SANDIA REPORT

SAND2004-6430

Unlimited Release

Printed February 2005

Finite Element Interface to Linear Solvers (FEI) Version 2.9: Guide and Reference Manual

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's National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

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

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: <http://www.osti.gov/bridge>

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: <http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online>

Finite Element Interface to Linear Solvers (FEI) Version 2.9: Users Guide and Reference Manual

Alan B. Williams
Advanced Computational Mechanics Architectures Dept.
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-0826

Abstract

The Finite Element Interface to Linear Solvers (FEI) is a linear system assembly library. Sparse systems of linear equations arise in many computational engineering applications, and the solution of linear systems is often the most computationally intensive portion of the application. Depending on the complexity of problems addressed by the application, there may be no single solver package capable of solving all of the linear systems that arise. This motivates the need to switch an application from one solver library to another, depending on the problem being solved. The interfaces provided by various solver libraries for data assembly and problem solution differ greatly, making it difficult to switch an application code from one library to another. The amount of library-specific code in an application can be greatly reduced by having an abstraction layer that puts a "common face" on various solver libraries.

The FEI has seen significant use by finite element applications at Sandia National Laboratories and Lawrence Livermore National Laboratory. The original FEI offered several advantages over using linear algebra libraries directly, but also imposed significant limitations and disadvantages. A new set of interfaces has been added with the goal of removing the limitations of the original FEI while maintaining and extending its strengths.

Acknowledgement

FEI design, development and testing has benefitted greatly from the involvement of many people over the years, including original developers who have long since moved on, as well as people at other laboratories. A partial list of people who deserve mention include: Robert Clay, Kim Mish, Ivan Otero, Lee Taylor, Carter Edwards, Jeff Keasler, Dave Stevens, Brad Wallin, Colin Aro, Mark Adams and Kendall Pierson.

The format of this report is based on information found in [10] and the example material at [2].

Contents

1	Introduction	7
1.1	Organization of Document	8
2	Linear System Assembly	8
3	“Old” versus “New” Interfaces	10
4	Original FEI Interface	11
4.1	FEI Strengths and Weaknesses	12
5	The “New” Interfaces	13
6	fei::VectorSpace	17
6.1	Shared Identifiers	18
6.2	Inactive Degrees of Freedom	18
7	fei::MatrixGraph	18
7.1	Connectivity Blocks, Connectivity Lists	19
7.2	Patterns	19
7.3	Constraint Relations	19
7.4	Slave Constraints	21
8	fei::Vector	22
8.1	Data Input	23
8.2	Data Access	23
9	fei::Matrix	23
10	fei::LinearSystem	23
10.1	Essential Boundary Conditions	24
10.2	Constraint Coefficient Loading	24
11	fei::Solver	25
12	fei::Factory	25
	References	26

Appendix

	Original FEI interface	27
	fei::VectorSpace Class Reference	29
	fei::MatrixGraph Class Reference	41
	fei::Matrix Class Reference	49
	fei::Vector Class Reference	53
	fei::LinearSystem Class Reference	55
	fei::Solver Class Reference	57
	fei::Factory Class Reference	59
	fei::VectorSpace::Factory Class Reference	60
	fei::MatrixGraph::Factory Class Reference	60
	fei::Matrix::Factory Class Reference	61
	fei::Vector::Factory Class Reference	61
	fei::LinearSystem::Factory Class Reference	62
	fei::Solver::Factory Class Reference	63

Figures

1	Trivial 2-element mesh showing numbered nodes at which a scalar field 'T' is defined. The resulting linear system in this case has 6 equations, with matrix nonzero locations indicated by shaded blocks.	9
2	Trivial 2-element mesh on 2 processors.	10
3	Interfaces in the fei:: namespace.	13
4	Simple mesh with 1 element refined, illustrating a hanging node constraint.	20
5	Trivial 2-element mesh with essential boundary conditions imposed for nodes 0 and 1.	24

Finite Element Interface to Linear Solvers (FEI) Version 2.9: Users Guide and Reference Manual

1 Introduction

Consider the assembly and solution of a linear system denoted by

$$Ax = b, \quad A \in \Re^{N \times N}, \quad x, b \in \Re^N \quad (1)$$

where N is the number of degrees of freedom in the problem being solved. In the context of finite element formulations the linear system is often denoted by $Ku = f$, with K being the global “stiffness” or “system” matrix, f the “load vector” or “forcing term”, and u the solution being sought.

The assembly of a linear system by a computational engineering application involves instantiating data structures (e.g., matrix and vector objects if using a C++ solver library) and populating them with coefficient data. The solution of the linear system involves passing the populated data structures to a solution algorithm and then recovering the solution. There are many software libraries available for solving linear systems, including Trilinos [7, 8], PETSc [4], HYPRE [5], FETI [9, 6] and Prometheus [1, 3] (to name only a few). Some libraries provide a selection of solution and preconditioning methods, while others provide a single specialized iterative algorithm or direct solver. Choosing the appropriate library to use in an application is often non-trivial and depends on several factors including the formulation chosen by the application, and the mathematical properties of the linear system. It is often desirable for an application to be able to choose a solver library at run-time, depending on the particular problem being solved.

While the tasks of assembling and solving a linear system are conceptually always the same for a given application, the implementation can vary greatly depending on which solver library is being used. Since the data structures being populated are almost always provided by the solver library, a significant amount of application code may need to be library-specific. In order to allow the application to switch easily between different solver libraries, the library-specific code that deals directly with solver data structures needs to be moved from the application into an abstraction layer. The abstraction layer makes use of the fact that solver libraries share many conceptual details related to making contributions to matrices and vectors, launching solution algorithms, etc. Thus the abstraction layer defines generic interfaces for the application to use, and library-specific implementations of those generic interfaces translate the data into the specific form required by the library. An application can then switch between various implementations of the generic interface without altering the application’s calling code.

The Finite Element Interface to Linear Solvers (FEI) serves as an abstraction layer for assembling linear systems. As the name implies, the FEI was originally intended for finite element applications. It consisted of a single large interface that accepted linear system contributions, handled the launching of an underlying solution method, and finally returned the solution data. This approach was found to be restrictive and overly opaque (hid too much, restricted access to underlying solver library and data structures). Rather than having a single interface that represents the entire underlying library and manages linear system assembly and solution, it is more flexible and natural to have several smaller, more modular interfaces which represent individual objects such as matrices, vectors,

vector-spaces, etc. In addition to the improved modularity, they allow more general use-cases (no longer focusing exclusively on nodal finite element formulations) and remove restrictions on access to underlying data and objects.

1.1 Organization of Document

This document will first describe general issues and terms associated with linear system assembly, then briefly describe the original FEI before describing the new modular interfaces. Each of the new interfaces is described in conceptual terms, and the details of method prototypes and arguments is left to the sections in the appendices, which are produced from in-code documentation. For examples demonstrating usage of the interfaces, see the programs `poisson3.C` and `cube3.C` in the FEI code distribution.

2 Linear System Assembly

The FEI is intended for applications which perform calculations on unstructured meshes and solve linear systems with sparse matrices. In this document a mesh is described as a collection of mesh-objects. There are several different kinds of mesh-objects, including nodes, edges, faces and elements. Note that there are other names for some of these, such as vertices, cells, etc. Additionally, in some contexts an entity such as a constraint-relation can be used as a mesh-object for the purposes of defining and placing degrees of freedom in the problem.

The term “degree of freedom” (DOF) denotes a component of a field which is defined at one or more mesh-objects. Examples of fields include temperature scalars, displacement vectors, etc. Some finite-element analyses are concerned only with a temperature field defined at the nodes of a mesh, in which case the linear system to be assembled has dimension equal to the number of nodes. Other analyses can be far more complex, defining a mixture of nodal displacement fields, edge pressures, flux on faces, etc.

To specify the location of a particular DOF in the problem, it is necessary to pair a field and a mesh-object; e.g., temperature at node 983. For vector fields, it is further necessary to specify an offset into the field. The linear system assembled from the fields defined over a mesh is a system of equations where each equation corresponds to a degree of freedom in the problem.

Using the FEI, an application assembles a linear system by providing information about the problem’s mesh and the fields of interest. Only mesh-objects on which solution-fields are defined need to be referenced. The process of describing the mesh and associated data consists of two major phases: the initialization phase, and the loading phase. In the initialization phase, the application defines its fields, topological and structural information such as the number and connectivity of mesh-objects (e.g., number of elements, number of nodes per element, and element-to-node connectivity lists for a nodal finite-element application). In the loading phase, the application provides coefficient data through submatrix and subvector contributions (element-stiffnesses, element-loads), boundary-condition specifications, etc.

The FEI can be thought of as a filter which accepts a variety of different types of contributions

Figure 1. Trivial 2-element mesh showing numbered nodes at which a scalar field 'T' is defined. The resulting linear system in this case has 6 equations, with matrix nonzero locations indicated by shaded blocks.

in forms that are naturally produced by computational engineering applications, and translates them into algebraic linear system contributions. It assists the assembly process by defining vector-spaces and matrix-graphs, assists with loading contributions of coefficient data, and with the imposition of boundary-conditions and constraint-relations. The FEI provides infrastructure services such as managing inter-process communication to ensure that contributions from shared mesh-objects are given to the linear system on the correct processor. The FEI also provides mappings from mesh-objects and fields on the application side to equations or global indices in the linear system.

Consider the trivial 2-element mesh of 2-D quadrilateral elements shown in figures 1 and 2. The mesh has a scalar field “T” defined at the nodes. The linear system having the structure shown would arise from a finite-element formulation that produces, for each element, an element-stiffness submatrix and an element-load subvector. The dimensions of the submatrices and subvectors is equal to the number of DOFs defined at mesh-objects connected to an element. In this case, that is the number of DOFs defined at connected nodes. Thus, the stiffness submatrices are of size 4x4 and the load subvectors have length 4.

When a mesh is decomposed to reside on multiple processors, the placement of the processor boundary generally results in some mesh-objects being shared by more than one processor. The shared mesh-objects are duplicated so that a copy resides on each sharing processor. Note that the mesh decomposition is performed either by the application or by a preprocessing step performed before the application is launched. Applications pass all data to the FEI locally on each processor, so the FEI inherits the mesh decomposition and makes no decisions regarding the distribution of mesh-objects. In the example, nodes 2 and 3 are shared by both processors. The algebraic linear system that is assembled by the FEI doesn’t have shared equations. Instead, each equation is assigned a unique “owning” processor. The FEI uses the arbitrary rule that the lowest-numbered sharing processor is given ownership of the shared mesh-object, and that processor also owns all equations

Figure 2. Trivial 2-element mesh on 2 processors.

in the linear system that correspond to DOFs on the shared mesh-object. It is also worth noting that matrices assembled by the FEI have a one-dimensional row-wise decomposition. In other words, processors own blocks of complete rows of the matrix.

3 “Old” versus “New” Interfaces

There are currently two distinctly different ways to assemble and solve a linear system using the FEI; through the “old” original FEI interface, or through the “new” interfaces. As will be described in more detail in the next section, the original FEI interface is a single large object which manages the entire process of linear system assembly, solution, and data return. The new interfaces are a group of several smaller more modular interfaces which play separate roles but are used together to essentially perform the same overall tasks as the original FEI. It should be noted that there are C language wrappers that allow the original FEI interface to be used from non-C++ applications. The new interfaces provide no inter-language support and can only be used from C++. The new interfaces represent an improvement over the old, offering increased modularity and flexibility as well as improved performance due to better implementation practices. The old interface will continue to be maintained since it is not trivial for applications to immediately switch to the new interfaces. However, it is hoped that the new interfaces will represent an attractive alternative.

The FEI library is written in the C++ programming language. FEI development began in 1997, and at that time C++ language support wasn’t entirely uniform from one compiler to another. Support for language features like templates, namespaces and the Standard Template Library has steadily improved in recent years, and as a result there are a couple of inconsistencies in FEI implementation code that are worth noting. The original FEI interface was expressed as an abstract base class called “FEI” and implemented by a class called “FEI.Implementation”, neither of which are in a namespace. The new interfaces are declared as abstract base classes in a namespace called

“fei::” and implemented by corresponding classes in a different namespace (“snl_fei::”). Some of the implementation code for the new fei classes use STL containers, but this can still be disabled by a compile-time macro if the code is to be used on a platform which has poor STL support. Some of the fei implementation classes are templates, and by default the headers in turn include the corresponding ‘.C’ files. This can also be controlled using a compile-time macro.

4 Original FEI Interface

The original FEI interface contained methods in 5 broad groups, as follows.

