Framework for the Analysis of Physical Behaviour with the Finite Element Method (FEM)

Bauhaus-Universität Weimar, Informatik im Bauwesen

2009

Contents

Framework for the Analysis of Physical Behaviour with the Finite Element Method (FEM)

1 Installation

Different forms of utilizing the Framework are supported:

Users who are only interested in utilizing the Framework for writing new applications without requiring access to the source code, can simply link corresponding class definitions to their own application development. For this, a Java class archive is provided under the name ***FEM-Framework.jar***, which can be downloaded and included in the build-path of the compilation of new applications:

e.g. *javac -sourcepath source -cp source;FEM-Framework.jar -d binary source\heat\*.java*

In this example the Java compiler expects input source files to be stored (-*sourcepath*) under the source path "source", user class files and annotation processors (-*cp*) to be stored under the class path "source" including the class archive "FEM-Framework.jar" and to place generated class files (-*d*) in subdirectory "binary". The user source files to be compiled are specified as "source\heat\*.java".

If access to source files of the Framework is desired, then a zip-file containing the complete source files of the Framework may be downloaded (***FEM-Framework.zip***). The zip-file may be unpacked and the sources may be edited and linked to the own application development or a new Java class archive may be generated and used as outlined above. This approach is advisable if only limited modifications are applied to the source code files.

If not only access to the source files is required but if major development efforts are to be undertaken, an "Integrated Development Environment – IDE" is a preferable choice. A powerful and free IDE is available, for example, under "eclipse" – Eclipse IDE for Java Developers (http://www.eclipse.org/downloads/). In this case, the zip-file with the source files may be unpacked and the sources may be imported into the IDE.

The additional overhead of installing and learning an IDE may seem prohibitive to many users but over time with many enhancements and modifications to be performed, the additional investment quickly pays off. New efforts often start by "only needing to do little development" but quickly turning into something much more complex as initially intended.

It is our experience that even students in class quickly appreciate the additional help and support from an IDE over the tedious development using console input with a context sensitive editor and a standard compiler invocation. They are ready to accept the additional efforts for learning an IDE and this even motivates them better.

2 Introduction

The Finite Element Method (FEM) is a general method for analyzing specific types of physical behaviour governed by a specific class of differential equations.

Specifically in civil engineering it is frequently used for solving problems in structural analysis, in heat transfer and fluid flow.

Although the fundamental concepts of this method have been long known, the viability of using this method only became possible with the availability of powerful computer hardware and software. In fact, it was an engineer in Germany – Konrad Zuse – who invented the main concepts of modern computers and application software because he was tired of having to solve problems in structural analysis with conventional techniques. These techniques like for example moment distribution methods were very tedious and developed for “hand calculations” by engineers. Initially, computer software was used to code the old methods that were developed with totally different restraints in mind.

The first publication credited with with using the term Finite Element Method was üublished in 1960 by Ray W.. Clough [1]. In the early 1960’s a group of structural engineers in Berkeley developed new application software for engineering problem solving based upon the Finite Element Method. Professors Ray Clough, Edward Wilson and Robert Taylor developed software as a basis for commercial products and software suitable for teaching the concepts of the FEM to graduate students. The software SAP (Structural Analysis Program) subsequently was the basis for many commercial products. The software CAL (Computer Aided Language) and FEAP (Finite Element Analysis Program) were mainly used for teaching and research purposes. Even though limitations of computer hardware in terms of storage capacity and processing power restricted the use of the software to relatively small problems, the generality of the approach could be shown and used for teaching the concepts in graduate courses.

In the 1970’s research and development around the FEM became a major aspect of engineering work and education. One of the main hindrances, however, was the fact that limited resources in memory storage capacity and computing power prevented the application of the method for large problems. A lot of research was invested in solving large systems (i.e. in FEM resulting in a system of a large number of linear equations) on computers with limited resources. Practical problems easily resulted in systems with thousands of equations. Thousands of equations were stressing the capabilities of computers often beyond the point of applicability. Civil engineers took advantage of certain characteristics of the resulting system of linear equations. In FEM these systems often are sparsely populated with a specific structure. “Banded” or “profiled” structures could be exploited by only storing and accessing information within a “band” or “profile” of a system of linear equations. Then in the mid-1970’s a new development in computer technology solved one major problem in this regard. A new machine called the VAX (Virtual Address eXtension) running under an operation system VMS (Virtual Memory System) made the company DEC (Digital Equipment Corporation) the main supplier of hardware for technical, scientific problem solving. It solved the storage capacity of internal memory restrictions by automatically expanding it by external memory – extending internal limits *virtually* by external ones. With this development, the limits on solving large problems were not restricted by limitations on internal memory anymore but only by computing power. For most practical purposes this meant that modern desktop computers were sufficiently powerful to solve most standard problems of engineering practice. In research, however, these limits are still being pushed further with new solution approaches or model-oriented, 3-dimensional analyses.

In the early 1980's a totally new area of application programming in structural engineering evolved with the field of CAD (Computer Aided Design). CAD was intended to support the creative design process of engineering problem solving. Original ideas foresaw CAD as a tool to interactively create digital models that could be used to visualize engineering context intuitively and to support digital simulations. Digital models were hoped to replace the cost-intensive "plastic models" used in process engineering. In structural engineering it was hoped that CAD would replace technical drawings as a basis for communication in engineering projects. It turned out that these expectations could not be fulfilled as easily as was hoped originally. Until today by far the majority of all projects in structural engineering is carried out on the basis of technical drawings, thus limiting the term CAD to Computer Aided *Drafting*.

With the separate evolvement of FEM and CAD for models used in structural analysis and structural design the need arose to integrate information shared between these two models. Consequently, in the 1980's the challenge of integrating separate models in structural engineering developed into growing importance. The term product modelling was coined to address the challenges. Technologies like STEP (Standards for The Exchange of Product data) were developed for solving these purposes.

In the early 1990's another important development took place in structural engineering. New concepts developed in computer science for implementing semantically rich application software were adopted in structural engineering. Previously, all information in engineering application software was reduced to an independent set of simple data types like integer numbers, float numbers and characters. The consistent handling of interrelations between this information was solely in the responsibility of the application programmer. With the new concepts of "Object Oriented Design and Programming" a far better improved environment for modelling engineering context became available. Engineering information could be modelled in a semantically rich way in form of engineering "objects" with formally defined interrelations between these objects.

This lead to another important development in form of an attempt to standardize the definition of engineering objects under the term IFC (Industry Foundation Classes) by an association called IAI (International Alliance for Interoperability). IFC's were developed to serve as an integration basis between separate software applications in civil engineering.

Modern applications in structural engineering are therefore based upon concepts of object-oriented programming with provisions for integrating information between separate applications.

It is the purpose of this framework to define a basis for Finite Element applications in structural engineering that is in conformance with the modern concepts mentioned above and that can be utilized to teach these concepts in graduate engineering education.

A major goal is the provision to easily **extend the functionality** of the implementation provided. Ideally, an engineering student can restrict his efforts to the engineering contents of a solution. The effort to include a new element for added functionality should ideally be restricted to the definition of the behaviour of the element and to clearly defined places within the code. The implementation provided in the context of this context includes functionality for

- two-dimensional structural analysis with beam, truss and spring elements,
- dynamic motion analysis,
- two- and three-dimensional stationary heat transfer analysis,
- transient heat analysis and for
- two- and three-dimensional linear elasticity analysis with constant stress elements.

More elements are available under the context of this framework but are not considered in this context in order to keep the concepts concise and easily understandable.

Another goal is the **integration of the analysis model with the design model**. For these purposes a CAD implementation is provided under the name *CADemia* that is built upon the same technological basis and that utilizes the same object model as the FEM implementation. Often, the graphical representation of input and output of analyses for the FEM is confined to a “passive” graphical display of information. The user can mainly view the information without being able to interactively access and utilize that information. Functionality for these purposes is also provided in the context of a specific application in order to discuss the benefits and shortcomings of such an approach as opposed to an integration with CAD.

The framework is not intended to serve as a basis for professional purposes in structural engineering. It lacks for considerations of professional usability and considerations of performance. It is strictly intended to demonstrate the underlying concepts and to teach modern application programming in structural engineering to engineering students.

3 Technological Basis of the Implementation

The technological basis of the implementation is the Java™ Standard Edition 6 (Java SE 6) with Java Development Kit 6 (JDK 6) by Sun Microsystems, Inc (Copyright © 1995-2006) [2]. The concepts and implementation are documented in an excellent way in a freely available online tutorial [3].

It is assumed that the reader is familiar with the concepts of object-oriented programming in general and with the basic concepts of JDK in particular.

Important concepts used in the object-oriented design of the implementation include the utilization of *abstract classes* for general functionality associated with objects in the FE design. This way, application specific objects can be derived from the abstract classes and only need to implement the *additional* application specific functionality. Also, the concept of *interfaces* is utilized to define specific functionality that must be implemented by classes to be used in the context of the object-oriented FEM framework.

The user interface provided is primarily directed towards console input and output. This is done in order to provide an implementation for engineering students that is as simple as possible and free of the additional concepts of graphical user interfaces. This interface, however, is strictly modularized and can easily be replaced by another one. Two other ones with restricted functionality are included in order to demonstrate this concept:

- a simple graphical user interface for the visualization of the model and
- a complete CAD interface for an interactive handling of the model.

The GUI (Graphical User Interface) concepts are based upon the Java Swing technology. The project Swing is described in the Java documentation as follows:

“The Swing toolkit includes a rich set of components for building GUIs and adding interactivity to Java applications. Swing includes all the components you would expect from a modern toolkit: table controls, list controls, tree controls, buttons, and labels.

Swing is far from a simple component toolkit, however. It includes rich undo support, a highly customizable text package, integrated internationalization and accessibility support. To truly leverage the cross platform capabilities of the Java platform Swing supports numerous look and feels, including the ability to create your own look and feel. The ability to create a custom look and feel is made easier with Synth, a look and feel specifically designed to be customized. Swing wouldn't be a component toolkit without the basic user interface primitives such as drag and drop, event handling, customizable painting and window management.

Swing is part of the Java Foundation Classes (JFC). The JFC also include other features important to a GUI program, such as the ability to add rich graphics functionality, and the ability to create a program that can work in different languages and by users with different input devices.”

The choice of this technology has several advantages.

- First, all functionality is fully available in source code thus enabling engineering students not having to rely on “black box” or “canned” software but rather being able to look “behind” implementations for fully understanding concepts, limitations and chances.
- Second, there are no requirements on a specific environment for this software to run in. Java is conceived as a “platform independent” technology. Resulting code will run under any environment supporting a Java Virtual Machine (JVM) which basically all commercially available hardware does. Many applications from totally other areas of work often running on the Internet also require JVM.
- Third, all of the technology required for these purposes is available free of cost under the Open Source Project (http://www.opensource.org/docs/definition.php) for free redistribution, free access to source code, free extension of derived work under the same concepts, with provisions for the integrity of the author’s source code, without discrimination against persons or groups, without discrimination against fields of endeavour, for distribution of license, with the requirements that a license must not be specific for a product, that a license must not restrict other products and that it must be technology-neutral.

