, 4th Edition*, McGraw-Hill Book Company, UK, Berkshire, England, 1991

5.2 *URLs for Related Documents*

FEI Home Page	*http://z.ca.sandia.gov/fei/*
ISIS++ Home Page	*http://z.ca.sandia.gov/isis/*
ESI Home Page	*http://z.ca.sandia.gov/esi/*

6. Appendix A: Glossary of Finite-Element Abstractions

Active Node List: the set of all nodes associated with elements located on a given processor. This enumeration is more precisely termed *the processor active node* list, in order to distinguish it from its blocked subset.

Array: a generic term used in this document to refer to a derived and indexed data type that includes lists, tables, vectors, matrices, or any other similar non-scalar data.

Assembly: the process by which compressed element matrices are accumulated into their respective positions into the system matrices, including matrix assembly (for the element stiffness matrices) and vector assembly (for the element load vectors). This operation combines an accumulation with a scatter operation for each element coefficient term.

Constraint: an algebraic relation that must be satisfied by the finite-element approximation as a side condition. Such constraints may be local (such as the linear algebraic relation implied by an impenentrability condition between two parts of the finite-element mesh), or they may be global (such as the overall divergence-free constraint required in some incompressible solid or fluid mechanics problems).

ConstrSet: an aggregation of constraint relations into a generic form.

Element: a geometric subdomain that forms one individual component of the material domain for the boundary-value-problem that is being modeled. The rules for constructing elements are bewilderingly diverse, but the basic idea is that the elements tile the problem domain so that the intergrals that define the underlying energy functional can be decomposed into a discrete sum of elemental contributions.

Element Block: a collection of elements lying on a single processor and satisfying two specific criteria: (a) all elements in the block have the same number of associated nodes, and (b) all associated nodes have the same pattern of solution unknowns.

Element Block Active Node List: the set of all nodes on a given processor associated with a given block of elements -- this particular active node list plays an important role in returning the computed solution to the finite-element program.

Element Matrix: the submatrices that result when the underlying energy functionals are integrated over a given element. In general, these matrices include a square element stiffness matrix (with order given by the sum of all the solution cardinalities over each node associated with the given element, plus the sum of any elemental solution parameters that may be present), an element load vector (with the same number of rows as the element stiffness), and in some problems (most notably, eigensolution applications), an element mass matrix of the same size as the element stiffness.

ElemSet: an aggregation of elements into a conveniently-sized collection.

External Nodes: nodes which are either: (a) are involved in local calculations (e.g., appear in local constraint relations) but are not found in the active node list, or (b) are in the local active node list and are involved in another processor's calculations, but are not in that processor's active node list

Field Solution Cardinality: the number of scalar solution components associated with a given mathematical field constructed via finite-element approximation. In a multiphysics setting, several interacting fields may be present within the computational simulation.

Global Matrix: synonymous with *System Matrix.*

Lagrange Multiplier Constraint: a method for appending algebraic constraints to a set of finite-element equations. This approach adds one new equation for each constraint, and generally produces an indefinite system of equations.

Load Vector: the right-hand-side vector obtained by assembling all the element load vectors during the element assembly process. In addition to element terms, the load vector may contain other entries associated with boundary conditions or with discrete constraint equations. In general, the term "load vector" may apply to either the assembled system-level right-hand-side or the individual elemental contributions, but the latter vectors are generally prefaced with the modifier "element" to indicate their smaller unassembled form.

Matrix: a rectangular two-dimensional array of numbers, where the length of each row is a constant. In short, the term *matrix* here is taken in its usual mathematical interpretation. See *table* for more general two-dimensional data representations.

Mesh: the aggregation of nodes and elements that tile the solution domain, and that collectively define the underlying finite-element interpolant used to approximation the solution of the physical problem being modeled. A finite-element mesh generally includes (at a minimum) a set of nodes, a collection of elements that tiles the material domain, and a set of connectivity data that associate each node with one or more elements (this latter data structure is also termed the mesh topology, or the connectivity array).