1. Structure Initialization: Methods for defining field sizes (number of scalar components), number of elements, nodes-per-element, passing connectivity-lists, etc.
2. Coefficient Data Loading: Methods for loading element-stiffness submatrices, element-load subvectors, boundary-conditions, etc.
3. Solution and Residual Calculation: Methods for launching underlying solver-library’s solution methods, residual calculations.
4. Solution Data Return: Methods for returning nodal solution data from underlying ‘x’ vector, etc.
5. Miscellaneous Queries for Attributes: Methods for querying field sizes, number of elements in an element-block, etc.

The names of all of the methods are given in an appendix. Over time the number of methods in the interface became so large as to be a significant motivating factor for developing the group of smaller interfaces described in later sections.

It is useful to illustrate basic usage of the interface with the following pseudo-code that assembles and solves a very simple problem. Note that for clarity and brevity the pseudo-code will omit most function arguments, and will not check function return-codes for errors, etc.

```
//Declare an instance of the FEI class.
FEI* fei = new FEI_Implementation(...); //arguments not shown

//define fields. e.g., fieldID 0 for displacement, with field-
//size equal to 3.
fei->initFields(...);

//define the characteristics of a block of elements. e.g., specify
//that there will be 8 nodes per element and each node will have
//field 0 defined.
fei->initElemBlock(...);

//loop over elements and pass element-node connectivity lists to
```

```

//the FEI instance.
for(i=0; i<nelem; ++i) {
    fei->initElem(...);
}

//advise FEI that initialization is complete.
fei->initComplete();

//loop over elements again and pass stiffness submatrices and
//load subvectors to the FEI instance.
for(i=0; i<nelem; ++i) {
    fei->sumInElemMatrix(...);
    fei->sumInElemRHS(...);
}

//pass boundary-condition specifications to the FEI instance.
fei->loadNodeBCs(...);

//advise FEI that coefficient loading is complete.
fei->loadComplete();

//launch the underlying linear solver method to solve the
//linear system.
fei->solve(...);

//obtain the solution corresponding to lists of node ids etc.
fei->getNodalFieldSolution(...);

```

This pseudo-code provides the briefest possible example to show how the original FEI interface is used to assemble and solve a linear system. For more complete and realistic examples, see the programs included with the FEI source-code distribution. Specifically, see the programs in the files `poisson.C` and `cube.C`.

4.1 FEI Strengths and Weaknesses

The original FEI interface provided a very strong abstraction layer insulating application code from the details of using a linear algebra library. For some nodal finite element formulations the interface is ideal and natural to use. Unfortunately these characteristics are also weaknesses. For many problems it is necessary to make use of special solver library features in order to use optimal solution methods. An overly opaque abstraction layer makes it very difficult to interact with particular library features in some cases. Also, the strong focus on nodal finite element formulations makes it difficult to consider other formulations, fields defined on other types of mesh-objects (other than nodes), etc.

5 The “New” Interfaces

The new interfaces are intended to correct several deficiencies of the original FEI interface. Instead of having a single interface that manages the entire process of linear system assembly and solution, there are now separate interfaces representing vector-spaces, matrix-graphs, matrices, vectors, linear systems and solvers. This allows finer and more flexible control of system assembly. Additionally, input methods have been generalized to allow the definition and use of arbitrary mesh-object types, rather than having terms like “node” hard-wired into the interface. This removes the focus on nodal finite element formulations, allowing for easy use of the interfaces for any formulation. Lastly, the abstraction layer has been made thinner and more light-weight. It still provides insulation from library-specific interfaces, but it is now easier to by-pass the abstraction layer when appropriate and gain access to the underlying library-specific objects. The new interfaces are intended to be used as “helpers” to assist with creating and using library-specific objects, but it is recognized that there are also situations when the application code may need to deal directly with those objects. If done intelligently and confined to certain “library-aware” code scopes, it can greatly enhance functionality by allowing access to library-specific features without defeating the abstraction layer and without limiting the application’s ability to switch from one library to another.

The fei interfaces are divided into two sub-groups, providing abstraction and infrastructure. Abstraction refers to the provision of a layer that allows an application code to work with a “generic” matrix or vector object without necessarily knowing the details of the data structure or which library provides it. Infrastructure refers to services provided by fei classes which assist with the initialization or usage of matrices, vectors, etc. An example of fei infrastructure is the `fei::VectorSpace`, which maps a set of degrees of freedom to an algebraic equation space, which is in turn needed in order to define sizes and data distribution when creating vectors, etc.

Figure 3. Interfaces in the `fei::` namespace.

There are 6 interfaces, separately representing the following entities:

- **fei::VectorSpace** describes the problem. It holds definitions of mesh-object types, fields, lists of mesh-objects on which fields are defined, lists of mesh-objects which are shared with other processors, etc. It ultimately maps sets of degrees of freedom to a globally consistent algebraic equation space.
- **fei::MatrixGraph** represents the connections between two vector-spaces. It accumulates connectivity lists (e.g., element-node lists, constraint-relation connectivities) and generates an algebraic matrix graph.
- **fei::Vector** is a thin container that provides an abstraction layer for library-specific vector objects. It provides methods for passing coefficient data to and from vectors. It also provides methods for communicating data between processors that own the corresponding mesh-objects, and processors that only share the mesh-objects. This allows users to access shared data on the local processor even if the data is owned by a different processor in the underlying library’s distributed vector.
- **fei::Matrix** is a thin container that provides an abstraction layer for library-specific matrix objects. It provides methods for inputting and accessing coefficient data, and also methods for communicating shared data between processors.
- **fei::LinearSystem** is a container that binds a matrix and two vectors (solution and right-hand-side) for the purposes of essential boundary condition enforcement, etc.
- **fei::Solver** is a thin container that provides an abstraction layer for library-specific solver objects or interfaces. Implementations of this convert “generic” fei:: matrices and vectors to the library-specific inputs required by particular solution methods.

These interfaces, like the original FEI interface, are abstract. They provide many of the same services provided by the original FEI interface in terms of accepting data that is naturally produced by the application (element submatrices, etc.) and translating the data into algebraic contributions for the underlying library-specific data objects.

There is also a factory interface for generating instances of these interfaces in code-scopes that aren’t aware of the run-time type of the objects. As will be described in more detail in following sections, Matrix, Vector and Solver instances will generally have run-time types specific to the particular linear algebra library being used, while VectorSpace, MatrixGraph and LinearSystem instances will not. Generally the linear algebra library being used provides some form of vector, matrix and solver, with interfaces and structures that are unique to the library. These are abstracted by the corresponding fei interfaces mentioned above. The other fei classes (VectorSpace, MatrixGraph and LinearSystem) provide infrastructure more than abstraction, since they often don’t have direct counterparts in linear algebra libraries.

A library-specific instance of the fei::Factory interface is generally created first. That can then be used in other code-scopes to generate instances of the other interfaces, ensuring that they have consistent run-time types without requiring that the application code be aware of what the run-time types are. The assumption here is that the application must create the factory in some code-scope that is aware of the particular linear algebra library being used. However, those portions of the application that manage the majority of linear system assembly can use the factory to create objects such as matrices and vectors without knowing which library is being used.

Some of the interfaces depend on the others, and this dictates the order in which they must be created and initialized in some cases. For instance, a vector-space is required as an input argument when creating a vector. A matrix-graph is required before creating a matrix, etc.

The following pseudo-code provides a rough outline of creating and using these interfaces to assemble and solve a linear system. (Note again that the pseudo-code omits many arguments and does not check error return codes, etc.)

```
//Assume an instance of fei::Factory has been created.
fei::Factory* factory = ...;

//Create vector-space and matrix-graph.
fei::VectorSpace* vecspace;
factory->createVectorSpace(MPI_COMM_WORLD, vecspace);

fei::MatrixGraph* matgraph;
factory->createMatrixGraph(vecspace, ... , matgraph);

//define a displacement field
int displ_field = 0, displ_field_size = 3;
vecspace->defineFields(1, &displ_field, &displ_field_size);

//define two mesh-object types
int nodeType = 0, edgeType = 1;
vecspace->defineIDTypes(1, &nodeType);
vecspace->defineIDTypes(1, &edgeType);

//define the characteristics of a block of elements. e.g., specify
//that there will be 8 nodes per element and each node will have
//the displacement field defined. (Argument declarations not shown.)
matgraph->definePattern(patternID, nodesPerElem, nodeType, displ_field);

//specify how many elements will be in this block of elements.
matgraph->initConnectivityBlock(blockID, nelelem, patternID);

//loop over elements and pass element-node connectivity lists to
//the matrix-graph.
for(i=0; i<nelem; ++i) {
    matgraph->initConnectivity(blockID, elems[i], nodelist[i]);
}

//advise the matrix-graph that initialization is complete.
matgraph->initComplete();

//create matrix, vectors, bind them into a linear-system
fei::Matrix* A;
fei::Vector *x, *b;
fei::LinearSystem* linearsystem;
```

```

factory->createMatrix(matgraph, A);
factory->createVector(matgraph, x);
factory->createVector(matgraph, b);
factory->createLinearSystem(matgraph, linearsystem);

linearsystem->setMatrix(A);
linearsystem->setSolutionVector(x);
linearsystem->setRHS(b);

//loop over elements again and pass stiffness submatrices and
//load subvectors to the matrix and rhs vector.
for(i=0; i<nelem; ++i) {
    A->sumIn(blockID, elems[i], ...);
    b->sumInElemRHS(blockID, elems[i], ...);
}

//pass boundary-condition specifications to the linear-system.
linearsystem->loadEssentialBCs(numBCnodes, bcnodeIDs, ...);

//advise linear-system that loading is complete (BCs can be imposed, etc.).
linearsystem->loadComplete();

//create an instance of fei::Solver and solve the linear-system
fei::Solver* solver;

factory->createSolver(solver);

solver->solve(linearsystem, ...);

//obtain the solution corresponding to lists of node ids etc.
x->copyOutFieldData(...);

```

As illustrated by the pseudo-code, the following sequence of primary steps is required to assemble a linear system.

- initialize basic problem features such as mesh-object types (identifier types), and field identifiers
- initialize relevant mesh connectivities which will induce the matrix graph (nonzero sparsity structure)
- pass in coefficient data (submatrices, subvectors, etc.)

There are steps not shown in the above pseudo-code which usually are required for real problems, including identification of shared mesh-objects for parallel computations, definition of constraint-relations, etc. The following sections provide more detailed descriptions of the individual interfaces,

along with usage guidelines and expected behavior, as well as arguments that are required when creating interface instances.

6 fei::VectorSpace

fei::VectorSpace provides methods for defining a solution-space (set of degrees of freedom) and for mapping that space to a globally consistent algebraic equation space. fei::VectorSpace seldom has a direct counterpart in linear algebra libraries; it provides infrastructure rather than abstraction. Factory classes use the algebraic vector-space information provided by fei::VectorSpace (such as global and local dimensions, etc.) in constructing library-specific vector objects.

Once initialized, fei::VectorSpace holds a description of all degrees of freedom defined for the problem and has a large number of query methods which can provide information such as the following:

- number of locally held mesh-objects
- lists of locally held mesh-object identifiers
- number of fields defined across the entire problem, and a list of those fields
- list of fields defined at a particular identifier
- global indices for equations corresponding to a particular identifier
- whether a particular identifier is locally owned
- number of scalar components in a particular field

For a complete list of queries available, refer to the class documentation, which is provided in an appendix (see table of contents).

fei::VectorSpace provides methods for defining mesh-object types (referred to as identifier-types or ID-types) and fields that will be active in the problem. See the methods **defineFields** and **defineIDTypes** in the fei::VectorSpace class documentation. Also, methods are provided for initializing information such as which fields will be active at specific mesh-objects (see **initSolutionEntries**). fei::VectorSpace accumulates lists of mesh-objects which have fields defined, and ultimately generates the information necessary to define an algebraic vector-space, including the total number of degrees of freedom in the problem and lists of indices that are local to each processor. fei::VectorSpace also provides methods for declaring which mesh-objects are shared by multiple processors (see **initSharedIDs**). The generated algebraic vector-space essentially consists of the set of global indices or equation numbers that arise from the mapped degrees of freedom. This information can be used in defining or constructing vector objects for almost any linear algebra library. VectorSpace can generate indices in “overlapping” or “non-overlapping” form. These terms are mainly relevant to parallel computing. An overlapping set of indices includes, on each processor, indices which correspond to mesh-objects that are either owned or shared. A non-overlapping set of indices includes only indices from owned mesh-objects. Thus, some of the indices in the overlapping set may be duplicated on multiple processors while indices in the non-overlapping set are unique in that they only appear on one processor. See the methods **getIndices_Owned**, **getIndices_SharedAndOwned**, etc.