Of course, there are also objections against using this technology for engineering or scientific purposes. The main objection for years was focused on performance of resulting implementations. Performance for large engineering problems traditionally was a major concern for the choice of technology. Historically it has happened before that technology that was believed to be superior in terms of concepts in computer science was not accepted in engineering because the resulting run-time performance was inferior to other technologies. The authors believe however, that advances in computer hardware now have solved this problem for all but the largest problems. For most practical purposes the technology is now believed to be totally appropriate and to deliver the performance required.

Another concern that was raised was geared towards the missing richness of functionality needed for scientific and engineering problem solving. Functionality needed in Linear Algebra, in Complex Data Structures, in Graphical Display and in Graphical User Interfaces were not developed to the same degree that was available in other environments. Again, the authors firmly believe that this problem has been resolved by now and that a developer can firmly rely on all general functionality to be available for the solution of his problems.

Finally, the authors believe it to be absolutely essential that engineers will have to be involved in developing new software solutions for advances in engineering. Initially, with the availability of powerful computer hardware, engineers were heavily involved in developing software for engineering problem solving. In the early 1990's however, with the emergence of Graphical User Interfaces and Object-Oriented Design and Implementation, a redesign of engineering applications required extensive Know-How not available for most engineers. Also, existing solutions had grown so rich in functionality that so-called "legacy systems" were established making it difficult for any new system to even come close in terms of functionality offered. Now, with the availability of the technology described above, the authors believe it not only to be possible but also a necessity that engineers will have to get involved in creating and implementing new solutions much more again. The burden of learning the new platform is regarded as very limited lending itself properly to engineers as well and the benefits are regarded as potentially huge.

Advances in civil and structural engineering in terms of software solutions will not appear automatically and cannot be created by computer science alone but rather have to be created by engineers who understand the problems and the technology as well.

4 Hierarchical Structure of the FEM Framework

Complex implementations should be structured in a modular way in order to clearly identify separate modules. Each module consists of a set of classes.

The FEM Framework is stored in the file system under the *root* directory FEM-Framework. The *root* directory of this directory structure may be renamed as required.

Java supports the definition of class hierarchies via the concept of packages. A package is stored in the file system of the operating system. The association with a package is defined in the source files of the implementation via the keyword *package* followed by a unique name for the package. The position in the class hierarchy is defined by a path definition relative to the root directory with the "**.**" operator as separator,

e.g. *package framework;*
package framework.model

The implementation of the general Framework is stored in the directory ***FEM-Framework*** as the root-directory with subdirectories for executable code and source code. Directory ***binary*** contains the Java runtime-code and ***source*** the complete source code for the implementation.

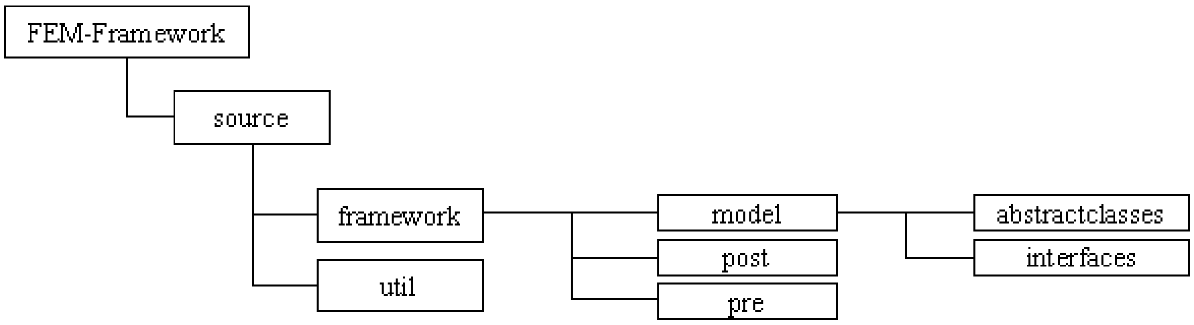
Subdirectory ***framework*** contains source code for general **model**, **post**- and **pre**-processing functionality that is independent from a specific application. Subdirectory **model** is differentiated into the implementation of abstract functionality and the interface definitions hierarchically structured in the corresponding subdirectories ***abstractclasses*** and ***interfaces***.

Subdirectory ***util*** provides general utility functionality. The source code of a class definition can access other classes in the hierarchy by importing the required classes from the corresponding packages using the keyword *import*.

e.g. *import framework.model.Model;*

This statement appears at the top of a source code that will need access to class Model in the subdirectory framework/model under the source directory.

The following table shows the package structure of the FEM Framework:

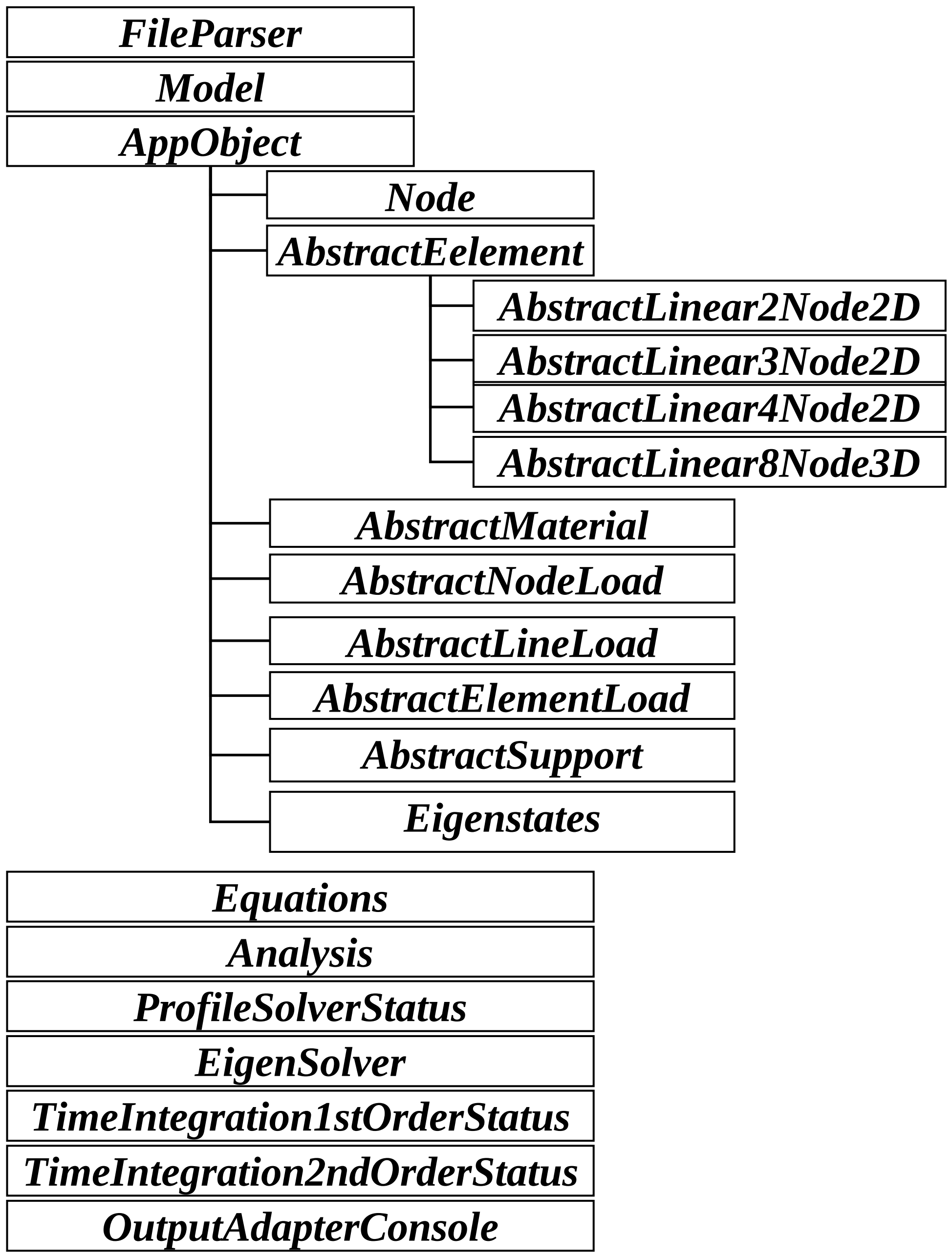


The general contents of individual packages are described in the following table:

framework	general FEM functionality that is independent from specific applications
framework.model	classes for generating and storing general, application-independent model functionality and for processing of model data, i.e. generating and solving system equations
framework.model.abstractclasses	application independent FEM components that cannot be instantiated
framework.model.interfaces	required general functionality that must be implemented by applications
framework.post	general functionality required for output of model definition and results
framework.pre	classes for controlling the flow of the analysis and for generating and solving the system of linear equations
util	several classes with utility functionalities

4.1 Class hierarchy of the Framework

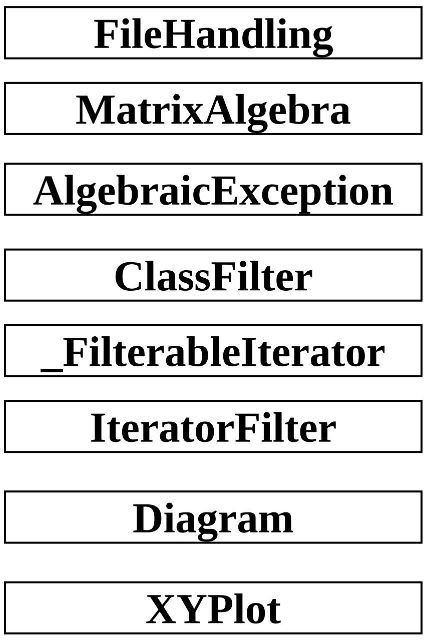
All classes in Java are implicitly derived from *java.lang.object*. The Framework defines a class *FileParser* for reading input values for a specific model from an external file (preprocessing). Class *Model* collects all objects for a specific application. Class *AppObject* collects all application objects (nodes, abstract elements, material, loads, supports and time integration). Class *Analysis* controls the flow of model analysis. Class *ProfileSolverStatus* supports the solution of a linear system of equations, class *EigenSolver* evaluates the eigensolution of a system of equations, classes *TimeIntegration1stOrder* and *TimeIntegration2ndOrder* perform the time integration analysis for first and second order differential equations. Class *OutputAdapterConsole* controls alphanumeric output of system results.



Additional utility functions are defined in the FEM-Framework under the hierarchy *util*. These provide general functionality for selecting and opening input files, for various tasks in linear algebra including *AlgebraicExceptions* for errors occurring when performing matrix operations and for iterating on model objects utilizing various filters for accessing specific objects.

The most important filter is the *ClassFilter* supporting access to and traversal of objects by class type.

Finally two class definitions are provided for displaying diagrams for information output and for plotting x,y-curves in an interactive window.



5 Basic concepts for the Implementation

Some important basic concepts need to be treated before the implementation of the Framework will be documented and explained. These include primarily the aspect of object identification, of object relations and of storing model information for use during run-time as well as persistently. Ordering of information for purposes of structured output is also an important aspect for accessing and storing model information. All of these concepts were originally developed in [4] and adopted for the purposes of this implementation.

5.1 Object Identification

The Finite Element Model consists of a set of objects of specific classes. The objects are classified as Nodes, Elements, Materials, Loads, Supports, etc.. Each class is a generalized data type consisting of attributes and methods.