Nodal Solution Cardinality: the total number of solution parameters defined at a node. At a block boundary in a multiphysics simulation, different blocks may have different views of the nodal solution cardinality, so care should be taken to determine the context in which this term is used, including whether it applies to the entire analysis (where it would represent the sum of all the field cardinalities over each field found at a node) or restricted to a single block of elements (where that sum would be restricted only to the set of fields approximated over elements of that block type).

Node: an interpolation point used to construct the finite-element interpolant utilized to approximate the solution of a given boundary-value-problem of computational physics.

NodeSet: a collection of nodal data records for nodes where specific data must be identified (as opposed to nodes that can be completely specified by the data in the mesh topology array). Examples include groups of nodes with identical boundary conditions, sets of shared or external nodes, etc..

Penalty Constraint: a method for appending algebraic constraints to a set of finite-element equations. This approach adds no new equations to the set, and will preserve a positive-definite character of the finite-element equations (assuming this characteristic existed beforehand), but penalty constraints may cause the resulting system to become poorly-conditioned, and thus difficult to solve.

Solution Cardinality: either the nodal solution cardinality or the field solution cardinality, depending upon the implicit context.

Stiffness Matrix: the square sparse coefficient matrix resulting from the collection of all the discrete elemental contributions to the underlying energy functional.

System Matrix: a general term referring to a matrix that corresponds to the entire problem domain (such as the system stiffness matrix), as opposed to any matrix or vector associated with only part of the domain (such as an element stiffness matrix).

Table: an two-dimensional array of numbers that generalizes the notion of matrix to the case where all rows are not required to be of the same length. A table may be rectangular (i.e., representable by a matrix) or it may have a "ragged" right edge because each row has its own particular length.

UNLIMITED RELEASE

INITIAL DISTRIBUTION:

1	MS 9003	D. R. Henson, 8900/8980
1	MS 9003	D. L. Lindner, 8902
1	MS 9011	P. W. Dean, 8903
1	MS 9011	B. V. Hess, 8910
1	MS 1202	M. R. Fox, 8920
1	MS 9012	S. C. Gray, 8930
1	MS 9037	J. C. Berry, 8930-1
1	MS 9019	B. A. Maxwell, 8940
1	MS 9011	J. C. Meza, 8950
1	MS 9214	C. H. Tong, 8950
1	MS 9011	M. H. Rogers, 8960
1	MS 9011	J. A. Larson, 8970
1	MS 9214	B. A. Allan, 8980
1	MS 9214	R. C. Armstrong, 8980
5	MS 9214	R. L. Clay, 8980
1	MS 9214	K. J. Perano, 8980
5	MS 9214	A. B. Williams, 8980
1	MS 9214	P. S. Wyckoff, 8980
1	MS 9012	K. R. Hughes, 8990
1	MS 9001	M. E. John, 8000
		Attn: R. C. Wayne, 2200/8400
		M. E. John, 8100
		L. A. West, 8200
		W. J. McLean, 8300
		P. N. Smith, 8500
		P. E. Brewer, 8600
		T. M. Dyer, 8700
		L. A. Hiles, 8800
		D. L. Crawford, 9900
3	MS 9018	Central Technical Files, 8940-2
1	MS 0899	Technical Library, 4916
1	MS 9021	Technical Communications Dept. 8815/Technical Library, 4916
1	MS 9021	Technical Communications Dept. 8815 for DOE/OSTI
1	MS 0806	J. A. Schutt, 4616
1	MS 9405	P. E. Nielan, 8743
3	MS 0437	L. M. Taylor, 9118
1	MS 0836	J. H. Biffle, 9121
1	MS 1111	S. A. Hutchinson, 9221
1	MS 1111	J. N. Shadid, 9221
1	MS 1110	R. S. Tuminaro, 9222
1	MS 0819	J. S. Peery, 9231

Kyran D. Mish (3)
Lawrence Livermore National Laboratory
Livermore, CA 94550

J. Robert Neely
Mailstop L035
Lawrence Livermore National Laboratory
Livermore, CA 94550

Ivan J. Otero
Mailstop L035
Lawrence Livermore National Laboratory
Livermore, CA 94550

Barry Smith
Mathematics and Computer Science Division
Argonne National Laboratory
9700 S Cass Ave
Argonne, IL 60439

This page intentionally left blank