6.1 Shared Identifiers

When a problem is run in a distributed-memory parallel setting, some mesh-objects are typically shared by multiple processors. It is necessary for the user to tell the `fei::VectorSpace` object which mesh-objects are shared and which processors share them, by passing lists of mesh-object IDs and lists of sharing processors to the `fei::VectorSpace::initSharedIDs` method. Furthermore, it is necessary to do this in a globally symmetric way. This means that if node 48 is shared by processors 2 and 3, for example, then the `initSharedIDs` method must be called on processor 2 and also on processor 3. Processor 2 must be told that it shares the node with processor 3, and processor 3 must be told that it shares the node with processor 2.

6.2 Inactive Degrees of Freedom

A later section details some `fei::MatrixGraph` functionality related to slave constraints and the projection of certain degrees of freedom out of the solution space. This corresponds to marking those degrees of freedom as inactive on a `fei::VectorSpace` object. The method used for this is **`markInactiveDOF`**. Note that the removal of inactive degrees of freedom is transparent to the user for the most part. The user simply initializes the slave constraints, and the `fei::MatrixGraph` internally interacts with the `fei::VectorSpace` as necessary to deactivate the appropriate degrees of freedom.

7 `fei::MatrixGraph`

`fei::MatrixGraph` accumulates connectivities and generates an algebraic matrix graph, defining the nonzero structure of the sparse matrix that is to be created. Like `fei::VectorSpace`, `fei::MatrixGraph` provides infrastructure rather than abstraction. A `fei::MatrixGraph` object requires two `fei::VectorSpace` objects at creation, which are referred to as a row-space and a column-space. If the column-space object provided at creation is `NULL`, the `fei::MatrixGraph` simply assumes that the column-space equals the row-space. The input `fei::VectorSpace` objects don't need to be fully initialized when the `fei::MatrixGraph` is created. If the user passes connectivities to the `fei::MatrixGraph` that include mesh-objects that haven't yet been initialized in the `fei::VectorSpace`, the `fei::MatrixGraph` will call the appropriate `fei::VectorSpace` methods to initialize them.

Entries in a matrix-graph represent connections between mesh-objects in the row-space and mesh-objects in the column-space. In the general case, each matrix-graph contribution is associated with two mesh-objects, one from the row-space and one from the column-space. The location of the contribution in the algebraic matrix-graph (row and column) is determined by looking up the equation or index associated with the row-space mesh-object and the column-space mesh-object, respectively. In the general case a submatrix contribution must be accompanied by two connectivity lists, row-space mesh-objects and column-space mesh-objects. These connectivity lists, together with information such as which fields are defined at each mesh-object, are mapped to a list of indices referred to as scatter-indices. The scatter-indices for the row-space and column-space together specify the locations in the matrix-graph of the components of the contribution. In the case of symmetric contributions (such as most finite-element stiffness submatrices) the row-space is the same as the column-space, and a single list of mesh-objects is used to specify both row and column locations. A

`fei::MatrixGraph` object can be queried for its row-space and vector-space objects using the methods `getRowSpace` and `getColumnSpace`.

7.1 Connectivity Blocks, Connectivity Lists

The term “connectivity list” refers to a list of mesh-objects which are connected in some way, and which will produce entries in the matrix-graph. An example of a connectivity list is a list of nodes that are associated with an element. In many finite-element formulations, element contributions are symmetric so that the nodes in an element-to-node connectivity list are both row-space nodes and column-space nodes. Some problems can produce non-symmetric contributions, which are specified by two connectivity lists, containing a list of mesh-objects in the row-space, and another list of mesh-objects in the column-space. The term “connectivity block” refers to a homogeneous group of contributions, and an example of a connectivity block is a group of elements with the same topology.

`fei::MatrixGraph` provides methods for declaring (initializing) connectivity blocks with either symmetric or non-symmetric contributions. Corresponding methods are provided to initialize connectivities, accepting one or two connectivity-lists. See the methods `initConnectivityBlock` and `initConnectivity` in the `fei::MatrixGraph` class documentation. When initializing a connectivity-block, the user must specify a “pattern” which describes the layout of contributions. Patterns are described in the next subsection.

7.2 Patterns

Connectivities for a large block of homogeneous contributions are initialized on the matrix-graph by first describing the layout of a single contribution, in terms of connected mesh-objects and associated fields, and then supplying connectivity lists for each contribution. This layout is referred to as a “pattern”. A pattern and a connectivity list together contain the information necessary to produce a set of scatter-indices. Since a pattern only applies to a single connectivity list, two patterns are required to describe a non-symmetric contribution.

`fei::MatrixGraph` provides methods for defining patterns of varying levels of complexity. A pattern defines the number of mesh-objects that will make up a connectivity list, along with the type of those mesh-objects, and also the fields that are associated with each mesh-object. A simple pattern might describe a list of mesh-objects which are all the same type and which all have a single associated field. More complicated patterns may involve a mixture of mesh-object types, and varying numbers of fields per mesh-object. See the different overloads of `definePattern` in the `fei::MatrixGraph` class documentation.

7.3 Constraint Relations

Figure 4 shows the trivial 2-element mesh of previous examples, now with 1 element refined (replaced with 4 smaller elements). The matrix shown has 12 rows and columns, representing the 11

Figure 4. Simple mesh with 1 element refined, illustrating a hanging node constraint.

nodes in the mesh, plus an additional row and column for the lagrange multiplier constraint that ties the hanging node (6) to nodes 2 and 3.

Each constraint is defined by a list of constrained mesh-objects, the field to be constrained at each mesh object, a list of coefficient weights and a right-hand-side value. If the solution field at the i -th constrained mesh-object is denoted by u_i , and the i -th coefficient weight is denoted by w_i then the constraint enforces the relationship

$$u_0 w_0 + u_1 w_1 + \cdots + u_n w_n = rhsvalue \quad (2)$$

where n is the number of mesh-objects in the constraint.

In general, when problems are solved subject to constraint-relations, we denote the linear-system as follows. Given a system of constraint relations, denoted by

$$Cu = g, \quad C \in \mathfrak{R}^{N_c \times N}, \quad g \in \mathfrak{R}^{N_c} \quad (3)$$

where N_c is the number of constraint relations and C denotes the rectangular matrix containing the coefficient weights, and with structure determined by the constraint connectivities (lists of constrained mesh-objects). If constraints are imposed using a lagrange multiplier formulation, a logically partitioned linear system arises

$$\begin{bmatrix} K & C^T \\ C & 0 \end{bmatrix} \begin{bmatrix} u \\ \lambda \end{bmatrix} = \begin{bmatrix} f \\ g \end{bmatrix} \quad (4)$$

and notably, the matrix is indefinite which is an important consideration if choosing an iterative solution method.

To impose a constraint-relation using the lagrange multiplier formulation, a user must define a mesh-object type for constraint-relations in addition to the other “real” mesh-object types being

used. A user-specified constraint identifier is then passed, along with the identifiers and types of the mesh-objects being constrained, and the constrained field at each mesh-object, to the `fei::MatrixGraph` method **initLagrangeConstraint**. This ensures that the required structure will exist in the matrix-graph. Note that it is necessary to separately load the constraint coefficient weights, and that is done using a method on the `fei::LinearSystem` class. Constraint-relations may be imposed using a penalty formulation, by instead calling the method **initPenaltyConstraint**. Constraint-relations imposed using the penalty formulation don't cause extra rows or columns to be added to the linear system.

7.4 Slave Constraints

A third approach for imposing constraint-relations is to use slave constraints. In the example illustrated in figure 4, the degree of freedom at node 6 can be considered a “slave”, with its value defined to be a linear combination of the degrees of freedom at the other nodes in the constraint. In this particular case, the value at node 6 will simply be the average of the values at nodes 2 and 3.

One of the data-filtering services provided by the `fei` is the removal of “master-slave” constraints during linear system assembly. The matrix in equation (4) is indefinite, and can be difficult to solve or precondition effectively.

The approach used for projecting the constrained system into a reduced space is described in a paper by Saint-Georges et al [11].

When constraints represent master-slave relations (one degree of freedom is slaved to a linear-combination of other degrees of freedom), the constraint matrix C from equation (3) can be expressed as

$$C = [D \quad -I] \quad (5)$$

and D is referred to as the dependency matrix. The solution vector u can be split into dependent and independent unknowns and written as an expression involving D ,

$$u_d = Du_i + g \quad (6)$$

and the global stiffness matrix K can be partitioned according to dependent and independent variables as follows.

$$K = \begin{bmatrix} K_{ii} & K_{id} \\ K_{di} & K_{dd} \end{bmatrix} \quad (7)$$

Then a reduced matrix K_r of size $N - N_c$ (N degrees-of-freedom, N_c constraints) is given by

$$K_r = K_{ii} + K_{id}D + D^T K_{di} + D^T K_{dd}D \quad (8)$$

and if K is symmetric and positive definite, then so is K_r . A reduced right-hand-side f_r is given by

$$f_r = f_i + D^T f_d + K_{id}g + D^T K_{dd}g \quad (9)$$

and the problem of solving equation (1) subject to equation (3) is equivalent to solving the reduced system

$$K_r u_i = f_r \quad (10)$$

which is usually much easier to solve with an iterative method. In some cases applications have been able to solve the reduced system when the unreduced system couldn't be solved. The reduction can

be carried out by the fei using local operations during element-wise assembly of the linear system. It is completely transparent to the user, and the solution data is returned in the original unreduced space. See the method **initSlaveConstraint** in the class documentation for fei::MatrixGraph. When slave constraints are used, the **initComplete** method internally generates a new fei::VectorSpace object for the row-space, and that object has mappings to the reduced equation space. The reduced row-space object can be obtained from the fei::MatrixGraph using the query method **getReducedRowSpace**. If there are no slave constraints, then **getReducedRowSpace** returns the same object as **getRowSpace**.

8 fei::Vector

fei::Vector is a thin container that provides an abstraction layer for library-specific vector objects. Note that fei::Vector is primarily targeted towards data input and output, and doesn't attempt to provide a comprehensive set of mathematical operations. This vector representation does not require that data be accessed only on the 'owning' processor. In other words, this representation may be used with an overlapping data decomposition. In most cases the underlying library-specific vector will have a non-overlapping data decomposition (each equation uniquely owned by a single processor). Overlapping data (shared by local processor but the equation is owned by another processor) may be assembled into an fei::Vector locally, and will be moved into the underlying non-overlapping vector on the correct processor when the **gatherFromOverlap** method is called. Conversely, if the user wants to retrieve overlapping data from the vector locally for an equation that resides on another processor, that data is not guaranteed to be available until the **scatterToOverlap** method is called. The **scatterToOverlap** method does communication necessary to populate shared-but-not-owned data in the fei::Vector from data in the underlying algebraic vector.

Several methods are provided for putting data into, and getting data out of the vector object. See the methods **sumIn**, **copyIn**, **copyOut**, **sumInFieldData**, **copyInFieldData**, **copyOutFieldData**, etc., in the class documentation for fei::Vector. The distinction between the “sumIn” and “copyIn” methods is that “sumIn” accumulates partial sums into any data that may already be present in the specified locations, while “copyIn” replaces any data that may already be present. These input methods have multiple overloads. Some accept coefficients with locations specified by corresponding mesh-objects and fields, while others allow locations to be specified by a connectivity-list identifier (e.g., an element-identifier for a load-vector contribution). The methods that take a connectivity-list identifier only work if the fei::Vector instance had a fei::MatrixGraph object supplied at creation. If the fei::Vector was created with a fei::VectorSpace object instead of a fei::MatrixGraph, then certain input methods aren't available. Still other fei::Vector input methods allow coefficient locations to be specified using “raw” global indices or equation numbers. Global indices can be obtained by obtaining the fei::Vector's fei::VectorSpace object and using query methods to map degrees of freedom to indices. Note that if the fei::Vector object has a fei::MatrixGraph object which has slave constraints, then the reduced vector-space must be used in order to obtain correct indices. Input methods that take mesh-objects and fields internally do the lookups to put data in the correct locations, accounting for inactive entries in the vector-space.

The abstract fei::Vector class is implemented by snl_fei::Vector, which is a template. snl_fei::Vector is templated on the type of the underlying library-specific vector, and has a query method, **getUnderlyingVector**, for accessing the library-specific vector. Thus, it is easy to get the underlying vector

from `fei::Vector` using `dynamic_cast` in code scopes that know which type to cast to.

8.1 Data Input

When locally owned data is input, `fei::Vector` relays it immediately to the underlying algebraic vector. When shared remotely owned data is input, `fei::Vector` holds it in temporary storage. When **`gatherToOverlap`** is called, `fei::Vector` moves the shared data to the owning processor and then relays it to the underlying algebraic vector. At that point the temporary storage is deleted.