In general, many instances (objects) will be derived from a single class definition. It is of vital importance that all of these objects can be uniquely identified. The scope of the identification in case of the FEM Framework is chosen to be the context of the application. There is no guarantee that the identification will be unique beyond the scope of the corresponding applications. Specifically when designed to work on the Internet, other applications will set up identifiers that are most likely to be unique in a global focus in form of a GUID (Globally Unique IDentifier). Java will generate unique identification for strictly internal purposes under the concept of Java references with the focus of a session. The user has no control over the contents of the identifier and little control over its utilization. Other languages (e.g. C++) will support access to internal identifiers for instance via the concept of specific data types (pointers). Internal identification will, however, only be generated when the application is started and will not be stored persistently. Therefore, a unique identification is only ensured during a single session of the application but not across sessions of an application. This is totally acceptable in the context of an individual session but it becomes a major problem when talking about integration of information between different models across different sessions. Many CAD systems, for example, will generate references to internal objects when the system is started, e.g. in form of a “handle” which is thrown away when the session is finished. Therefore, any external reference to engineering objects in such systems, e.g. wall number xyz, will not be valid anymore next time the application is started.

Also, internal references can only be established to objects that already exist. Sometimes it becomes very useful to refer to objects that are defined to be part of the system but that have not been allocated yet, i.e. there is no address pointing to a specific memory location. This is equivalent to referring to a person that is known by name without any knowledge of the address where to find that person. That knowledge is only established, e.g. by looking up the current address, when access to that person is required. In case the address was recorded some time ago, there is no assurance that it will still be valid. Either a mechanism will be established for ensuring the validity of a specific address at any point in time or an address will not be recorded at all but rather a mechanism will be established for looking up the current address only when needed.

For these reasons, an additional type of external identification was included with the FEM Framework that is based upon Unicode-String identifiers that are defined by the user and that are stored persistently across different sessions of an application and that can even be used across applications.

Therefore, each object of the FEM Framework will have an Id (e.g. ***nodeName***) as an external identifier. This Id is defined by the user and passed as an argument to the constructor of the object. In general, the parser will read it from persistent storage and include it in the invocation of the corresponding constructor when generating the new object.

e.g.

```
new Node ( nodeName, crds, nodalDegreesOfFreedom);
```

Any other object can now refer to this node via its ***nodeName***. This name will be assured to be unique within the context of the Framework and stored persistently for further use across separate sessions of the application or across totally separate applications.

In a next step, the Id is passed to the Framework via the constructor of the object. This is accomplished by deriving any new object from class *AppObject* and passing the ***id*** to it via the constructor of the super class *AppObject*.

e.g.

```
public class Node extends AppObject implements INode  {

    public Node ( String id, double[ ] crds, int ndof) {
        super(id);
        spatialDimension = crds.length;
        setCoordinates(crds);
        this.nodalDegreesOfFreedom=ndof;
    }
```

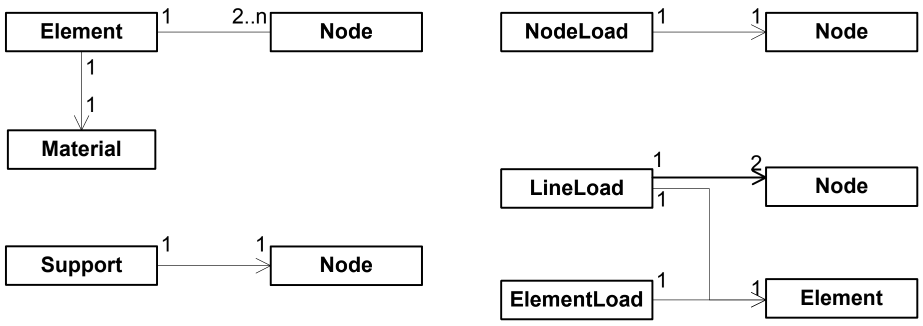
The Framework ensures the id to be unique and also ensures its availability for establishing relations to this new object.

5.2 Relations between Objects

Objects of the FEM Framework are characterized by various interrelations to other objects in the model. An Element will be composed of several (at least two) Nodes and must have a specific Material definition.

A node load may be associated with a specific Node, a line load with two specific Nodes or a specific Element and an element load may be associated with a specific Element.

A support condition for the system must be defined for exactly one Node.



In the FEM Framework the object relations will be specified by the user based upon the external identifiers. This will be done in an external Unicode-file that will be the basis for generating a model and that will serve as a persistent storage of the model.

Initially, all relations are solely defined on the basis of persistent string identifiers. After reading the complete set of FEM components, for example at the end of each application parser, internal references to related components are evaluated based upon the string identifiers by specific functionality that needs to be defined for all components requiring access to related components. For example, a Node component does not require relations to other FEM components and thus does not need to define the corresponding functionality. However, an element needs to set up relations to its corresponding Node, Material and possibly Cross Section objects and therefore needs to provide the corresponding functionality for evaluating the necessary internal references for accessing these objects. This is accomplished in the general class *AbstractElement,* which is valid for any Element of the Framework, for *nodeId*, *materialId* and possibly *crossSectionId* that are used to retrieve the required object references from the model.

```
public void setReferences(Model m_model) throws FemException {
    // establish reference to each node of the element
    for (int i = 0; i < nodesPerElement; i++) {
        node[i] = (INode) m_model.getModel().getObject(nodeId[i]);
        if (node[i] == null) throw new FemException("*** Abstract Element: Node "
                        + nodeId[i] + " of Element " + getId() + " is not defined ***");
    }
    // establish reference to the element material
    material = (IMaterial) m_model.getObject(materialId);
    if (material == null) throw new FemException("*** Abstract Element: Material "
                        + materialId + " of Element " + getId() + " is not defined ***");
    // some elements require specification of a cross section
    if(crossSectionId != null) crossSection =(ICrossSection)
                                        m_model.getObject(crossSectionId);
}
```

Setting up relations via string identifiers (names) allows for a strict separation of a uniquely defined object from its internal reference (address). As outlined above, the concept is comparable to referring to a person by name without any knowledge about the current whereabouts and even existence. Both can be established when required from a separate address book.

5.3 Persistent Storage of Object Model Information

The external Unicode-file mentioned above is also used for storing the analysis model persistently. It contains all information for a complete description of the model. It is however not an efficient basis for the analysis of the model at run-time. For these purposes a suitable data structure is generated transiently when the application is started. A parser is provided for parsing the external Unicode-file and for generating the transient data structure of the analysis model on the basis of the persistent model information.

The structure of the external Unicode-file is determined by a number of keywords characterizing specific information following each keyword until a blank line is encountered.

Class ***FileParser*** is provided for parsing of model information that is persistently stored in an input file in Unicode format.

The functionality is contained in *package framework.pre*. It needs to import general Java functionality about Java file input and output (IO). Separate sections of the input file are defined for each individual set of components. Each section is defined by a preceding **keyword**.

Overview over supported **keywords** and associated **number** of input items in class *FileParser*:

```
//      identifier                          1 modelName

//      dimensions                          2 spatialDimensions, nodalDegreesOfFreedom

//      nodes                               1 number of nodal degrees of freedom
//                                          2 name, x
//                                          3 name, x, y
//                                          4 name, x, y, z

//      nodeGroup                           1 mandatory first line for ID prefix
//                                          1 x
//                                          2 x, y
//                                          3 x, y, z

//      uniformNodeArray    dimension=1     4 nodeInitial, x, xInterval, nIntervals
//                          dimension=2     7 nodeInitial, x, xInterval, nIntervalsX,
//                                            y, yInterval, nIntervalsY
//                          dimension=3    10 nodeInitial, x, xInterval, nIntervalsX,
//                                            y, yInterval, nIntervalsY,
//                                            z, zInterval, nIntervalsZ

//      variableNodeArray   dimension=1     ? meshSpacings
//                                          2 nodeInitial, x
//                          dimension=2     ? meshSpacings
//                                          3 nodeInitial, x, y
//                          dimension=3     ? meshSpacings
//                                          4: nodeInitial, x, y, z

//      eigenstates                         2 name, numberOfStates
```

Class *FileParser* provides a constructor of class *FileParser* which will require the name of the file to be parsed as a parameter and will throw exceptions if errors occur during input and output or during parsing of information. It will then check if the file to be parsed exists and it will subsequently set the size of the file. It will create an instance of class *BufferedReader* and *FileReader* for parsing the input file.

Finally, it will provide methods for processing predefined keywords in order to analyze the contents associated with it.

Class *FileParser* will also provide a method *getModel* for retrieving a model object.

The process of parsing input for a FEM analysis from a file is rather schematic:

- A **keyword** identifies a set of corresponding data (lines) to be read,
- each subsequent line contains a specified **number of arguments** defining the information needed for generating a new FEM component. Each line is analyzed item by item (token by token),
  arguments are separated by predefined separators, i.e. blank, tab, comma
  StringTokenizer tokenizer = new StringTokenizer(s, " \t,");
- the constructor of the new component is invoked with the items as parameters and
- this is carried on until an empty line is encountered completing the particular keyword section.

Once a keyword section is completed, method *reset* will cause reading the input file from the beginning again for the next keyword to be processed.

Method *parseIdentifier* is defined in the framework. Keyword **identifier** requires input of a string for a new model identifier, otherwise an exception will be thrown. The string will be passed to the constructor of a new object of class model.

Method *parseDimensions* is defined in the framework. Keyword **dimensions** requires input of two integer numbers indicating the spatial dimensionality of the problem to be solved and the number of nodal degrees of freedom (e.g. 2   1 defines a 2-dimensional problem with 1 degree of freedom per node).

Method *parseNodes* is also defined in the framework. Keyword **nodes** is used for parsing node components of a FEM model. It throws an exception if a given node identifier already exists (not unique) and if the number of parameters found in the input line is not 1 (spatial dimension), 2 (name and x-value), 3 (name and x,y-values) or 4 (name and x,y,z-values).

Nodes may also be generated. For this purpose two different keywords are available.

Keyword ***nodeGroup*** will generate a set of node components to be specified by a prefix for nodal identifiers of the complete group and nodal coordinates. The prefix is to be specified in the first line after the keyword. Nodal coordinates are specified in a set of subsequent lines. Nodal identifiers will be generated from the prefix and an incrementally generated counter with six digits.

nodeGroup		
edge node		
10.0	20.0	30.0
20.0	20.0	30.0

create two new node components with the following string IDs generated

"edge node000000"
"edge node000001"

Keyword ***uniformNodeArray*** will generate a mesh of equally spaced node components. This is accomplished by entering an initial identifier for all nodes to be generated, followed by starting coordinate(s), the spacing(s) of the mesh and the number of intervals in each direction. This process is supported for 1, 2 and 3 dimensions in which case the starting coordinate, the mesh spacing and the number of intervals need to be specified for each dimension. Unique identifiers for each node will be generated based upon the initial identifier which will then be concatenated with integer numbers for each successive interval, e.g.

uniformNodeArray			
N	0.	2.	10

Generate a 1-dimensional array of 10 nodes at a spacing of 2. units starting from the origin.

New nodes will be generated with respect to the origin specified (0.) Each node will have a unique identifier based upon the initial identifier (N000000 through N000009).

uniformNodeArray						
n	0.	1.	3	1.	1.	3

Generate a 2-dimensional array of 3x3 nodes from x=0. at a spacing of 1 for 3 nodes and from y=1. at a spacing of 1. for 3 nodes.

New nodes will be generated with respect to the origin specified (0.;1.) Each node will have a unique identifier based upon the initial identifier appended by another 3 digits in x- and 3 digits in y-direction generated sequentially (n000000, n000001, n000002, n001000, n001001, n001002, n002000, n002001, n002002).

Keyword ***variableNodeArray*** will generate a non-uniformly spaced mesh of nodes based upon a sequence of mesh distances to be entered in the first line after the keyword. The mesh distances to be specified will define a sequence of distances (offset) with respect to the origin (initial coordinates). The number of distances entered will determine the number of nodes to be generated. For this purpose, simply an initial nodal identifier and nodal coordinates for the mesh origin need to be entered, e.g.

variableNodeArray			
0.	1.	3.	6.
N	0.	0.	

Will generate an array of nodes at a mesh spacing of 0, 1, 3 and 6 units from the origin for each spatial dimension, here 2.

New nodes will be generated with respect to the origin specified (0., 0.) Each node will have a unique identifier based upon the initial identifier appended by another 3 digits in x- and 3 digits in y-direction generated sequentially (N000000, N000001, N000002, N000003, N001000 through N003003).

variableNodeArray			
0.	1.	3.	6.
N	0.	0.	0.

Will generate an array of nodes at a mesh spacing of 0, 1, 3 and 6 units from the origin for each spatial dimension, e.g. 3.

New nodes will be generated with respect to the origin specified (0., 0., 0.) Each node will have a unique identifier based upon the initial identifier appended by another 2 digits in x-, 2 digits in y- and 2 digits in z-direction generated sequentially (N000000, N000001, N000002, N000003, N000010 through N030303).

Method *parseEigenstates* is used for parsing eigenvalue and eigenvector information of an FEM model. Keyword **eigenstates** requires input of an identifier and the number of eigenstates to be considered.

The algorithm implemented for evaluation of eigenstates is a vector iteration for the smallest eigenvalues of the general eigenvalue problem with symmetric real profile matrices of the form:

$$\mathbf{A}\mathbf{x} = m\mathbf{B}\mathbf{x}$$

Matrix **B** is assumed to be a diagonal matrix which can be stored in a single-dimensional array. Matrix **B** is evaluated internally either as the specific heat matrix in heat transfer analysis or as the mass matrix in structural dynamics analysis.

Finally, a new *Eigenstates* object is added to the model depending if the general case with a general B-matrix is applicable or not.

5.4 Transient Storage of Object Model Information

The structure of the transient storage is defined by data structures available as Java container classes.

The transient, internal data structure of the model is generated when reading the persistent model information from an external file. It is implemented in ***class Model*** and consists of an ***objectMap*** of type ***HashMap*** providing a mapping between persistent object IDs (Strings) and references to the associated application objects. This structure is used for ensuring uniqueness of persistent object IDs and for accessing application objects via their corresponding IDs.

private Map<String, AppObject> objectMap = new HashMap<String, AppObject>( );

It furthermore consists of a ***componentSet*** of type ***TreeSet*** providing access to all application objects via its type, e.g. elements, nodes, etc.. This is used for traversal of application objects by type.

private TreeSet<AppObject> componentSet;

A *TreeSet* is a data structure in Java that provides a hierarchical ordering of elements in lexicographical, “natural” order. It provides guaranteed log(n) time cost for the basic operations (add, remove and contains).

6 Defining Element Geometries

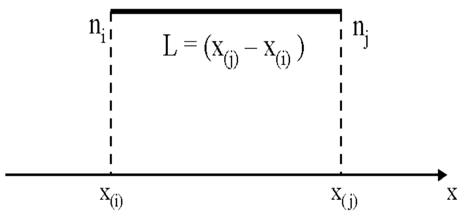
The geometry of elements to be used in a FEM analysis can be defined totally independent from a particular problem to be solved. It basically involves a definition of coordinate transformations from natural, global to normalized coordinates with the evaluation of the determinant of the Jacobian matrix, the form functions and the form function derivatives.

There is no need to store all this information for each instance of individual elements. Evaluation of this information results for example in the element matrix that will then be used to set up the system matrix. In case this information will need to be used for other purposes again, e.g. computing element forces or visualization of information, the information may be regenerated. This is important as there may be a large number of elements to be used in a particular analysis. For these reasons, element attributes are declared as ***static*** meaning utilization as class attributes that may be overwritten for each instance of that class.

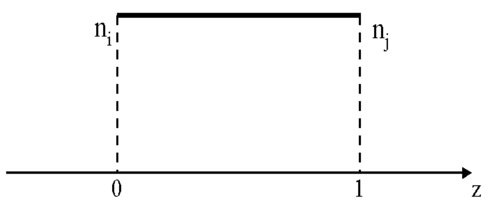
6.1 Element Geometry for 1D Elements with 2 Nodes

The 1-dimensional elements chosen to model the geometry of the problem can be described in terms of the **global coordinate x**. The two nodes of each element are named global node numbers: n_i and n_j. The nodal coordinates are then given for node i by $x_{(i)}$, for node j by $x_{(j)}$.

For the purposes of a general geometric description in the context of a FEM analysis it is more convenient to model the elements in **normalized coordinates z**. Normalized coordinates are the same for all elements.



global coordinate



normalized coordinate

The element chosen is determined by its nodal coordinates and linear interpolation polynomials for coordinate values within the element. Coordinate z is assumed to vary linearly within the element and global coordinate values are determined by geometric interpolation using shape functions for each node:

$$x(z) = \{x_{(0)} \quad x_{(1)}\} \begin{Bmatrix} (1-z) \\ z \end{Bmatrix}$$

$$\mathbf{x} = \mathbf{X} \quad \mathbf{s}(z)$$

with: **X** nodal coordinate matrix

s(z) shape function vector

The derivative of the shape function vector **s** with respect to the coordinate z is called the **normalized derivative of the shape function vector** and is denoted with $\mathbf{S}_z$.

$$\mathbf{S}_z = \frac{\partial s \ (z)}{\partial z} = \begin{Bmatrix} -1 \\ 1 \end{Bmatrix}$$

The coordinate increment dx is defined by

$$dx = \frac{\partial x}{\partial z}\, dz \qquad \text{with: } \frac{\partial x}{\partial z} = \frac{\partial}{\partial z}\big(\mathbf{X}\,\mathbf{s}(z)\big) = \mathbf{X}\,\frac{\partial \mathbf{s}}{\partial z} = \mathbf{X}\,\mathbf{S}_z$$

$$dx = \mathbf{X}\,\mathbf{S}_z\, dz = \begin{Bmatrix} x_{(0)} & x_{(1)} \end{Bmatrix} \begin{Bmatrix} -1 \\ 1 \end{Bmatrix} dz = (x_{(1)} - x_{(0)})\, dz = L\, dz$$

$$dz = \frac{1}{L} dx = Z_x\, dx$$

This can be used to evaluate the shape function derivative matrix $\mathbf{S}_x$.

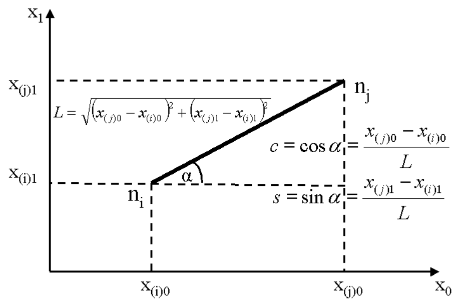
$$\mathbf{S}_x = \mathbf{S}_z * \mathbf{Z}_x = \begin{Bmatrix} -1 \\ 1 \end{Bmatrix} \frac{1}{L}$$

and thus the product $S_x S_x^T$ may be evaluated to $\quad \mathbf{S}_x\,\mathbf{S}_x^T = \frac{1}{L^2}\begin{Bmatrix} \mathbf{1} & \mathbf{-1} \\ \mathbf{-1} & \mathbf{1} \end{Bmatrix}$

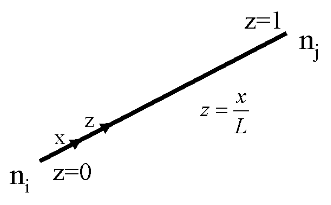
Matrix $\mathbf{S}_x$ is independent of coordinate z and thus integration may be performed analytically.

6.2 Element Geometry for 1D Elements with 2 Nodes in 2D space

The 1-dimensional elements chosen to model the geometry of the problem can be described in terms of the **global coordinates** x_i. The two nodes of each element are named global node numbers: n_i and n_j. The nodal coordinates are then given for node i by $x_{(i)0}$, $x_{(i)1}$, for node j by $x_{(j)0}$, $x_{(j)1}$.
For the purposes of a general geometric description in the context of a FEM analysis it is more convenient to model the elements in **normalized coordinates z**. Normalized coordinates are the same for all elements.


x_1
$x_{(j)1}$
$x_{(i)1}$
$L = \sqrt{(x_{(j)0} - x_{(i)0})^2 + (x_{(j)1} - x_{(i)1})^2}$
n_j
n_i
α
$c = \cos\alpha = \frac{x_{(j)0} - x_{(i)0}}{L}$
$s = \sin\alpha = \frac{x_{(j)1} - x_{(i)1}}{L}$
$x_{(i)0}$
$x_{(j)0}$
x_0

global coordinate



normalized coordinate

The element chosen is determined by its nodal coordinates and linear interpolation polynomials for coordinate values within the element. Coordinate z is assumed to vary linearly within the element and global coordinate values are determined by geometric interpolation using shape functions for each node:

$$\mathbf{x}(z) = \begin{Bmatrix} x_{(i)0} & x_{(i)1} \\ x_{(j)0} & x_{(j)1} \end{Bmatrix} \begin{Bmatrix} 1 - z \\ z \end{Bmatrix} \qquad \text{with:} \quad \mathbf{X} \quad \text{nodal coordinate matrix}$$

$$\mathbf{x} = \mathbf{X} \quad \mathbf{s}(z) \qquad \mathbf{s}(z) \quad \text{shape function vector}$$

The derivative of the shape function vector **s** with respect to the coordinate z is called the **normalized derivative of the shape function vector** and is denoted with $\mathbf{S}_z$.

$$\mathbf{S}_z = \frac{\partial s\ (z)}{\partial z} = \begin{Bmatrix} -1 \\ 1 \end{Bmatrix} \qquad \mathbf{S}_x = \frac{1}{L}\begin{bmatrix} c & 0 \\ s & 0 \\ 0 & c \\ 0 & s \end{bmatrix}\begin{Bmatrix} -1 \\ 1 \end{Bmatrix} = \frac{1}{L}\begin{Bmatrix} -c \\ -s \\ c \\ s \end{Bmatrix}$$

This can be used to evaluate the shape function derivative matrix $\mathbf{S}_x$ and thus the product $\mathbf{S}_x\,\mathbf{S}_x^T$ may be evaluated to:

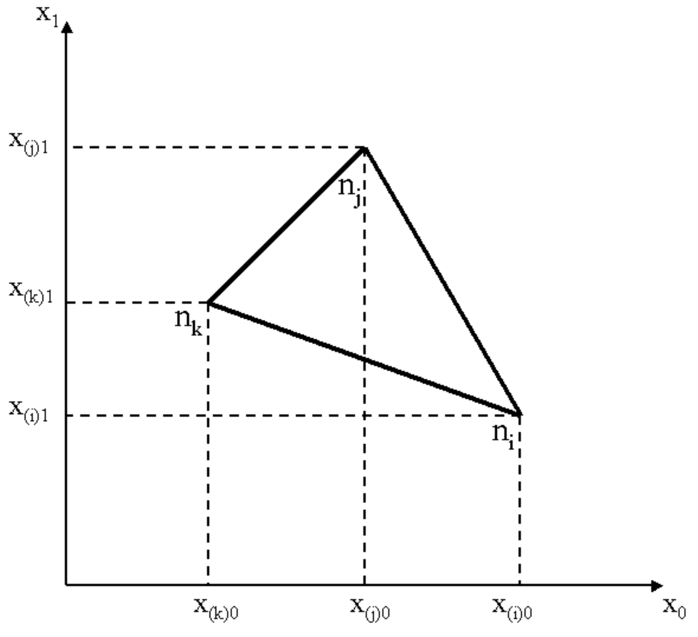
$$\mathbf{S}_x\,\mathbf{S}_x^T = \frac{1}{L^2}\begin{bmatrix} c^2 & cs & -c^2 & -cs \\ cs & s^2 & -cs & -s^2 \\ -c^2 & -cs & c^2 & cs \\ -cs & -s^2 & cs & s^2 \end{bmatrix}$$

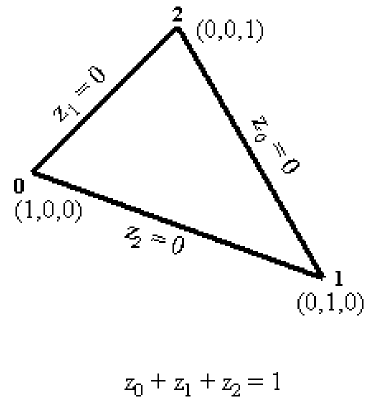
Matrix $\mathbf{S}_x$ is independent of coordinate z and thus integration may be performed analytically.

6.3 Element Geometry for 2D Elements with 3 Nodes

The triangular elements chosen to model the geometry of the problem can be described in terms of the 2-dimensional **global coordinates** x_0 and x_1. The three nodes of each element are ordered in counter-clockwise direction and named global node numbers: n_i, n_j and n_k. The nodal coordinates are then given for node i by ($x_{(i)0}$, $x_{(i)1}$), for node j by ($x_{(j)0}$, $x_{(j)1}$) and for node k by ($x_{(k)0}$, $x_{(k)1}$).

For the purposes of a general geometric description in the context of a FEM analysis it is more convenient to model the triangular elements in **normalized coordinates z_0, z_1 and z_2**. Normalized coordinates are the same for all triangular elements. At node 0 the coordinates are (1,0,0), at node 1 (0,1,0) and at node 2 (0,0,1). The edge between node 0 an 1 has a coordinate value of z_0=0, the edge between node 1 and 2 has coordinate value z_1=0 and between 2 and 0 has z_2=0. Normalization and redundancy of coordinate values require in addition the condition $z_0+z_1+z_2=1$ to be fulfilled anywhere in the element.





global coordinates **normalized triangular coordinates**

The triangular element chosen is determined by its nodal coordinates and linear interpolation polynomials for coordinate values within the element. Coordinates z_0, z_1 and z_2 are assumed to vary linearly within the element and global coordinate values are determined by geometric interpolation using shape functions for each node:

$$\begin{Bmatrix} x_0 \\ x_1 \end{Bmatrix} = \begin{Bmatrix} x_{(0)0} & x_{(1)0} & x_{(2)0} \\ x_{(0)1} & x_{(1)1} & x_{(2)1} \end{Bmatrix} * \begin{Bmatrix} z_0 \\ z_1 \\ z_2 \end{Bmatrix} \qquad \text{with: } \mathbf{X} \quad \text{nodal coordinate matrix}$$

$$\mathbf{x} \quad = \quad \mathbf{X} \quad \mathbf{s}(z) \qquad \mathbf{s}(z) \quad \text{shape function vector}$$

The derivative of the shape function vector **s** with respect to the coordinate z_i is called the i-th **normalized derivative of the shape function vector** and is denoted with $\mathbf{s}_i$. For the linear triangle the derivatives $\mathbf{s}_i$ are unit vectors. In general the derivatives are functions $\mathbf{s}_i(\mathbf{z})$ of the normalized coordinates. They are arranged columnwise in a matrix $\mathbf{S_z}$.

$$\mathbf{s}_i = \frac{\partial \mathrm{s}}{\partial z_i} \qquad \mathrm{s}_\mathbf{0} = \begin{Bmatrix} 1 \\ 0 \\ 0 \end{Bmatrix} \qquad \mathrm{s}_\mathbf{1} = \begin{Bmatrix} 0 \\ 1 \\ 0 \end{Bmatrix} \qquad \mathrm{s}_\mathbf{2} = \begin{Bmatrix} 0 \\ 0 \\ 1 \end{Bmatrix} \qquad \mathbf{S}_z = \begin{Bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{Bmatrix}$$

The coordinate increments dx_i are defined by

$$\begin{Bmatrix} dx_0 \\ dx_1 \end{Bmatrix} = \begin{Bmatrix} \frac{\partial x_0}{\partial z_0} & \frac{\partial x_0}{\partial z_1} & \frac{\partial x_0}{\partial z_2} \\ \frac{\partial x_1}{\partial z_0} & \frac{\partial x_1}{\partial z_1} & \frac{\partial x_1}{\partial z_2} \end{Bmatrix} * \begin{Bmatrix} dz_0 \\ dz_1 \\ dz_2 \end{Bmatrix}$$

with: $\frac{\partial \mathbf{x}}{\partial z_i} = \frac{\partial}{\partial z_i} \left(\mathbf{X}\, \mathbf{s}(z) \right) = \mathbf{X} \frac{\partial \mathrm{s}}{\partial z_i} = \mathbf{X}\, \mathbf{S}_z$

we get: $\begin{Bmatrix} dx_0 \\ dx_1 \end{Bmatrix} = \mathbf{X}\,\mathbf{S}_z * \begin{Bmatrix} dz_0 \\ dz_1 \\ dz_2 \end{Bmatrix} = \begin{Bmatrix} x_{(0)0} & x_{(1)0} & x_{(2)0} \\ x_{(0)1} & x_{(1)1} & x_{(2)1} \end{Bmatrix} * \begin{Bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{Bmatrix} * \begin{Bmatrix} dz_0 \\ dz_1 \\ dz_2 \end{Bmatrix}$

$$\begin{Bmatrix} dx_0 \\ dx_1 \end{Bmatrix} = \begin{Bmatrix} x_{(0)0} & x_{(1)0} & x_{(2)0} \\ x_{(0)1} & x_{(1)1} & x_{(2)1} \end{Bmatrix} * \begin{Bmatrix} dz_0 \\ dz_1 \\ dz_2 \end{Bmatrix}$$

$$d\mathbf{x} \quad = \quad \mathbf{X} \quad * \quad d\mathbf{z}$$

The normalization and redundancy condition $z_0+z_1+z_2=1$ defined above results in the condition $dz_0+dz_1+dz_2=0$. From this linear dependency it follows that $dz_2 = -dz_0 - dz_1$. Replacing dz_2 in the equation above gives the following result:

$$\begin{Bmatrix} dx_0 \\ dx_1 \end{Bmatrix} = \begin{Bmatrix} x_{(0)0} - x_{(2)0} & x_{(1)0} - x_{(2)0} \\ x_{(0)1} - x_{(2)1} & x_{(1)1} - x_{(2)1} \end{Bmatrix} * \begin{Bmatrix} dz_0 \\ dz_1 \end{Bmatrix} = \begin{Bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{Bmatrix} * \begin{Bmatrix} dz_0 \\ dz_1 \end{Bmatrix}$$

$$d\mathbf{x} \quad = \quad \mathbf{X}_{zu} \quad * \quad d\mathbf{z}_u$$

with: index **u** indicating the reduced form

The inverse of matrix $\mathbf{X}_{zu}$ can be evaluated analytically under the assumption that the area of the element is not equal to zero. It will be denoted $\mathbf{Z}_{xu}$.

$$d\mathbf{z}_u = \mathbf{Z}_{xu} * d\mathbf{x}$$

with: $$\mathbf{Z}_{xu} = \frac{1}{c_{00} * c_{11} - c_{01} * c_{10}} \begin{Bmatrix} c_{11} & -c_{10} \\ -c_{01} & c_{00} \end{Bmatrix} = \frac{1}{\det \mathrm{X}_{zu}} * \begin{Bmatrix} c_{11} & -c_{01} \\ -c_{10} & c_{00} \end{Bmatrix}$$

Utilizing again condition $dz_2 = -dz_0 - dz_1$ we get:

$$\begin{Bmatrix} dz_0 \\ dz_1 \\ dz_2 \end{Bmatrix} = \frac{1}{\det \mathrm{X}_{zu}} \begin{Bmatrix} c_{11} & -c_{01} \\ -c_{10} & c_{00} \\ c_{10} - c_{11} & c_{01} - c_{00} \end{Bmatrix} * \begin{Bmatrix} dx_0 \\ dx_1 \end{Bmatrix}$$

$$d\mathbf{z} \quad = \quad \mathbf{Z}_x \quad * \quad d\mathbf{x}$$

The derivative of the shape function vector **s** with respect to the global coordinate x_i is given by

$$\frac{\partial \mathbf{s}}{\partial \mathrm{x}_i} = \frac{\partial \mathbf{s}}{\partial \mathrm{z}_0}\frac{\partial \mathrm{z}_0}{\partial \mathrm{x}_i} + \frac{\partial \mathbf{s}}{\partial \mathrm{z}_1}\frac{\partial \mathrm{z}_1}{\partial \mathrm{x}_i} + \frac{\partial \mathbf{s}}{\partial \mathrm{z}_2}\frac{\partial \mathrm{z}_2}{\partial \mathrm{x}_i}$$

The derivative of **s** with respect to x_i is arranged column-wise in matrix $\mathbf{S}_x$.

$$\begin{Bmatrix} \frac{\partial \mathrm{z}_0}{\partial \mathrm{x}_0} & \frac{\partial \mathrm{z}_0}{\partial \mathrm{x}_1} \\ \frac{\partial \mathrm{z}_1}{\partial \mathrm{x}_0} & \frac{\partial \mathrm{z}_1}{\partial \mathrm{x}_1} \\ \frac{\partial \mathrm{z}_2}{\partial \mathrm{x}_0} & \frac{\partial \mathrm{z}_2}{\partial \mathrm{x}_1} \end{Bmatrix} = \mathbf{S}_z * \mathbf{Z}_x = \mathbf{Z}_x = \frac{1}{\det \mathrm{X}_{zu}} * \begin{Bmatrix} c_{11} & -c_{01} \\ -c_{10} & c_{00} \\ c_{10} - c_{11} & c_{01} - c_{00} \end{Bmatrix}$$

$$\mathbf{S}_x \quad = \quad \mathbf{Z}_x$$

Matrix $\mathbf{S}_x$ is independent of coordinates **z** and thus integration may be performed analytically.