8.2 Data Access

When locally-owned data is accessed, `fei::Vector` retrieves it from the underlying algebraic vector directly. In order to access shared remotely owned data (overlapped data), it is necessary first to call the method **`scatterToOverlap`**. This method does the communication necessary to re-create and populate temporary storage with the shared data by retrieving that data from the underlying algebraic vector on the owning processor and sending it to the sharing processors.

9 `fei::Matrix`

`fei::Matrix` is a thin container that provides an abstraction layer for library-specific matrix objects. Like `fei::Vector`, `fei::Matrix` is implemented by a template in the `snl_fei::` namespace.

This matrix representation does not require that data be accessed only on the 'owning' processor. In other words, this representation may be used with an overlapping data decomposition in much the same way as the `fei::Vector` described in the previous section. In most cases the underlying library-specific matrix will have a non-overlapping data decomposition (each equation uniquely owned by a single processor). Overlapping data may be assembled into this abstract matrix locally, and will be funneled into the underlying non-overlapping matrix on the correct processor when the **`gatherFromOverlap`** method is called. Conversely, if the user wants to retrieve overlapping data from the matrix locally, that data is not guaranteed to be available until the **`scatterToOverlap`** method is called. For specific information on data access, see the methods **`sumIn`**, **`copyIn`**, **`sumInFieldData`**, **`copyOutRow`**, **`putScalar`**, etc., in the class documentation for `fei::Matrix`. Additionally, matrix attributes may be queried using methods such as **`getGlobalNumRows`**, **`getRowLength`**, etc.

10 `fei::LinearSystem`

`fei::LinearSystem` is a container that binds a matrix and two vectors (solution and right-hand-side) together, providing a convenient way to pass a linear system as a single argument between various code scopes. `fei::LinearSystem` also provides methods for performing operations such as essential boundary condition enforcement, and constraint-relation coefficient loading.

10.1 Essential Boundary Conditions

For specific information on loading essential boundary conditions, see the method **loadEssentialBCs** in the class documentation for `fei::LinearSystem`. The approach used for imposing essential

Figure 5. Trivial 2-element mesh with essential boundary conditions imposed for nodes 0 and 1.

boundary conditions is as follows. The corresponding rows and columns in the global matrix are zeroed and 1's are placed on the diagonal. (Before a column is zeroed, its coefficients are multiplied by the boundary condition's prescribed value and subtracted into the appropriate positions in the right-hand-side.) Then, the prescribed values are placed in the right-hand-side and the system is solved. This results in the prescribed values being placed in the solution by the solver. Naturally this can result in the boundary condition being enforced only as accurately as the tolerance which was set on the solver. If the user wishes to ensure exact boundary condition enforcement they can specify that an alternate approach is taken whereby zeros are placed in appropriate positions in the right-hand-side before the system is solved, and the prescribed values are explicitly placed in the solution vector after the solver finishes. (This is specified at run-time, by passing the string "EXPLICIT_BC_ENFORCEMENT" to `fei::LinearSystem` using the **parameters** method, and by using the method **setBCValuesOnVector** after the solve.)

10.2 Constraint Coefficient Loading

For constraint-relations, see the methods **loadLagrangeConstraint** and **loadPenaltyConstraint**. Note that for the constraint-loading methods, the user must supply a constraint-identifier that corresponds to a constraint that was already initialized on the `fei::MatrixGraph` object that is held by the `fei::LinearSystem`'s `fei::Matrix`. `fei::LinearSystem` then places the coefficients at the appropriate locations in the matrix and right-hand-side vector.

For loading penalty constraint coefficients, the following modification is made to the linear system. Refer to equation 2 for the definition of the weights. Let the matrix be denoted by A , and the right-hand-side by b . Let n be the number of constrained degrees of freedom. Note that the array “index” contains mappings from the constrained mesh-objects and fields, to the corresponding indices in the global equation space.

```
for(i=0; i<n; ++i) {
    b[index[i]] += w[i] * rhsvalue * penaltyvalue;

    for(j=0; j<n; ++j) {
        A[index[i],index[j]] += w[i] * w[j] * penaltyvalue;
    }
}
```

11 fei::Solver

fei::Solver is also an abstract interface. The primary method provided by this interface is **solve**. Implementations of this interface accept a fei::LinearSystem object as an argument to the solve method, and from that they extract “generic” fei::Matrix and fei::Vector objects, and finally obtain the library-specific underlying data objects to pass to a library-specific solution algorithm. The solve method also accepts an additional fei::Matrix argument that can be used as a preconditioning matrix. Some solver libraries allow for using a separate user-assembled matrix as the basis for calculating the preconditioner, and this handles that scenario.

12 fei::Factory

fei::Factory is an interface that provides methods for creating instances of each of the other fei:: interfaces, as well as the original FEI interface. Implementations of this interface produce instances of fei interfaces, making sure their run-time types are consistent with the particular linear algebra library being used. As previously mentioned, the interfaces fei::VectorSpace, fei::MatrixGraph and fei::LinearSystem provide infrastructure rather than abstraction and so their types don’t change with the underlying library. fei::Vector, fei::Matrix and fei::Solver, however, almost always have direct concrete counterparts in the underlying library and so these objects serve as thin wrappers containing library-specific objects.

References

- [1] Prometheus web site. http://www.cs.berkeley.edu/~madams/Prom_intro.html.
- [2] Sand report example web page with notes and example files. <http://www.cs.sandia.gov/~rolf/SANDreport/index.html>.
- [3] M. F. Adams. *Multigrid Equation Solvers for Large Scale Nonlinear Finite Element Simulations*. PhD thesis, University of California, Berkeley, 1998. Tech. Report UCB//CSD-99-1033.
- [4] S. Balay, K. Buschelman, W. Gropp, D Kaushik, M. Knepley, L. Curfman McInnes, B. Smith, and H. Zhang. Petsc 2.0 users manual. Technical report ANL-95/11 - Revision 2.1.6, Argonne National Laboratory, August 2003.
- [5] R.D. Falgout and U. M. Yang. Hypre: A library of high performance preconditioners. Technical Report UCRL-JC-146175, Lawrence Livermore National Laboratory, Livermore, California 94550, 2002.
- [6] C. Farhat and K. Pierson. The second generation of feti methods and their application to the parallel solution of large-scale linear and geometrically nonlinear structural analysis problems. *Computer Methods in Applied Mechanics and Engineering*, 184:333–374, 2000.
- [7] M. A. Heroux et al. An overview of trilinos. Technical report SAND2003-2927, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, August 2003.
- [8] M. A. Heroux and J. M. Willenbring. Trilinos users guide. Technical report SAND2003-2952, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, August 2003.
- [9] M. Lesoinne and K. Pierson. Feti-dp: An efficient, scalable and unified dual-primal feti method. *12th International Conference on Domain Decomposition Methods*, 2001.
- [10] Tamara K. Locke. Guide to preparing SAND reports. Technical report SAND98-0730, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, May 1998.
- [11] P. Saint-Georges, Y. Notay, and G. Warzee. Efficient iterative solution of constrained finite element analyses. *Comput. Methods Appl. Mech. Engrg.*, 160:101–114, 1998.

Original FEI interface

The names of the FEI methods, without their arguments, are given below to provide an overview of the functionality that the interface addresses. The FEI interface contained 5 broad groups of methods, as follows.

1. Structure Initialization: Methods for defining field sizes (number of scalar components), number of elements, nodes-per-element, passing connectivity-lists, etc.

setSolveType	parameters
initFields	initCoefAccessPattern
initElemBlock	initCoefAccess
initElem	setIDLLists
initSharedNodes	initCRMult
initSlaveVariable	initCRPen
initComplete	

2. Coefficient Data Loading: Methods for loading element-stiffness submatrices, element-load subvectors, boundary-conditions, etc.

sumInElem	loadNodeBCs
sumInElemMatrix	loadElemBCs
sumInElemRHS	loadElemTransfer
sumIntoMatrix	putIntoMatrix
sumIntoRHS	putIntoRHS
loadCRMult	setCurrentMatrix
loadCRPen	setCurrentRHS
putNodalFieldData	setMatScalars
putBlockNodeSolution	setRHSScalars
putBlockFieldNodeSolution	putSubstructureFieldSolution
putBlockElemSolution	putSubstructureFieldData
loadComplete	putCRMMultipliers

3. Solution and Residual Calculation: Methods for launching underlying solver- library's solution methods, residual calculations.

residualNorm
solve

4. Solution Data Return: Methods for returning nodal solution data from underlying x vector, etc.

getNodalFieldSolution	getBlockFieldNodeSolution
getNodalSolution	getBlockNodeSolution
getBlockElemSolution	getSubstructureFieldSolution
getCRMMultipliers	

5. Miscellaneous Queries for Attributes: Methods for querying field sizes, number of elements in an element-block, etc.

getFieldSize	iterations
getEqnNumbers	allocatedSize
getNumLocalNodes	cumulative_MPI_Wtimes
getLocalNodeIDList	version
getNumCRMultipiers	getBlockNodeIDList
getCRMultIDList	getBlockElemIDList
getFromMatrix	getFromRHS
getNumSolnParams	resetSystem
getNumElemBlocks	resetMatrix
getNumBlockActNodes	resetRHSVector
getNumBlockActEqns	resetInitialGuess
getNumNodesPerElement	deleteMultCRs
getNumEqnsPerElement	getSubstructureSize
getNumBlockElements	getSubstructureIDList
getNumBlockElemDOF	

fei::VectorSpace Class Reference

Abstract class (interface) containing the methods for defining a solution-space (a set of degrees-of-freedom) and mapping that space to a globally unique set of indices.

Example: define a displacement field over a set of node-identifiers, and map that to a set of equation-numbers.

There are multiple ways to use an instance of this interface:

For generating vectors:

1. Define fields and identifier-types
2. Initialize active fields over sets of identifiers
3. Obtain index offset and range information via one of the methods **getGlobalIndexOffsets()** (p. 35), **'getIndices_Owned()' (p. 36)**, **'getIndices_SharedAndOwned()' (p. 37)**, etc., to use in constructing or initializing a vector object.

For generating matrices:

1. Define fields and identifier-types
2. Construct an instance of **fei::MatrixGraph** (p. 41) (using this **VectorSpace** (p. 29) as a constructor argument or initialization argument) and proceed to initialize connectivities and other structural attributes on the Structure object.
3. Obtain matrix-graph information from the **fei::MatrixGraph** (p. 41) object to use in constructing or initializing a matrix object.

virtual int fei::VectorSpace::defineFields (int *numFields*, const int * *fieldIDs*, const int * *fieldSizes*) [pure virtual]

Define fields that will occur in this solution space.

Example: a temperature field might be defined as fieldID 0, size 1.

Example: a velocity field might be defined as fieldID 5, size 3.

Parameters:

numFields Input. Length of the fieldIDs and fieldSizes lists.

fieldIDs Input. List of user-supplied field-identifiers. Convention: Active solution-space fields should generally be denoted by non-negative field-identifiers, while "other" fields (such as geometric coordinates) should be denoted by negative field-identifiers.

fieldSizes Input. List of user-specified field-sizes. A field-size is the number of scalar components that make up a field.

Returns:

error-code 0 if successful

```
virtual int fei::VectorSpace::defineIDTypes (int numIDTypes, const int * idTypes) [pure virtual]
```

Define identifier-types in this solution space.

For example, define node-identifiers to be type 0, edge-identifiers to be type 1, lagrange-multiplier identifiers to be type 2, etc.

identifier-types need not be zero-based or contiguous.

Parameters:

numIDTypes Number of distinct identifier-types

idTypes User-supplied list of identifier-types

Returns:

error-code 0 if successful

```
virtual int fei::VectorSpace::getBlkIndices_Owned (int lenBlkIndices, int * globalBlkIndices, int * blkSizes, int & numBlkIndices) [pure virtual]
```

Obtain list of global block indices owned by local processor.

Only available after initComplete has been called.

Parameters:

lenBlkIndices Input. Length of user-allocated 'globalBlkIndices' list.

globalBlkIndices User-allocated list. On output, will contain all indices owned by local processor.

blkSizes User-allocated list. On output, will contain the number of scalars (point-indices) associated with each corresponding block-index.

numBlkIndices Output. Number of indices. If 'numBlkIndices' is different than 'lenBlkIndices', then globalBlkIndices will contain 'min(lenBlkIndices, numBlkIndices)' of the local processor's indices.

```
virtual int fei::VectorSpace::getBlkIndices_SharedAndOwned (int lenBlkIndices, int * globalBlkIndices, int * blkSizes, int & numBlkIndices) [pure virtual]
```

Obtain list of global block indices on local processor, including ones that are locally owned as well as shared-but-not-owned.