The **area of a triangular element** can be evaluated from the cross product of the vectors from node 0 to node 2 ($\mathbf{x}_{(0)} - \mathbf{x}_{(2)}$) and from node 1 to node 2 ($\mathbf{x}_{(1)} - \mathbf{x}_{(2)}$).

$$\text{area} = 0.5 * |\, (\mathbf{x}_{(0)} - \mathbf{x}_{(2)}) \times (\mathbf{x}_{(1)} - \mathbf{x}_{(2)}) \,| = 0.5 * \left|\, \begin{pmatrix} x_{(0)0} - x_{(2)0} \\ x_{(0)1} - x_{(2)1} \end{pmatrix} \times \begin{pmatrix} x_{(1)0} - x_{(2)0} \\ x_{1(1)} - x_{(2)1} \end{pmatrix} \,\right|$$

$$= 0.5 * \left|\, \begin{Bmatrix} c_{00} \\ c_{10} \end{Bmatrix} \times \begin{Bmatrix} c_{01} \\ c_{11} \end{Bmatrix} \,\right|$$

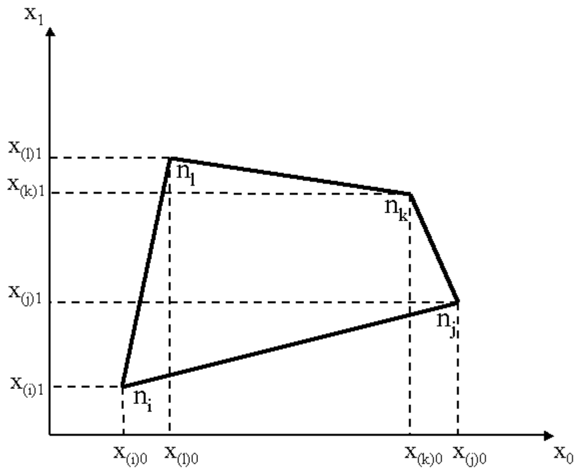
$$= 0.5 * (c_{00} * c_{11} - c_{01} * c_{10}) = 0.5 * \det \mathrm{X}_{zu}$$

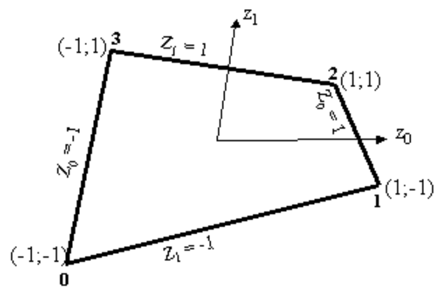
6.4 Element Geometry for 2D Elements with 4 Nodes

The 4-node elements chosen to model the geometry of the problem can be described in terms of the 2-dimensional **global coordinates x_0 and x_1**. The four nodes of each element are ordered in counter-clockwise direction and named global node numbers: n_i, n_j , n_k and n_l.

The nodal coordinates are then given for node i by ($x_{(i)0}$, $x_{(i)1}$), for node j by ($x_{(j)0}$, $x_{(j)1}$), for node k by ($x_{(k)0}$, $x_{(k)1}$) and for node l by ($x_{(l)0}$, $x_{(l)1}$).

For the purposes of a general geometric description in the context of a FEM analysis it is more convenient to model the elements in **normalized coordinates z_0 and z_1**. Normalized coordinates are the same for all 4-node elements. At node 0 the coordinates are (-1,-1), at node 12 (1,-1), at node 2 (1,1) and at node 3 (-1, 1).





global coordinates **normalized coordinates**

The rectangular element chosen is determined by its nodal coordinates and linear interpolation polynomials for coordinate values within the element. Coordinates z_0 and z_1 are assumed to vary linearly within the element and global coordinate values are determined by geometric interpolation using shape functions for each node.

$$\begin{Bmatrix} x_0 \\ x_1 \end{Bmatrix} = \begin{Bmatrix} x_{(0)0} & x_{(1)0} & x_{(2)0} & x_{(3)0} \\ x_{(0)1} & x_{(1)1} & x_{(2)1} & x_{(3)1} \end{Bmatrix} *0{,}25* \begin{Bmatrix} (1-z_0)(1-z_1) \\ (1+z_0)(1-z_1) \\ (1+z_0)(1+z_1) \\ (1-z_0)(1+z_1) \end{Bmatrix}$$

x = **X** **s**(z)

with: **X** nodal coordinate matrix

s(z) shape function vector

The derivatives of the shape function vector **s** with respect to the coordinate z_i is called the i-th **normalized derivative of the shape function vector** and is denoted with $\mathbf{s}_i$. For the linear rectangle the derivatives $\mathbf{s}_i$ are shown below. The derivatives are functions $\mathbf{s}_i$(z) of the normalized coordinates. They are arranged columnwise in a matrix $\mathbf{S_z}$.

$$\mathbf{s}_i = \frac{\partial s(z)}{\partial z_i} \qquad \mathbf{s}_0 = ¼ * \begin{Bmatrix} -(1-z_1) \\ (1-z_1) \\ (1+z_1) \\ -(1+z_1) \end{Bmatrix} \qquad \mathbf{s}_1 = ¼ * \begin{Bmatrix} -(1-z_0) \\ -(1+z_0) \\ (1+z_0) \\ (1-z_0) \end{Bmatrix}$$

$$\mathbf{S}_z = ¼ * \begin{Bmatrix} -(1-z_1) & -(1-z_0) \\ (1-z_1) & -(1+z_0) \\ (1+z_1) & (1+z_0) \\ -(1+z_1) & (1-z_0) \end{Bmatrix}$$

The coordinate increments dx_i are defined by

$$\begin{Bmatrix} dx_0 \\ dx_1 \end{Bmatrix} = \begin{Bmatrix} \frac{\partial x_0}{\partial z_0} & \frac{\partial x_0}{\partial z_1} \\ \frac{\partial x_1}{\partial z_0} & \frac{\partial x_1}{\partial z_1} \end{Bmatrix} * \begin{Bmatrix} dz_0 \\ dz_1 \end{Bmatrix}$$

with: $\frac{\partial \mathbf{x}}{\partial z_i} = \frac{\partial}{\partial z_i} \left(\mathbf{X}\, \mathbf{s}(z) \right) = \mathbf{X} \frac{\partial \mathbf{s}}{\partial z_i} = \mathbf{X}\, \mathbf{S}_z$

$$\begin{Bmatrix} dx_0 \\ dx_1 \end{Bmatrix} = \mathbf{X}\,\mathbf{S}_z * \begin{Bmatrix} dz_0 \\ dz_1 \end{Bmatrix} = \begin{Bmatrix} x_{(0)0} & x_{(1)0} & x_{(2)0} & x_{(3)0} \\ x_{(0)1} & x_{(1)1} & x_{(2)1} & x_{(3)1} \end{Bmatrix} * ¼ \begin{Bmatrix} -(1-z_1) & -(1-z_0) \\ (1-z_1) & -(1+z_0) \\ (1+z_1) & (1+z_0) \\ -(1+z_1) & (1-z_0) \end{Bmatrix} * \begin{Bmatrix} dz_0 \\ dz_1 \end{Bmatrix}$$

$$d\mathbf{x} \quad = \quad \mathbf{X} \quad * \quad \mathbf{S}_z \quad * \quad d\mathbf{z}$$

The product $\mathbf{X}\,\mathbf{S}_z$ may be expressed in the following form:

$$\begin{Bmatrix} dx_0 \\ dx_1 \end{Bmatrix} = \begin{Bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{Bmatrix} * \begin{Bmatrix} dz_0 \\ dz_1 \end{Bmatrix}$$

$$d\mathbf{x} \quad = \quad \mathbf{X}_z \quad * \quad d\mathbf{z}$$

with: $c_{00} = ¼ * (- x_{(0)0} (1-z_1) + x_{(1)0} (1-z_1) + x_{(2)0} (1+z_1) - x_{(3)0} (1+z_1))$

$c_{01} = ¼ * (- x_{(0)0} (1-z_0) - x_{(1)0} (1+z_0) + x_{(2)0} (1+z_0) + x_{(3)0} (1-z_0))$

$c_{10} = ¼ * (- x_{(0)1} (1-z_1) + x_{(1)1} (1-z_1) + x_{(2)1} (1+z_1) - x_{(3)1} (1+z_1))$

$c_{11} = ¼ * (- x_{(0)1} (1-z_0) - x_{(1)1} (1+z_0) + x_{(2)1} (1+z_0) + x_{(3)1} (1-z_0))$

Matrix $\mathbf{X}_z$ is commonly denoted as Jacobian matrix. The determinant of the Jacobian matrix is known in the literature simply as 'the Jacobian' which expresses the relation between coordinate increments in global and normalized coordinates in general:

$dx_0\, dx_1\, dx_2 = \det \mathbf{X}_z\, dz_0\, dz_1\, dz_2$

The inverse of matrix $\mathbf{X}_z$ can be evaluated analytically under the assumption that the determinant of the matrix $\mathbf{X}_z$ is not equal to zero. It will be denoted $\mathbf{Z}_x$.

$d\mathbf{z} = \mathbf{Z}_x * d\mathbf{x}$

with: $\mathbf{Z}_x = \frac{1}{c_{00} * c_{11} - c_{01} * c_{10}} \begin{Bmatrix} c_{11} & -c_{10} \\ -c_{01} & c_{00} \end{Bmatrix} = \frac{1}{\det \mathbf{X}_z} * \begin{Bmatrix} c_{11} & -c_{01} \\ -c_{10} & c_{00} \end{Bmatrix}$

The derivative of the shape function vector **s** with respect to the global coordinate x_i is given by

$$\frac{\partial s}{\partial x_i} = \frac{\partial s}{\partial z_0}\frac{\partial z_0}{\partial x_i} + \frac{\partial s}{\partial z_1}\frac{\partial z_1}{\partial x_i}$$

The derivative of **s** with respect to x_i is arranged column-wise in a matrix $\mathbf{S}_x$.

$$\left\{\frac{\partial s}{\partial x_0} \quad \frac{\partial s}{\partial x_1}\right\} = \mathbf{S}_z * \mathbf{Z}_x = \tfrac{1}{4} * \begin{Bmatrix} -(1-z_1) & -(1-z_0) \\ (1-z_1) & -(1+z_0) \\ (1+z_1) & (1+z_0) \\ -(1+z_1) & (1-z_0) \end{Bmatrix} \frac{1}{\det X_z} * \begin{Bmatrix} c_{11} & -c_{01} \\ -c_{10} & c_{00} \end{Bmatrix}$$

$$\mathbf{S}_x \quad = \quad \mathbf{S}_z \quad \mathbf{Z}_x$$

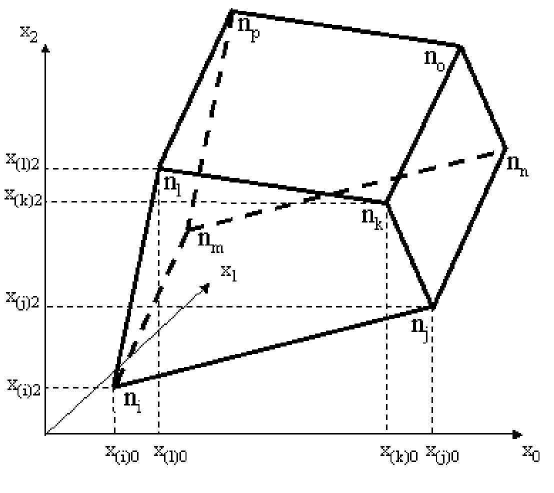
$$\mathbf{S}_x = \frac{0{,}25}{\det X_z} \begin{Bmatrix} -c_{11}(1-z_1)+c_{10}(1-z_0) & c_{01}(1-z_1)-c_{00}(1-z_0) \\ c_{11}(1-z_1)+c_{10}(1+z_0) & -c_{01}(1-z_1)-c_{00}(1+z_0) \\ c_{11}(1+z_1)-c_{10}(1+z_0) & -c_{01}(1+z_1)+c_{00}(1+z_0) \\ -c_{11}(1+z_1)-c_{10}(1-z_0) & c_{01}(1+z_1)+c_{00}(1-z_0) \end{Bmatrix}$$

Matrix $\mathbf{S}_x$ depends on coordinates z_0 and z_1 and thus integration may be performed numerically.