Only available after initComplete has been called.

Parameters:

lenBlkIndices Input. Length of user-allocated 'globalBlkIndices' list.

globalBlkIndices User-allocated list. On output, will contain all indices owned or shared by local processor.

blkSizes User-allocated list. On output, will contain the number of scalars (point-indices) associated with each corresponding block-index.

numBlkIndices Output. Number of indices. If 'numBlkIndices' is different than 'lenBlkIndices', then globalBlkIndices will contain 'min(lenBlkIndices, numBlkIndices)' of the local processor's indices.

virtual MPI_Comm fei::VectorSpace::getCommunicator () [pure virtual]

Return the MPI communicator held by this object.

When built/run in serial mode, MPI_Comm is defined to be int.

virtual int fei::VectorSpace::getFieldList (int *idType*, int *ID*, int *lenFieldIDs*, int **fieldIDs*, int & *numFields*) [pure virtual]

Given a particular identifier, request the list of fields that are associated with that identifier.

Parameters:

idType Identifier-type

ID Specified identifier

lenFieldIDs Input. Length of user-allocated 'fieldIDs' list.

fieldIDs Input. User-allocated list, length must be at least as large as the value produced by **getNumFields()** (p. ??) for this ID.

numFields Output. Number of fields. If numFields > lenFieldIDs, then fieldIDs will contain the first 'lenFieldIDs' field identifiers.

virtual int fei::VectorSpace::getFields (int *len*, int **fields*, int & *numFields*) [pure virtual]

Query for the list of fields defined for this vector-space.

Parameters:

len Input, length of the user-allocated list 'fields'.

fields Input/Output, user-allocated list, on exit contents will contain fields that are defined for this vector-space.

numFields Output, number of fields that are defined for this vector-space. If numFields is less than user-provided 'len', then only 'numFields' positions in 'fields' are referenced. If numFields is greater than user-provided len, then 'fields' is filled with the first 'len' field-ids that are defined for this vector-space.

virtual int fei::VectorSpace::getFieldSize (int *fieldID*, int & *fieldSize*) [pure virtual]

Request the field-size for a specified field-identifier.

Parameters:

fieldID Input. Specified field-identifier

fieldSize Output. If the specified field-identifier is not found, then this argument is not referenced.

Returns:

error-code 0 if successful. If the specified field-identifier is not found, then -1 is returned.

virtual int fei::VectorSpace::getGlobalBlkIndex (int *idType*, int *ID*, int & *globalBlkIndex*)
[pure virtual]

Given a particular identifier, request the corresponding global block-index.

Parameters:

idType Input. Identifier-type of the identifier being queried.

ID Input. Identifier for which a block-index is being requested.

globalBlkIndex Output. This is the global block-index of the specified identifier.

Returns:

error-code 0 if successful. If the specified degree-of-freedom is not found, -1 is returned.

virtual int fei::VectorSpace::getGlobalBlkIndexOffsets (int *lenGlobalBlkOffsets*, int * *globalBlkOffsets*) [pure virtual]

Request the global block-index offsets.

Indices are zero-based.

Parameters:

lenGlobalBlkOffsets Input. This value gives the length of the user-allocated array *globalBlkOffsets*. Should be *numPartitions*+1.

globalBlkOffsets Output. Caller-allocated array of length *lenGlobalBlkOffsets*. On exit, contains global-block-offsets.

globalBlkOffsets[*i*] is first global block-offset on processor *i*, for *i* in 0 .. *numPartitions* - 1

globalBlkOffsets[*i*+1] - *globalBlkOffsets*[*i*] is the number of block-indices on the *i*-th processor

Returns:

error-code 0 if successful

virtual int fei::VectorSpace::getGlobalBlkIndices (int *numIDs*, const int * *IDs*, int *idType*, int * *globalBlkIndices*) [pure virtual]

Given a list of IDs, fill an output-list of the global-block-indices that correspond to each ID.

Parameters:

numIDs Input. Length of the IDs list and of the globalBlkIndices list.

IDs Input. User-provided list of identifiers.

idType Input. Type of the IDs for which block-indices are being requested.

globalBlkIndices Output. User-allocated list which, on exit, will contain the requested indices. Note that the length of this list is assumed to be numIDs.

Returns:

error-code 0 if successful Note that for any IDs that are not found, the corresponding global-index will be -1.

virtual int fei::VectorSpace::getGlobalIndex (int *idType*, int *ID*, int & *globalIndex*) [pure virtual]

Given a particular degree-of-freedom, request the corresponding global index.

In this case, the degree-of-freedom is specified simply by an identifier and identifier-type, without specifying a field. This is intended to be used for requesting global indices for constraint-identifiers or other identifiers which don't have associated fields. If the specified identifier actually does have associated fields, then the output globalIndex will be the global-index corresponding to the first component of the first associated field.

Parameters:

idType Input. Identifier-type of the location at which the specified degree-of-freedom resides. Must be one of the identifier-types previously defined via a call to '**defineIDTypes()** (p. 30)'.

ID Input. Identifier for the location being specified, such as a node-identifier, etc.

globalIndex Output. This is the global index of the specified degree-of-freedom. Not referenced if the specified degree-of-freedom is not found.

Returns:

error-code 0 if successful. If the specified degree-of-freedom is not found, -1 is returned.

virtual int fei::VectorSpace::getGlobalIndex (int *idType*, int *ID*, int *fieldID*, int & *globalIndex*) [pure virtual]

Given a particular degree-of-freedom, request the corresponding global index.

A particular degree-of-freedom is specified as a component of a particular field, residing at a particular location (ID).

Parameters:

idType Input. Identifier-type of the location at which the specified degree-of-freedom resides. Must be one of the identifier-types previously defined via a call to '**defineIDTypes()** (p. 30)'.

ID Input. Identifier for the location being specified, such as a node-identifier, etc.

fieldID Input. Identifier for the field being specified.

globalIndex Output. This is the global index of the specified degree-of-freedom. Not referenced if the specified degree-of-freedom is not found.

Returns:

error-code 0 if successful. If the specified degree-of-freedom is not found, -1 is returned.

virtual int fei::VectorSpace::getGlobalIndex (int *idType*, int *ID*, int *fieldID*, int *fieldOffset*, int *whichComponentOfField*, int & *globalIndex*) [pure virtual]

Given a particular degree-of-freedom, request the corresponding global index.

A particular degree-of-freedom is specified as a component of a particular field, residing at a particular location (ID).

Parameters:

idType Input. Identifier-type of the location at which the specified degree-of-freedom resides. Must be one of the identifier-types previously defined via a call to '**defineIDTypes()** (p. 30)'.

ID Input. Identifier for the location being specified, such as a node-identifier, etc.

fieldID Input. Identifier for the field being specified.

fieldOffset Input. In case there is more than one field with the specified fieldID residing at the specified ID, this provides an offset into those fields. If only one field with specified field-ID, then this parameter is 0.

whichComp Input. Specifies a scalar component within the field. If the field only has 1 scalar component, then this parameter is 0.

globalIndex Output. This is the global index of the specified degree-of-freedom. Not referenced if the specified degree-of-freedom is not found.

Returns:

error-code 0 if successful. If the specified degree-of-freedom is not found, -1 is returned.

virtual int fei::VectorSpace::getGlobalIndexOffsets (int *lenGlobalOffsets*, int * *globalOffsets*)
[pure virtual]

Request the global index offsets.

Indices are zero-based.

Parameters:

lenGlobalOffsets Input. This value gives the length of the user-allocated array *globalOffsets*. Should be *numPartitions*+1.

globalOffsets Output. Caller-allocated array of length *lenGlobalOffsets*. On exit, contains global-offsets.

globalOffsets[*i*] is first global offset on processor *i*, for *i* in 0 .. *numPartitions* - 1

globalOffsets[*i*+1] - *globalOffsets*[*i*] is the number of indices on the *i*-th processor

Returns:

error-code 0 if successful

virtual int fei::VectorSpace::getGlobalIndices (int *numIDs*, const int * *IDs*, const int * *idTypes*, const int * *fieldIDs*, int * *globalIndices*) [pure virtual]

Given a list of IDs, fill an output-list of the global-indices that correspond to the first instance of the specified field at each ID.

Somewhat more general version of the **getGlobalIndices()** (p. 35) method above.

Parameters:

numIDs Input. Length of the IDs list.

IDs Input. User-provided list of identifiers.

idTypes Input. List of length *numIDs*, specifying the types of the IDs for which indices are being requested.

fieldIDs Input. List of length *numIDs*, specifying a field at each ID.

globalIndices Output. User-allocated list which, on exit, will contain the requested indices. Note that the length of this list is assumed to be *numIDs***getFieldSize(fieldID)*.

Returns:

error-code 0 if successful Note that for any IDs that are not found, or IDs which don't have the specified field, the corresponding global-index will be -1.

virtual int fei::VectorSpace::getGlobalIndices (int *numIDs*, const int * *IDs*, int *idType*, int *fieldID*, int * *globalIndices*) [pure virtual]

Given a list of IDs, fill an output-list of the global-indices that correspond to the first instance of the specified field at each ID.

Parameters:

numIDs Input. Length of the IDs.

IDs Input. User-provided list of identifiers.

idType Input. Type of the IDs for which indices are being requested.

fieldID Input. Specified field

globalIndices Output. User-allocated list which, on exit, will contain the requested indices. Note that the length of this list is assumed to be `numIDs*getFieldSize(fieldID)`.

Returns:

error-code 0 if successful Note that for any IDs that are not found, or IDs which don't have the specified field, the corresponding global-index will be -1.

```
virtual int fei::VectorSpace::getIDTypes (int len, int * idTypes, int & numIDTypes) [pure virtual]
```

Query for the list of identifier-types defined for this vector-space.

Parameters:

len Input, length of the user-allocated list '*idTypes*'.

idTypes Input/Output, user-allocated list, on exit contents will contain id-types that are defined for this vector-space.

numIDTypes Output, number of id-types that are defined for this vector-space. If *numIDTypes* is less than user-provided '*len*', then only '*numIDTypes*' positions in '*idTypes*' are referenced. If *numIDTypes* is greater than user-provided *len*, then '*idTypes*' is filled with the first '*len*' id-types that are defined for this vector-space.

```
virtual int fei::VectorSpace::getIndices_Owned (int lenIndices, int * globalIndices, int & numIndices) const [pure virtual]
```

Obtain list of global indices owned by local processor.

Only available after `initComplete` has been called.

Parameters:

lenIndices Input. Length of user-allocated '*globalIndices*' list.

globalIndices User-allocated list. On output, will contain all indices owned by local processor.

numIndices Output. Number of indices. If '*numIndices*' is different than '*lenIndices*', then *globalIndices* will contain '`min(lenIndices, numIndices)`' of the local processor's indices.

virtual int fei::VectorSpace::getIndices_SharedAndOwned (int *lenIndices*, int * *globalIndices*, int & *numIndices*) const [pure virtual]

Obtain list of global indices on local processor, including ones that are locally owned as well as shared-but-not-owned.

Only available after `initComplete` has been called.

Parameters:

lenIndices Input. Length of user-allocated 'globalIndices' list.

globalBlkIndices User-allocated list. On output, will contain all indices owned or shared by local processor.

numIndices Output. Number of indices. If 'numIndices' is different than 'lenIndices', then globalIndices will contain 'min(lenIndices, numIndices)' of the local processor's indices.

virtual int fei::VectorSpace::getLocalIDs (int *idtype*, int *lenList*, int * *IDs*, int & *numLocalIDs*)
[pure virtual]

Obtain a list of the local identifiers.

Note that this includes identifiers that are locally shared but not owned.

virtual int fei::VectorSpace::getNumBlkIndices_SharedAndOwned (int & *numBlkIndices*)
const [pure virtual]

Query number of block indices on local processor, including ones that are locally owned as well as shared-but-not-owned.

Only available after `initComplete` has been called.

virtual int fei::VectorSpace::getNumIndices_SharedAndOwned (int & *numIndices*) const
[pure virtual]

Query number of indices on local processor, including ones that are locally owned as well as shared-but-not-owned.

Only available after `initComplete` has been called.

virtual int fei::VectorSpace::getNumPartitions (int & *numPartitions*) [pure virtual]

Request the number of partitions.

(For MPI implementations, partitions is a synonym for processes.) The main purpose of this function is to give the user a way to calculate the length of the list that needs to be allocated before calling '**getGlobalIndexOffsets()** (p. 35)'.

Parameters:

numPartitions Output. Number of partitions or processors.

Returns:

error-code 0 if successful

virtual int fei::VectorSpace::initComplete () [pure virtual]

Indicate that initialization is complete.