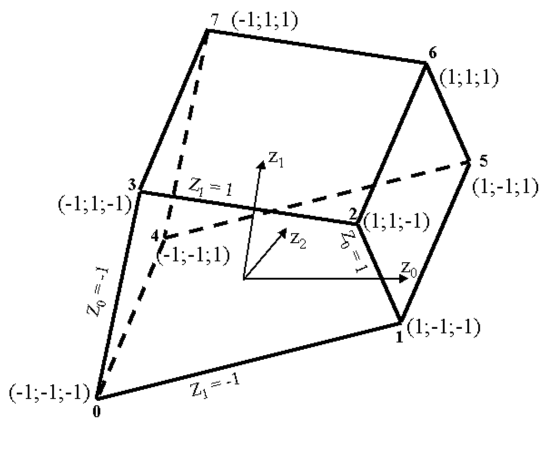
6.5 Element Geometry for 3D Elements with 8 Nodes

The 8-node elements chosen to model the geometry of the problem can be described in terms of the 3-dimensional **global coordinates x_0, x_1 and x_2**. The eight nodes of each element are ordered in counter-clockwise direction and named global node numbers: n_i, n_j , n_k, n_l, n_m, n_n, n_o and n_p. The nodal coordinates are then given for any node i by ($x_{(i)0}$, $x_{(i)1}$, $x_{(i)2}$).

For the purposes of a general geometric description in the context of a FEM analysis it is more convenient to model the elements in **normalized coordinates z_0, z_1 and z_2**. Normalized coordinates are the same for all 8-node elements. At node 0 the coordinates are (-.5,-.5, -.5), at node 1 (.5,-.5, -.5), etc..



global coordinates



normalized coordinates

The rectangular element chosen is determined by its nodal coordinates and linear interpolation polynomials for coordinate values within the element. Coordinates z_0, z_1 and z_2 are assumed to vary linearly within the element and global coordinate values are determined by geometric interpolation using shape functions for each node.

$$\begin{Bmatrix} x_0 \\ x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} x_{(0)0} & x_{(1)0} & x_{(2)0} & x_{(3)0} & x_{(4)0} & x_{(5)0} & x_{(6)0} & x_{(7)0} \\ x_{(0)1} & x_{(1)1} & x_{(2)1} & x_{(3)1} & x_{(4)1} & x_{(5)1} & x_{(6)1} & x_{(7)1} \\ x_{(0)2} & x_{(1)2} & x_{(2)2} & x_{(3)2} & x_{(4)2} & x_{(5)2} & x_{(6)2} & x_{(7)2} \end{Bmatrix} * \frac{1}{8} * \begin{Bmatrix} (1-z_0)(1-z_1)(1-z_2) \\ (1+z_0)(1-z_1)(1-z_2) \\ (1+z_0)(1+z_1)(1-z_2) \\ (1-z_0)(1+z_1)(1-z_2) \\ (1-z_0)(1-z_1)(1+z_2) \\ (1+z_0)(1-z_1)(1+z_2) \\ (1+z_0)(1+z_1)(1+z_2) \\ (1-z_0)(1+z_1)(1+z_2) \end{Bmatrix}$$

x = **X** **s**(z)

with: **X** nodal coordinate matrix

s(z) shape function vector

The derivatives of the shape function vector **s** with respect to the coordinate z_i is called the i-th **normalized derivative of the shape function vector** and is denoted with $\mathbf{s}_i$. For the linear rectangle the derivatives $\mathbf{s}_i$ are shown below. The derivatives are functions $\mathbf{s}_i(z)$ of the normalized coordinates. They are arranged columnwise in a matrix $\mathbf{S_z}$.

$$\mathbf{S}_z = \frac{1}{8} * \begin{Bmatrix} -(1-z_1)(1-z_2) & -(1-z_0)(1-z_2) & -(1-z_0)(1-z_1) \\ (1-z_1)(1-z_2) & -(1+z_0)(1-z_2) & -(1+z_0)(1-z_1) \\ (1+z_1)(1-z_2) & (1+z_0)(1-z_2) & -(1+z_0)(1+z_1) \\ -(1+z_1)(1-z_2) & (1-z_0)(1-z_2) & -(1-z_0)(1+z_1) \\ -(1-z_1)(1+z_2) & -(1-z_0)(1+z_2) & (1-z_0)(1-z_1) \\ (1-z_1)(1+z_2) & -(1+z_0)(1+z_2) & (1+z_0)(1-z_1) \\ (1+z_1)(1+z_2) & (1+z_0)(1+z_2) & (1+z_0)(1+z_1) \\ -(1+z_1)(1+z_2) & (1-z_0)(1+z_2) & (1-z_0)(1+z_1) \end{Bmatrix}$$

The coordinate increments dx_i are defined by

$$\begin{Bmatrix} dx_0 \\ dx_1 \\ dx_2 \end{Bmatrix} = \begin{Bmatrix} \frac{\partial x_0}{\partial z_0} & \frac{\partial x_0}{\partial z_1} & \frac{\partial x_0}{\partial z_2} \\ \frac{\partial x_1}{\partial z_0} & \frac{\partial x_1}{\partial z_1} & \frac{\partial x_1}{\partial z_2} \\ \frac{\partial x_2}{\partial z_0} & \frac{\partial x_2}{\partial z_1} & \frac{\partial x_2}{\partial z_2} \end{Bmatrix} * \begin{Bmatrix} dz_0 \\ dz_1 \\ dz_2 \end{Bmatrix}$$

with: $\frac{\partial \mathbf{x}}{\partial z_i} = \frac{\partial}{\partial z_i} \left(\mathbf{X}\, \mathbf{s}(z) \right) = \mathbf{X} \frac{\partial \mathbf{s}}{\partial z_i} = \mathbf{X}\, \mathbf{S}_z$

$$\begin{Bmatrix} dx_0 \\ dx_1 \\ dx_2 \end{Bmatrix} = \mathbf{X}\,\mathbf{S}_z \; * \begin{Bmatrix} dz_0 \\ dz_1 \\ dz_2 \end{Bmatrix}$$

The product $\mathbf{X}\,\mathbf{S}_z$ may be expressed in the following form:

$$\begin{Bmatrix} dx_0 \\ dx_1 \\ dx_2 \end{Bmatrix} = \begin{Bmatrix} c_{00} & c_{01} & c_{02} \\ c_{10} & c_{11} & c_{12} \\ c_{20} & c_{21} & c_{22} \end{Bmatrix} * \begin{Bmatrix} dz_0 \\ dz_1 \\ dz_2 \end{Bmatrix}$$

$$d\mathbf{x} \quad = \quad \mathbf{X}_z \quad * \quad d\mathbf{z}$$

Matrix $\mathbf{X}_z$ is commonly denoted as Jacobian matrix. The determinant of the Jacobian matrix is known in the literature simply as ‘the Jacobian’ which expresses the relation between coordinate increments in global and normalized coordinates in general:

$$dx_0\, dx_1\, dx_2 = \det \mathbf{X}_z\, dz_0\, dz_1\, dz_2$$

The inverse of matrix $\mathbf{X}_z$ can be evaluated analytically under the assumption that the determinant of the matrix $\mathbf{X}_z$ is not equal to zero. It will be denoted $\mathbf{Z}_x$.

$$d\mathbf{z} = \mathbf{Z}_x * d\mathbf{x}$$

with: $$\mathbf{Z}_x = \frac{1}{\det \mathbf{X}_z} * \begin{Bmatrix} (c_{11}c_{22} - c_{12}c_{21}) & (c_{02}c_{21} - c_{01}c_{22}) & (c_{01}c_{12} - c_{02}c_{11}) \\ (c_{12}c_{20} - c_{10}c_{22}) & (c_{00}c_{22} - c_{02}c_{20}) & (c_{02}c_{10} - c_{00}c_{12}) \\ (c_{10}c_{21} - c_{11}c_{20}) & (c_{01}c_{20} - c_{00}c_{21}) & (c_{00}c_{11} - c_{01}c_{10}) \end{Bmatrix}$$

and: $$\det \mathbf{X}_z = c_{00} * (c_{11} * c_{22} - c_{12} * c_{21}) - c_{01} * (c_{10} * c_{22} - c_{12} * c_{20}) + c_{02} * (c_{10} * c_{21} - c_{11} * c_{20})$$

The derivative of the shape function vector **s** with respect to the global coordinate x_i is given by

$$\frac{\partial \mathbf{s}}{\partial x_i} = \frac{\partial \mathbf{s}}{\partial z_0}\frac{\partial z_0}{\partial x_i} + \frac{\partial \mathbf{s}}{\partial z_1}\frac{\partial z_1}{\partial x_i} + \frac{\partial \mathbf{s}}{\partial z_2}\frac{\partial z_2}{\partial x_i}$$

The derivative of **s** with respect to x_i is arranged column-wise in a matrix $\mathbf{S}_x$.

$$\mathbf{S}_x = \mathbf{S}_z * \mathbf{Z}_x$$

Matrix $\mathbf{S}_x$ depends on coordinates z_0, z_1 and z_2 and thus integration may be performed numerically.

7 General Functionality of FEM Components

General functionality for defining the model of the FEM Framework is combined in ***package framework.model*** with **class *AppObject*** and **class *Model***. Both of these are totally independent from a specific application of an FEM analysis. Correspondingly, functionality of FEM components that are independent from a specific application are collected in a subdirectory for the implementation and another one for the required functionality (interfaces) of FEM components.

Class *AppObject* is contained in *package framework.model*. It is defined to manage administrational aspects of all FEM components for any possible application. It requires to import the interface that all application components must implement. Class *AppObject* is required to implement the interfaces *IAppObject* and *Comparable*.

Each component of the FEM Framework (e.g. node, element, load, support) will be derived from superclass *AppObject*. This class has only a single attribute – the external string identifier, it has a constructor for generating a new *AppObject* with a predefined *id,* a method for returning the identifier of an existing *AppObject* (*getId*) and a method for ordering of application objects (*compareTo*).

```
package framework.model;

import java.io.Serializable;
import framework.model.interfaces.IAppObject;

public class AppObject implements IAppObject, Comparable<Object>, Serializable {
    protected String id;

    // constructor generates a new AppObject with the specified object id
    protected AppObject(String m_id)  {  this.id = m_id; }

    // returns the object Id  as String
    public String getId( )  { return id; }

    // see java.lang.Comparable#compareTo(java.lang.Object)
    public int compareTo(Object object)  {
         String s = ((AppObject) object).id;
         return id.compareTo(s);
    }
}
```

Ordering of objects is defined in the **natural order** of the persistent string identifiers of the application objects ((AppObject) object).id). Any string can be defined as an identifier. The natural order is defined by the Unicode values of the string - this means in numerically and alphabetically ascending order of string characters in their respective position in the string id.