This is a collective function, must be called on all processors. At this time ownership of shared IDs will be assigned, and the global index space calculated.

Returns:

error-code 0 if successful

virtual int fei::VectorSpace::initSharedIDs (int *numShared*, int *idType*, const int * *sharedIDs*, const int * *numProcsPerID*, const int * *sharingProcs*) [pure virtual]

Identify a set of identifiers as being shared with other processors.

The shared ids must be identified in a globally symmetric way. i.e., if the local processor identifies id x as being shared with processor p, then processor p MUST identify id x as being shared with the local processor.

Parameters:

numShared Input. Length of the lists *sharedIDs* and *numSharingProcsPerID*.

idType Input. The identifier-type of the ids that are being identified as shared.

sharedIDs Input. List of shared identifiers.

numProcsPerID Input. List of length *numShared*, and the i-th entry gives the number of processors being identified as sharing the i-th *sharedID*.

sharingProcs Input. Table with '*numShared*' rows, and each row is of length *numProcsPerID*. This table contains the sharing processor ranks.

Returns:

error-code 0 if successful

virtual int fei::VectorSpace::initSharedIDs (int *numShared*, int *idType*, const int * *sharedIDs*, const int * *numProcsPerID*, const int * *sharingProcs*) [pure virtual]

Identify a set of identifiers as being shared with other processors.

The shared ids must be identified in a globally symmetric way. i.e., if the local processor identifies id x as being shared with processor p, then processor p MUST identify id x as being shared with the local processor.

Parameters:

numShared Input. Length of the lists *sharedIDs* and *numSharingProcsPerID*.

idType Input. The identifier-type of the ids that are being identified as shared.

sharedIDs Input. List of shared identifiers.

numProcsPerID Input. List of length *numShared*, and the i-th entry gives the number of processors being identified as sharing the i-th sharedID.

sharingProcs Input. Packed list of length $\text{sum}(\text{numProcsPerID})$, containing the sharing processor ranks.

Returns:

error-code 0 if successful

virtual int fei::VectorSpace::initSolutionEntries (int *idType*, int *numIDs*, const int * *IDs*)
[pure virtual]

Add a set of identifiers to the solution-space.

These solution-space entries consist of identifiers that don't have associated fields.

Example: Lagrange-multiplier constraint identifiers.

This method may also be used for initializing a finite-element solution-space where the user knows that the entire problem contains only one scalar field (e.g., temperature) and so it is sufficient to define a solution space on identifiers without associating fields with those identifiers. (This will achieve a performance gain for the structure- definition, graph-generation and matrix/vector assembly.)

Parameters:

idType Input. The identifier-type over which the solution- space is being defined. Must be one of the *idTypes* defined previously via '**defineIDTypes()** (p. 30)'.

numIDs Input. Number of identifiers being added to the solution- space.

IDs Input. List of length *numIDs*. Identifiers being added to the solution-space.

Returns:

error-code 0 if successful

virtual int fei::VectorSpace::initSolutionEntries (int *fieldID*, int *numFieldInstances*, int *idType*, int *numIDs*, const int * *IDs*) [pure virtual]

Add a set of identifiers to this solution-space.

These solution-space entries consist of fields residing at identifiers.

Example: temperature field at a set of finite-element nodes.

Parameters:

fieldID Input. The field-identifier to be added. Must be one of the fieldIDs defined previously via '**defineFields()** (p. 29)'.

numFieldInstances Input. It is possible to have multiple fields of the same fieldID at each mesh location. e.g., you could have 2 pressure fields at each edge.

idType Input. The identifier-type over which the active field is being initialized. Must be one of the idTypes defined previously via '**defineIDTypes()** (p. 30)'.

numIDs Input. Length of the IDs list.

IDs Input List of identifiers over which 'fieldID' is active.

Returns:

error-code 0 if successful

virtual int fei::VectorSpace::markInactiveDOF (int *idType*, int *ID*, int *fieldID*, int *fieldOffset*, int *whichComponentOfField*) [pure virtual]

Mark a degree-of-freedom as inactive.

This function should not be called until AFTER all degrees-of-freedom have been added to this solution-space. After all inactive degrees-of-freedom have been marked inactive, **initComplete()** (p. 38) must be called to calculate the correct reduced set of global indices, etc. If global indices are requested for an inactive degree-of-freedom, they will be -1.

virtual int fei::VectorSpace::parameters (int *numParams*, const char *const * *paramStrings*) [pure virtual]

Set parameter strings.

Parameters:

numParams Number of parameters being supplied.

paramStrings List of 'numParams' strings.

The documentation for this class was generated from the following file:

- fei_VectorSpace.h

fei::MatrixGraph Class Reference

Abstract class (interface) containing the methods for defining/initializing a structure, and generating the corresponding matrix graph.

virtual int fei::MatrixGraph::createGraph (fei::SparseRowGraph *& *locallyOwnedRows*, fei::SparseRowGraph *& *remotelyOwnedRows*, bool *blockEntryGraph*) [pure virtual]

Generate a sparse row-based graph from structural data that has been accumulated.

Don't use this until after **initComplete()** (p. 44) has been called.

Parameters:

- locallyOwnedRows* Those rows of a matrix that would be owned by the local processor.
- remotelyOwnedRows* Those rows of the matrix that would be owned by remote processors, but for which the equations arise from mesh-objects that are shared (and thus contributed to) on the local processor.
- blockEntryGraph* Specifies whether the graph should be constructed on a block-entry or point-entry basis. If there is only 1 scalar DOF at each mesh-object, then a block-entry graph is the same as a point-entry graph.

virtual int fei::MatrixGraph::definePattern (int *patternID*, int *numIDs*, const int * *idTypes*, const int * *numFieldsPerID*, const int * *fieldIDs*) [pure virtual]

Define a pattern to use for subsequent blocked-contributions.

Examples include element-contributions.

This is the most general of the pattern-definition methods. This method defines a pattern consisting of a mixture of identifier-types, with each identifier having an arbitrary list of associated fields.

Parameters:

- patternID* Input. Identifier to be used later when referring to this pattern.
- numIDs* Input. number of identifiers per pattern 'instance'.
- idTypes* Input. List of length *numIDs*. Specifies the type of each identifier to be contributed for instances of this pattern. Each of the *idTypes* must be one of the *idTypes* defined for a **VectorSpace** (p. 29) that is associated with this **MatrixGraph** (p. 41). *idTypes* are defined via the method **VectorSpace::defineIDTypes()** (p. 30).
- numFieldsPerID* Input. List of length *numIDs*. *i*-th entry gives the number of fields to be associated with the *i*-th identifier in a contribution.
- fieldIDs* Input. Packed list of length *sum(numFieldsPerID[i])*. Contains the *fieldIDs* to be associated with the identifiers for a contribution.

Returns:

error-code 0 if successful

```
virtual int fei::MatrixGraph::definePattern (int patternID, int numIDs, int idType, const int *  
numFieldsPerID, const int *fieldIDs) [pure virtual]
```

Define a pattern to use for subsequent blocked-contributions.

Examples include element-contributions.

This is the 'middle' of the pattern-definition methods, in terms of the complexity of pattern that can be defined. This method defines patterns for contributions where the identifiers are all of the same type, but an arbitrary list of fields can be associated with each identifier.

Parameters:

patternID Input. Identifier to be used later when referring to this pattern.

numIDs Input. number of identifiers per pattern 'instance'.

idType Input. Specifies which type of identifiers are associated with instances of this pattern. Must be one of the *idTypes* defined for a **VectorSpace** (p. 29) that is associated with this **MatrixGraph** (p. 41). *idTypes* are defined via the method **VectorSpace::defineIDTypes()** (p. 30).

numFieldsPerID Input. List of length *numIDs*. *i*-th entry gives the number of fields to be associated with the *i*-th identifier in a contribution.

fieldIDs Input. Packed list of length $\text{sum}(\text{numFieldsPerID}[i])$. Contains the *fieldIDs* to be associated with the identifiers for a contribution.

Returns:

error-code 0 if successful

```
virtual int fei::MatrixGraph::definePattern (int patternID, int numIDs, int idType, int fieldID)  
[pure virtual]
```

Define a pattern to use for subsequent blocked-contributions.

Examples include element-contributions.

This is the simplest of the 3 pattern-definition methods that associate fields with identifiers (there is one pattern-definition method above that allows for specifying a pattern of identifiers that don't have associated fields). This method defines patterns for contributions where a single field is associated with each identifier in a list of identifiers, and all the identifiers in the list are of the same type.

Parameters:

patternID Input. Identifier to be used later when referring to this pattern.

numIDs Input. number of identifiers per pattern 'instance'.

idType Input. Specifies which type of identifiers are associated with instances of this pattern. Must be one of the *idTypes* defined for a **VectorSpace** (p. 29) that is associated with this **MatrixGraph** (p. 41). *idTypes* are defined via the method **VectorSpace::defineIDTypes()** (p. 30).

fieldID Input. field-identifier for the single field that is to reside at each identifier.

Returns:

error-code 0 if successful

virtual int fei::MatrixGraph::definePattern (int *patternID*, int *numIDs*, int *idType*) [pure virtual]

Define a pattern to use for subsequent blocked-contributions.

Examples include element-contributions.

This is the simplest of the pattern-definition methods. IMPORTANT NOTE: this method does not associate a field with the identifiers. Only use this method for problems where you explicitly don't want or need to associate fields with identifiers. Examples would include problems where only a single scalar field exists across the entire mesh and thus doesn't need to be explicitly referenced. Other cases where this might be used is for non finite-element problems that don't have identifier/field pairs.

Parameters:

patternID Input. Identifier to be used later when referring to this pattern.

numIDs Input. number of identifiers per pattern 'instance'.

idType Input. Specifies which type of identifiers are associated with instances of this pattern. Must be one of the *idTypes* defined for a **VectorSpace** (p. 29) that is associated with this **MatrixGraph** (p. 41). *idTypes* are defined via the method **VectorSpace::defineIDTypes()** (p. 30).

Returns:

error-code 0 if successful

virtual int fei::MatrixGraph::getReducedRowSpace (fei::VectorSpace *& *reducedSpace*) [pure virtual]

Obtain the **VectorSpace** (p. 29) corresponding to the reduced row-space for this **MatrixGraph** (p. 41) object.

If there are no slave-constraints, or if the **MatrixGraph** (p. 41) implementation doesn't support slave-reduction, then this will return the same solution-space as **getRowSpace()** (p. ??).

virtual int fei::MatrixGraph::initComplete () [pure virtual]

Signal the **MatrixGraph** (p. 41) object that initialization is complete.

At this point **MatrixGraph** (p. 41) implementations will perform internal synchronizations etc. This will generally be a collective method.

virtual int fei::MatrixGraph::initConnectivity (int *idType*, int *numRows*, const int * *rowIDs*, const int * *rowLengths*, const int *const * *columnIDs*) [pure virtual]

Initialize a set of arbitrary positions in the graph by providing data in a "raw" or "purely algebraic" format similar to what might be used with a standard sparse CSR (compressed sparse row) matrix.

Parameters:

idType identifier-type

numRows Number of rows, length of the following 'rowIDs' list.

rowIDs List of length 'numRows', specifying identifiers in the row-space.

rowLengths List of length numRows, giving the number of column IDs for each row ID.

columnIDs C-style table (list of lists) containing the column IDs. Number of rows is numRows, length of i-th row is rowLengths[i].

virtual int fei::MatrixGraph::initConnectivity (int *idType*, int *fieldID*, int *numRows*, const int * *rowIDs*, const int * *rowOffsets*, const int * *packedColumnIDs*) [pure virtual]

Initialize a set of arbitrary positions in the graph by providing data in a "raw" or "purely algebraic" format similar to what might be used with a standard sparse CSR (compressed sparse row) matrix.

Also specify a fieldID to be associated with these graph positions.

Parameters:

idType identifier-type

fieldID field-identifier

numRows Number of rows, length of the following 'rowIDs' list.

rowIDs List of length 'numRows', specifying identifiers in the row-space.

rowOffsets List of length numRows+1, giving offsets into the 'packedColumnIDs' list at which each row begins. i.e., the column IDs for rowIDs[i] are packedColumnIDs[rowOffsets[i]...rowOffsets[i+1]-1].

packedColumnIDs Packed list of length rowOffsets[numRows], containing the column IDs.

virtual int fei::MatrixGraph::initConnectivity (int *idType*, int *numRows*, const int * *rowIDs*, const int * *rowOffsets*, const int * *packedColumnIDs*) [pure virtual]

Initialize a set of arbitrary positions in the graph by providing data in a "raw" or "purely algebraic" format similar to what might be used with a standard sparse CSR (compressed sparse row) matrix.

Parameters:

idType identifier-type

numRows Number of rows, length of the following 'rowIDs' list.

rowIDs List of length 'numRows', specifying identifiers in the row-space.

rowOffsets List of length numRows+1, giving offsets into the 'packedColumnIDs' list at which each row begins. i.e., the column IDs for rowIDs[i] are packedColumnIDs[rowOffsets[i]...row-Offsets[i+1]-1].

packedColumnIDs Packed list of length rowOffsets[numRows], containing the column IDs.

virtual int fei::MatrixGraph::initConnectivity (int *rowPatternID*, const int * *rowConnectedIds*, int *colPatternID*, const int * *colConnectedIds*) [pure virtual]

Make a contribution to the **MatrixGraph** (p. 41)'s connectivity.

This overloading of **initConnectivity()** (p. 46) provides for structurally non-symmetric entries.

Parameters:

rowPatternID Input. Must correspond to a Pattern ID that was previously used in a call to **define-Pattern()** (p. 43).

rowConnectedIds Input. List of the identifiers that form the connectivity list for the row-space.

colPatternID Input. Must correspond to a Pattern ID that was previously used in a call to **define-Pattern()** (p. 43).

colConnectedIds Input. List of the identifiers that form the connectivity list for the column-space.

Returns:

error-code 0 if successful

virtual int fei::MatrixGraph::initConnectivity (int *blockID*, int *connectivityID*, const int * *rowConnectedIds*, const int * *colConnectedIds*) [pure virtual]

Make a contribution to the **MatrixGraph** (p. 41)'s connectivity.

This overloading of **initConnectivity()** (p. 46) provides for structurally non-symmetric entries.

Parameters:

blockID Input. Must correspond to a blockID that was previously used in a call to **initConnectivity-Block()** (p. 47).

connectivityID Input. Identifier for this connectivity list. May be an element-identifier, etc.

rowConnectedIds Input. List of the identifiers that form the connectivity list for the row-space.

colConnectedIds Input. List of the identifiers that form the connectivity list for the column-space.

Returns:

error-code 0 if successful

virtual int fei::MatrixGraph::initConnectivity (int *blockID*, int *connectivityID*, const int * *connectedIds*) [pure virtual]

Make a contribution to the **MatrixGraph** (p. 41)'s connectivity.

Examples would include element-node connectivity lists, etc.

Parameters:

blockID Input. Must correspond to a blockID that was previously used in a call to **initConnectivity-Block()** (p. 47).

connectivityID Input. Identifier for this connectivity list. May be an element-identifier, etc.

connectedIds Input. List of the identifiers that form this connectivity list.

Returns:

error-code 0 if successful

virtual int fei::MatrixGraph::initConnectivityBlock (int *blockID*, int *numConnLists*, int *rowPatternID*, int *colPatternID*) [pure virtual]

Initialize a block of connectivity contributions.

An example is a block of elements which share a common layout of nodes/fields per element.

This method accepts two pattern-ids, implying that connectivities in this block describe a non-symmetric structure. See the other overloading of this method for the symmetric case.

Parameters:

blockID Input. User-specified identifier for this block. Will generally be required to be non-negative.

numConnLists Input. Number of connectivity-lists that will be supplied for this block.

rowPatternID Input. Descriptor for the row-connectivities to be provided. Must be a pattern that was previously defined via **definePattern()** (p. 43).

colPatternID Input. Descriptor for the column-connectivities to be provided. Must be a pattern that was previously defined via **definePattern()** (p.43).

Returns:

error-code 0 if successful

virtual int fei::MatrixGraph::initConnectivityBlock (int *blockID*, int *numConnLists*, int *patternID*) [pure virtual]

Initialize a block of connectivity contributions.

An example is a block of elements which share a common layout of nodes/fields per element.

This method accepts only one pattern-id, implying that connectivities in this block describe a symmetric structure. See the other overloading of this method for the non-symmetric case.

Parameters:

blockID Input. User-specified identifier for this block. Will generally be required to be non-negative.

numConnLists Input. Number of connectivity-lists that will be supplied for this block.

patternID Input. Descriptor for the connectivities to be provided. Must be a pattern that was previously defined via **definePattern()** (p.43).

Returns:

error-code 0 if successful

virtual int fei::MatrixGraph::initSlaveConstraint (int *numIDs*, const int * *idTypes*, const int * *IDs*, const int * *fieldIDs*, int *offsetOfSlave*, int *offsetIntoSlaveField*, const double * *weights*, double *rhsValue*) [pure virtual]

Initialize a slave constraint.

(Note to self: document the parameters.)

virtual int fei::MatrixGraph::parameters (int *numParams*, const char *const * *paramStrings*) [pure virtual]

Set parameter strings.

Parameters:

numParams Number of parameters being supplied.

paramStrings List of 'numParams' strings.

virtual int fei::MatrixGraph::setColumnSpace (fei::VectorSpace * *columnSpace*) [pure virtual]

Provide a **VectorSpace** (p. 29) to be used for looking up indices, field-masks, etc., for the column-space.

If no column-**VectorSpace** (p. 29) is provided, it will be assumed that the column-space equals the row-space.

Returns:

error-code 0 if successful

virtual int fei::MatrixGraph::setRowSpace (fei::VectorSpace * *rowSpace*) [pure virtual]

Provide a **VectorSpace** (p. 29) to be used for looking up indices, field-masks, etc., for the row-space.

If no column-**VectorSpace** (p. 29) is provided, it will be assumed that the column-space equals the row-space.

Returns:

error-code 0 if successful

The documentation for this class was generated from the following file:

- fei_MatrixGraph.h

fei::Matrix Class Reference

Abstract representation of an algebraic matrix.

This representation does not require that data be accessed only on the 'owning' processor. In other words, this representation may be used with an overlapping data decomposition. In most cases the underlying library-specific matrix will have a non-overlapping data decomposition (each equation uniquely owned by a single processor). Overlapping data may be assembled into this abstract matrix locally, and will be funneled into the underlying non-overlapping matrix on the correct processor when the **gatherFromOverlap()** (p. ??) method is called. Conversely, if the user wants to retrieve overlapping data from the matrix locally, that data is not guaranteed to be available until the **scatterToOverlap()** method is called.

virtual int fei::Matrix::copyIn (int numRows, const int * rows, int numCols, const int * cols, const double *const * values, int format = 0) [pure virtual]

Copy coefficients into the matrix, overwriting any coefficients that may already exist at the specified row/column locations.

Parameters:

numRows

rows

numCols

cols

values

format For compatibility with old **FEI** (p. ??) **elemFormat**... 0 means row-wise or row-major, 3 means column-major. Others not recognized

virtual int fei::Matrix::copyOutRow (int row, int len, double * coefs, int * indices, int & rowLength) [pure virtual]

Obtain a copy of the coefficients and indices for a row of the matrix.

Parameters:

row Global 0-based equation number

coefs Caller-allocated array, length 'len', to be filled with coefficients

indices Caller-allocated array, length 'len', to be filled with indices. (These indices will be global 0-based equation numbers.)

len Length of the caller-allocated coefs and indices arrays

rowLength Output. Actual length of this row. Not referenced if row is not in the local portion of the matrix.

Returns:

error-code non-zero if any error occurs.

virtual int fei::Matrix::getRowLength (int *row*, int & *length*) [pure virtual]

Get the length of a row of the matrix.

Parameters:

row Global 0-based equation number

length Output. Length of the row.

Returns:

error-code non-zero if any error occurs.

virtual int fei::Matrix::sumIn (int *blockID*, int *connectivityID*, const double *const * *values*, int *format* = 0) [pure virtual]

Sum coefficients, associated with a connectivity-block that was initialized on the **MatrixGraph** (p. 41) object, into this matrix.

Parameters:

blockID

connectivityID

values

format For compatibility with old **FEI** (p. ??) elemFormat... 0 means row-wise or row-major, 3 means column-major. Others not recognized

virtual int fei::Matrix::sumIn (int *numRows*, const int * *rows*, int *numCols*, const int * *cols*, const double *const * *values*, int *format* = 0) [pure virtual]

Sum coefficients into the matrix, adding them to any coefficients that may already exist at the specified row/column locations.

Parameters:

numRows

rows

numCols

cols

values

format For compatibility with old **FEI** (p. ??) elemFormat... 0 means row-wise or row-major, 3 means column-major. Others not recognized

virtual int fei::Matrix::sumInFieldData (int *fieldID*, int *idType*, int *rowID*, int *colID*, const double * *data*, int *format* = 0) [pure virtual]

Sum coefficients into the matrix, specifying row/column locations by identifier/fieldID pairs.

Parameters:

fieldID Input. field-identifier for which data is being input.

idType Input. The identifier-type of the identifiers.

rowID Input. Identifier in row-space, for which data is being input.

colID Input. Identifier in column-space, for which data is being input.

data Input. 1-D list representing a packed table of data. Data may be backed in row-major or column-major order and this may be specified with the 'format' argument. The "table" of data is of size num-rows X num-columns and num-rows is the field-size (i.e., number of scalar components that make up the field) of 'fieldID', as is num-columns.

format For compatibility with old **FEI** (p. ??) elemFormat... 0 means row-wise or row-major, 3 means column-major. Others not recognized

Returns:

error-code 0 if successful

virtual int fei::Matrix::sumInFieldData (int *fieldID*, int *idType*, int *rowID*, int *colID*, const double *const * *data*, int *format* = 0) [pure virtual]

Sum coefficients into the matrix, specifying row/column locations by identifier/fieldID pairs.

Parameters:

fieldID Input. field-identifier for which data is being input.

idType Input. The identifier-type of the identifiers.

rowID Input. Identifier in row-space, for which data is being input.

colID Input. Identifier in column-space, for which data is being input.

data Input. C-style table of data. num-rows is the field-size (i.e., number of scalar components that make up the field) of 'fieldID', as is num-columns.

format For compatibility with old **FEI** (p. ??) elemFormat... 0 means row-wise or row-major, 3 means column-major. Others not recognized

Returns:

error-code 0 if successful

virtual int fei::Matrix::writeToFile (const char * *filename*, bool *matrixMarketFmt* = true)
[pure virtual]

Write the matrix contents into the specified file.

Parameters:

filename Text name of the file to be created or overwritten. If in a parallel environment, each processor will take turns writing into the file.

matrixMarketFmt Optional argument, defaults to true. If true the contents of the file will be Matrix-Market real array format. If not true, the contents of the file will contain the matrix global dimensions on the first line, and all following lines will contain a space-separated triple with global row index first, global column index second and coefficient value third. Note also that if *matrixMarketFmt* is true, indices will be output in 1-based form, but if not true, indices will be 0-based.

Returns:

error-code 0 if successful, -1 if some error occurs such as failure to open file.

virtual int fei::Matrix::writeToStream (ostream & *ostrm*, bool *matrixMarketFmt* = true)
[pure virtual]

Write the matrix contents into the specified ostream.

Parameters:

ostrm ostream to be written to.

matrixMarketFmt Optional argument, defaults to true. If true the data will be written in MatrixMarket real array format. If not true, the stream will receive the matrix global dimensions on the first line, and all following lines will contain a space-separated triple with global row index first, global column index second and coefficient value third. Note also that if *matrixMarketFmt* is true, indices will be output in 1-based form, but if not true, indices will be 0-based.

Returns:

error-code 0 if successful, -1 if some error occurs.

The documentation for this class was generated from the following file:

- fei_Matrix.h

fei::Vector Class Reference

Abstract representation of an algebraic vector.

This representation does not require that data be accessed only on the 'owning' processor. In other words, this representation may be used with an overlapping data decomposition. In most cases the underlying library-specific vector will have a non-overlapping data decomposition (each equation uniquely owned by a single processor). Overlapping data (shared by local processor but the equation is owned by another processor) may be assembled into this abstract vector locally, and will be moved into the underlying non-overlapping vector on the correct processor when the **gatherFromOverlap()** (p. ??) method is called. Conversely, if the user wants to retrieve overlapping data from the vector locally for an equation that resides on another processor, that data is not guaranteed to be available until the **scatterToOverlap()** (p. 53) method is called. The **scatterToOverlap()** (p. 53) method does communication necessary to populate shared-but-not-owned data in the fei::Vector from data in the underlying algebraic vector.

From the point of view of fei::Vector, there are two types of data: owned and shared-but-not-owned.

Data Input (passing user data into the vector):

When locally-owned data is input, fei::Vector relays it immediately to the underlying algebraic vector. When shared-but-not-owned data is input, fei::Vector holds it in temporary storage. When **gatherToOverlap()** is called, fei::Vector moves it to the owning processor and then relays it to the underlying algebraic vector. At that point the temporary storage is deleted.

Data Access (retrieving data from the vector):

When locally-owned data is accessed, fei::Vector retrieves it from the underlying algebraic vector directly. In order to access shared-but-not-owned data (overlapped data), it is necessary first to call the method **scatterToOverlap()** (p. 53). This method does the communication necessary to re-create and populate temporary storage with the shared data by retrieving that data from the underlying algebraic vector on the owning processor and sending it to the sharing processors.

virtual int fei::Vector::copyOut (int *numValues*, const int * *indices*, double * *values*) [pure virtual]

Retrieve a copy of values from the vector for the specified indices.

Note that if the specified indices are not local in the underlying non-overlapping data decomposition, these values are not guaranteed to be correct until after the **scatterToOverlap()** (p. 53) method has been called.

virtual int fei::Vector::scatterToOverlap () [pure virtual]

Scatter data from the underlying non-overlapping data decomposition to the overlapping data decomposition.

In other words, update values for shared indices from underlying uniquely owned data.

virtual int fei::Vector::writeToFile (const char * *filename*, bool *matrixMarketFmt* = true)
[pure virtual]

Write the vector's contents into the specified file.

Parameters:

filename Text name of the file to be created or overwritten. If in a parallel environment, each processor will take turns writing into the file.

matrixMarketFmt Optional argument, defaults to true. If true the contents of the file will be Matrix-Market real array format. If not true, the contents of the file will contain the vector's global dimension on the first line, and all following lines will contain a space-separated pair with global index first and coefficient value second.

Returns:

error-code 0 if successful, -1 if some error occurs such as failure to open file.

virtual int fei::Vector::writeToStream (ostream & *ostrm*, bool *matrixMarketFmt* = true)
[pure virtual]

Write the vector's contents to the specified ostream.

Parameters:

ostrm ostream to be written to.

matrixMarketFmt Optional argument, defaults to true. If true the contents of the vector will be written in MatrixMarket real array format. If not true, the stream will be given the vector's global dimension on the first line, and all following lines will contain a space-separated pair with global index first and coefficient value second.

Returns:

error-code 0 if successful, -1 if some error occurs such as failure to open file.

The documentation for this class was generated from the following file:

- fei_Vector.h

fei::LinearSystem Class Reference

A simple container to bind a matrix and two vectors together as the matrix, rhs and solution of a linear system.

virtual int fei::LinearSystem::loadEssentialBCs (int *numIDs*, const int * *IDs*, int *idType*, int *fieldID*, int *fieldSize*, const double *const * *gammaValues*, const double *const * *alphaValues*)
[pure virtual]

Essential boundary-condition function that's similar to the 'old' **FEI** (p. ??)'s boundary-condition-loading function.

For each component of each field, a gamma-value and an alpha-value is supplied. If alpha is nonzero, then the boundary condition value is gamma/alpha. If alpha is zero, then no boundary condition is applied for that component.

Parameters:

numIDs

IDs

idType

fieldID

fieldSize

gammaValues Input. C-style table of values, num-rows = numIDs, num-cols = fieldSize.

alphaValues Input. C-style table of values, num-rows = numIDs, num-cols = fieldSize.

virtual int fei::LinearSystem::loadLagrangeConstraint (int *constraintID*, const double * *weights*, double *rhsValue*) [pure virtual]

Lagrange constraint coefficient loading function.

Parameters:

constraintID Input. Must be an identifier of a lagrange constraint that was initialized on the **fei::Matrix-Graph** (p. 41) object which was used to construct the matrix for this linear system.

weights Input. List, with length given by the sum of the sizes of the constrained fields.

rhsValue

virtual int fei::LinearSystem::loadPenaltyConstraint (int *constraintID*, const double * *weights*, double *penaltyValue*, double *rhsValue*) [pure virtual]

Penalty constraint coefficient loading function.

Parameters:

constraintID Input. Must be an identifier of a lagrange constraint that was initialized on the **fei::Matrix-Graph** (p. 41) object which was used to construct the matrix for this linear system.

weights Input. List, with length given by the sum of the sizes of the constrained fields.

penaltyValue

rhsValue

```
virtual int fei::LinearSystem::parameters (int numParams, const char *const * paramStrings)  
[pure virtual]
```

Set parameters on this object.

Currently two parameters are recognized: "debugOutput 'path'" where 'path' is the path to the location where debug-log files will be produced.

"name 'string'" where 'string' is an identifier that will be used in debug-log file-names.

The documentation for this class was generated from the following file:

- fei_LinearSystem.h

fei::Solver Class Reference

Interface for requesting that a linear-system be solved.

```
#include <fei_Solver.h>
```

Inheritance diagram for fei::Solver::

Public Methods

- virtual **~Solver** ()
virtual destructor
- virtual int **solve** (**fei::LinearSystem** *linearSystem, **fei::Matrix** *preconditioningMatrix, int numParams, const char *const *solverParams, int &iterationsTaken, int &status)=0
Solve a linear system.

The documentation for this class was generated from the following file:

- fei_Solver.h

fei::Factory Class Reference

Interface for creating fei:: interface instances. This interface inherits the various fei:: factory interfaces which in turn provide the method prototypes for creating instances of the corresponding interfaces. In addition to inheriting the fei:: factories, fei::Factory provides a method for creating instances of the 'old' **FEI** class.

Inheritance diagram for fei::Factory::

Public Methods

- virtual **~Factory** ()
virtual destructor
- virtual int **parameters** (int numParams, const char *const *paramStrings)=0
Set parameters.
- virtual int **createFEI** (LibraryWrapper *wrapper, MPI_Comm comm, **FEI** *&fei)=0
*Produce an instance of the "old" **FEI** class (implements the **FEI** 2.1 interface specification). The run-time type of this class is **FELImplementation**, which is implemented by the original body of FEI implementation code.*
- virtual int **createFEI** (MPI_Comm comm, **FEI** *&fei)=0
*Produce an instance of the "old" **FEI** class, with run-time type **snl_fei::Super**, which implements the **FEI** 2.1 interface specification using the newer modular implementation classes. In many cases **snl_fei::Super** should provide significantly better performance than **FEL Implementation**.*
- virtual int **getOutputLevel** ()=0
Query screen output-level (set by parameter-string "outputLevel n").

The documentation for this class was generated from the following file:

- fei_Factory.h

fei::VectorSpace::Factory Class Reference

Inheritance diagram for fei::VectorSpace::Factory::

Public Methods

virtual int fei::VectorSpace::Factory::createVectorSpace (MPLComm, const char * *name*, fei::VectorSpace *& *solnSpace*) [pure virtual]

Produce an instance of a **VectorSpace** (p. 29).

name may be NULL.

The documentation for this class was generated from the following file:

- fei_VectorSpace.h

fei::MatrixGraph::Factory Class Reference

Inheritance diagram for fei::MatrixGraph::Factory::

virtual int fei::MatrixGraph::Factory::createMatrixGraph (fei::VectorSpace * *rowSpace*, fei::VectorSpace * *columnSpace*, const char * *name*, fei::MatrixGraph *& *structure*) [pure virtual]

Produce an instance of a **MatrixGraph** (p. 41).

Either or both of *columnSpace* and *name* may be NULL. If *columnSpace* is NULL, it will be assumed that the structure to be created/defined is symmetric. i.e., *columnSpace* will be assumed to be identically equal to *rowSpace*.

The documentation for this class was generated from the following file:

- fei_MatrixGraph.h

fei::Matrix::Factory Class Reference

Inheritance diagram for fei::Matrix::Factory::

Public Methods

- virtual ~**Factory** ()

Usual virtual destructor.

- virtual int **createMatrix** (fei::MatrixGraph *matrixGraph, fei::Matrix *&matrix)=0

*Produce an instance of a **Matrix** (p. 49).*

The documentation for this class was generated from the following file:

- fei_Matrix.h

fei::Vector::Factory Class Reference

Inheritance diagram for fei::Vector::Factory::

Public Methods

- virtual **~Factory** ()
Usual virtual destructor.
- virtual int **createVector** (**fei::VectorSpace** *vecSpace, **fei::Vector** *&vector)=0
*Produce an instance of a **Vector** (p. 53) using a **VectorSpace** (p. 29).*
- virtual int **createVector** (**fei::VectorSpace** *vecSpace, bool isSolutionVector, **fei::Vector** *&vector)=0
*Produce an instance of a **Vector** (p. 53) using a **VectorSpace** (p. 29).*
- virtual int **createVector** (**fei::MatrixGraph** *matrixGraph, **fei::Vector** *&vector)=0
*Produce an instance of a **Vector** (p. 53) using a **MatrixGraph** (p. 41).*
- virtual int **createVector** (**fei::MatrixGraph** *matrixGraph, bool isSolutionVector, **fei::Vector** *&vector)=0
*Produce an instance of a **Vector** (p. 53) using a **MatrixGraph** (p. 41).*

The documentation for this class was generated from the following file:

- fei_Vector.h

fei::LinearSystem::Factory Class Reference

Inheritance diagram for fei::LinearSystem::Factory::

Public Methods

- virtual **~Factory** ()
Usual virtual destructor.

- virtual int **createLinearSystem** (**fei::MatrixGraph** *matrixGraph, **fei::LinearSystem** *&linearSystem)=0

*Produce an instance of a **LinearSystem** (p. 55).*

The documentation for this class was generated from the following file:

- fei_LinearSystem.h

fei::Solver::Factory Class Reference

Inheritance diagram for fei::Solver::Factory::

Public Methods

- virtual ~**Factory** ()
Usual virtual destructor.
- virtual int **createSolver** (**fei::Solver** *&solver)=0
*Produce an instance of a **Solver** (p. 57).*

The documentation for this class was generated from the following file:

- fei_Solver.h

Distribution:

Internal:

1	MS 0384	T.C. Bickel	9100
1	MS 9003	K.E. Washington	8900
1	MS 0384	H.S. Morgan	9140
1	MS 0382	J.R. Stewart	9143
1	MS 0382	E.A. Boucheron	9141
1	MS 0380	K.F. Alvin	9142
1	MS 1110	D.E. Womble	9214
1	MS 9917	S.W. Thomas	8962
1	MS 9915	M.L. Koszykowski	8961
1	MS 0382	H.C. Edwards	9143
1	MS 0382	K.D. Copps	9143
1	MS 0382	G.D. Sjaardema	9143
1	MS 0382	J.R. Overfelt	9143
1	MS 0382	K.N. Belcourt	9143
1	MS 0382	K.M. Aragon	9143
1	MS 0382	D.M. Brethauer	9143
1	MS 0382	M.E. Hamilton	9143
10	MS 0382	A.B. Williams	9143
1	MS 0382	S.W. Bova	9141
1	MS 0382	S.P. Domino	9141
1	MS 0382	T.O. Okusanya	9141
1	MS 0382	C.K. Newman	9141
1	MS 0382	R.R. Lober	9141
1	MS 0382	A.A. Lorber	9141
1	MS 0382	S.R. Subia	9141
1	MS 0380	J.D. Hales	9142
1	MS 0380	K.H. Pierson	9142
1	MS 0380	M.K. Bhardwaj	9142
1	MS 0380	G.M. Reese	9142
1	MS 0380	T.F. Walsh	9142
1	MS 0382	P.K. Notz	9114
1	MS 0826	D.K. Gartling	9100
1	MS 0836	R.E. Hogan	9116
1	MS 0847	C.R. Dohrmann	9124

1	MS 9159	M.F. Adams	9214
1	MS 1110	M.A. Heroux	9214
1	MS 0316	R. Hooper	9233
1	MS 0316	J.N. Shadid	9233
1	MS 0316	T.M. Smith	9233
1	MS 0370	R.A. Bartlett	9211
1	MS 1110	D.M. Day	9214
1	MS 0316	R.J. Hoekstra	9233
1	MS 9159	R.S. Tuminaro	9214
1	MS 9018	Central Technical Files	8945-1
2	MS 0899	Technical Library	9616

External:

Evi Dube
Lawrence Livermore National Laboratory
P.O. Box 808 - Mail Stop L-130
Livermore, CA 94551-0808

Brad Wallin
Lawrence Livermore National Laboratory
P.O. Box 808 - Mail Stop L-130
Livermore, CA 94551-0808

Jeff Keasler
Lawrence Livermore National Laboratory
P.O. Box 808 - Mail Stop L-130
Livermore, CA 94551-0808

James Reus
Lawrence Livermore National Laboratory
P.O. Box 808 - Mail Stop L-130
Livermore, CA 94551-0808

David Stevens
Lawrence Livermore National Laboratory
P.O. Box 808 - Mail Stop L-130
Livermore, CA 94551-0808