It is therefore to be noted that for example *node1, node2, node3, … , node10* would **naturally** be ordered the following way: *node1, node10, node2, node3,* … Therefore, a 'correct' ordering would be achieved by naming the identifiers *node01, node02, node03, ….., node10*.

Class *Model* provides the basic functionality required for managing an internal data model of a Finite Element Analysis. A *model* object basically combines and organizes a collection of all FEM components - identified by its corresponding string identifiers. Relations between components are supported via a mapping between unique string identifiers and corresponding internal references in the *objectMap*. Access to components by different types - e.g. elements, nodes, etc. – is supported by the *componentSet*. More than a single model object may be instantiated from class *Model* each containing a collection of FEM components relevant for a specific analysis. The status of an analysis may be checked for system solved, eigensolution performed and time step integration completed.

Class *Model* is generally defined by the following functionality:

```
//  CLASS : Model manages a general FEM model
//
//  FEM object ids and references are collected in HashMap "objectMap"
//  FEM components are collected in TreeSet "componentSet"
//
//  Model(String id)
//  getModel()
//  getId()
//  getObject(String objectId)
//  getComponentSet()
//  getSolved (),  getEigen (),  getTimeint()          flags for status of analysis
//  setSolved (),  setEigen (),  setTimeint()
//  containsObjectID(String objectId)
//  add(AppObject component)
//  remove(String identifier)
//  setReferences()
//  iterator()                          componentSet iterator
//  iterator(Class<?> classType)
//  size()                              number of components in componentSet
//  size(Class<?> classType)
//  clearMap()
//  clearSet()
//  print()
```

Class *Model* requires to import some general Java functionality which is mainly associated with general data structures, i.e. container classes like *Map* and *Set* and the corresponding *iterator* objects.

In addition it requires functionality for setting up the necessary relations which is contained in the interface for referenced objects. Finally it requires some utility functionality as for example for setting up filterable classes.

Class *Model* of the FEM Framework is derived from the superclass *AppObject*. It will instantiate a new *objectMap* for the model and declare a *componentSet*. The constructor of a model object will store the *id* as a private attribute using the constructor of the superclass *AppObject* and it will instantiate a new *componentSet* for each new model object.

```
public class Model implements IModel {

    private Map<String, AppObject> objectMap =
                                            new HashMap<String, AppObject>( );
    private TreeSet<AppObject> componentSet;
    private Model m_model;
    private String modelId;
    private Iterator<?> iter;
    boolean solved=false, eigen=false, timeint=false;
    //......Constructor................................................
    public Model(String id) {
        this.modelId = id;;
        componentSet = new TreeSet<AppObject>();
        m_model = this;
    }
```

Class *Model* instantiates a new Java container structure objectMap of type Hash*Map*. The constructor of each model object instantiates a new object of the container structure TreeSet under the name *componentSet*.

A TreeSet implements the Set interface, backed by a TreeMap instance. This class guarantees that the sorted set will be in ascending element order, sorted according to the natural order of the elements (see Comparable), or by the comparator provided at set creation time, depending on which constructor is used. The empty constructor constructs a new, empty set, sorted according to the elements' natural order.

A TreeSet was chosen in order to have an efficient data structure (guaranteed log(n) time cost for the basic operations) and to support a lexicographical ordering of objects according to their identifiers.

Note: The ordering scheme must be observed for string identifiers. For example, identifiers *n1, n11, n2* would be ordered in the order shown, but this is usually not what is desired and correspondingly the identifiers should be named accordingly: *n01, n02, n11*.

The HashMap *objectMap* is used in the FEM Framework for storing the external string identifiers of **all** FEM objects in the *objectMap* and mapping these when starting the application to the corresponding internal references of the FEM objects. For this purpose, each corresponding class of the FEM Framework (e.g. Element, Material, Load, Support) must provide a method *setReferences()* for retrieving the internal reference of its objects from the *objectMap* based upon the corresponding external string identifier.

A HashMap structure is described in the Java documentation as follows:

“A HashMap is a Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key. (The

HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time. This implementation provides constant-time performance for the basic operations (get and put), assuming the hash function disperses the elements properly among the buckets."

In the context of the FEM Framework, the *objectMap* can be regarded as an address book. Originally, the model is set up exclusively by names. Only when the analysis is started, all addresses (internal references) are established based upon the address book.

Class *Model* also provides a static methods for retrieving the current model object (*getModel*) a method for retrieving the internal reference of an application object from the *objectMap (getObject)* via its persistent string identifier, a method for retrieving application objects from the componentSet and six methods for retrieving and storing the status of equation solving, eigensolutions and time integration.

Class *Model* provides a method for checking if an existing application object is already contained in the *objectMap (containsObjectID),* for adding a new component of the FEM Framework to the *componentSet* (*add*) and for storing its internal identifier (reference) along with the corresponding external string identifier into the *objectMap*. It also provides functionality for removing an object from the *componentSet* (*remove*).

```
public boolean containsObjectID(String objectId) {
    return objectMap.containsKey(objectId);
}
public void add(AppObject component)  {
    If (objectMap.containsKey(component.getId()))
        throw new RuntimeException("*** "+component.getId()+ " is not unique! ***");
    objectMap.put(component.getId( ), component);
    componentSet.add(component);
}
public boolean remove(String identifier) {
    AppObject obj = objectMap.remove(identifier);
    return componentSet.remove(obj);
}
```

The management of application objects is supported by method (*setReferences*) of class *Model* that will evaluate internal references of application objects, i.e where to find those objects when needed. This method will iterate on all objects requiring access to other objects by invoking method *setReferences* of those objects. Method *setReferences* will evaluate for a specific object the internal references required for establishing the necessary relations to other application objects based upon their corresponding persistent string identifiers.

Objects requiring access to other, related objects need to implement interface *IReferencedObject*. This is accomplished by any such class (e.g. *AbstractElement*) via implementing the corresponding interface (e.g. *IElement*) that will then be derived from interface *IReferenceObject*.

For example, any FEM *Element* object when being evaluated will require access to its corresponding *Node* and *Material* objects. Therefore, any class implementing FEM *Elements* needs to be derived from abstract class *AbstractElement* which implements interface *IElement* that is derived from interface *IReferencedObject*. On the other hand, class *Node* does not require access to related objects and thus does not need to implement interface *IReferencedObject*.

Interface *IReferencedObject* will require method *setReferences* to be implemented by all corresponding classes (e.g. an FEM *Element* class) that belong to a specific model by setting up internal references to the related objects.

```
public void setReferences() throws FemException {
    iter = iterator(IReferencedObject.class);
    while (iter.hasNext()) {
        IReferencedObject obj = (IReferencedObject) iter.next();
        obj.setReferences(m_model);
    }
}
```

Class Model further provides methods for iterating the complete componentSet, for iterating the *componentSet* with respect to specific class types (*iterator*), for retrieving the number of objects in the complete *componentSet* and for retrieving the number of objects of a specific class type in the *componentSet (size)*. All components of all different types are contained in a single *componentSet* that can be iterated according to specific class filters. The concepts used for these purposes are that of a *ClassFilter* for specific class types and a class *IteratorFilter* (*filterableIterator*) for the *componentSet* both of which are contained in package util.

Two methods are provided for emptying the object map and the component set in case separate models are to be analyzed.

```
public Iterator<?> iterator( )  { return componentSet.iterator( ); }
public Iterator<?> iterator(Class<?> classType)  {
    return ClassFilter.filterableIterator(componentSet.iterator(),
                                                                        new ClassFilter(classType));
}
public int size( )  { return componentSet.size( ); }
public int size(Class<?>  classType)  {
    int size = 0;
    iter = ClassFilter.filterableIterator(componentSet.iterator(),
                                                         new ClassFilter(classType));
    while(iter.hasNext( ))  {
        iter.next( );
        size++;
    }
    return size;
}
```

Three more classes contained in ***package framework.model*** will capture and throw exceptional conditions when accessing FEM data (*AccessException*), analysing the FEM model data (*FEMException*) and for numerical evaluation of data (*AlgebraicException*):

7.1 Implementation of generalized FEM components

The implementation of general Model Components contains one class – Node – that is totally independent of a specific application and five abstract classes – AbstractElement, AbstractMaterial, AbstractNodeLoad, AbstractElementLoad, AbstractSupport – that need to be further refined for their specific utilization in derived classes.

Class *Node* is part of the package framework.model. It needs to import classes for number formatting and an interface for node objects *framework.model.interfaces.INode*.

Class Node contains the definition and the functionality of a generalized Node object in a FEM analysis. It provides a constructor for storing the string id, the nodal coordinates and the number of nodal degrees of freedom of node objects.

```
public class Node extends AppObject implements INode  {
    private int spatialDimension, nodalDegreesOfFreedom, index[ ];
    private double coordinates[ ], nodalDOF[ ];
    private double [ ][ ] variables, derivatives;

    public Node(String id, double[ ] crds, int ndof)  {
        super(id);
        spatialDimension = crds.length;
        setCoordinates(crds);
        this.nodalDegreesOfFreedom=ndof;
    }
```

Class *Node* also implements a method for defining the system indices of the linear system of equations, i.e. the indices of the system unknowns. For this purpose, all nodes are traversed in natural order and each node number is incremented by its nodal degrees of freedom. The result is stored in vector *index*.

In case, the nodal degrees of freedom is equal to 1, e.g. two-dimensional heat transfer, the number of system indices is equal to the number of nodes and the indices are ordered in natural order. In case, the nodal degrees of freedom is equal to 2, e.g. two-dimensional linear elasticity, the number of system indices is equal to (2 * the number of nodes). In general, the FEM Framework is set up to support different nodes with different numbers of nodal degrees of freedom by querying and incrementing the respective number of nodal degrees of freedom for each individual node separately.

```
public void setSystemIndex(int k)  {
    index = new int[nodalDegreesOfFreedom];
    for (int i = 0; i < nodalDegreesOfFreedom; i++)
        index[i] = k++;
    return k;
}
```

Class Node further provides methods for defining and storing the number of nodal degrees of freedom for each individual Node object (*setNumberOfNodalDOF*), for defining the values of predefined states of nodal degrees of freedom, e.g. support with predefined value - temperature, displacement - most often to be 0 (*setNodalDOF*), for defining the nodal time history of field variables (*setNodalHistory*) and for defining the nodal coordinates (*setCoordinates*).

```
public void setNumberOfNodalDOF(int dof)  { nodalDegreesOfFreedom = dof; }

public void setNodalDOF(double[ ] dof)  {
    nodalDOF = new double[nodalDegreesOfFreedom];
    for (int i = 0; i < nodalDegreesOfFreedom; i++)  this.nodalDOF[i] = dof[i];
}

public void setNodalHistory(double[ ][ ] _variables, double[ ][ ] _derivatives) {
    this.variables   = _variables;
    this.derivatives = _derivatives;
}

public void setCoordinates(double[ ] crds)  {
    if(crds.length != spatialDimensions)
        throw new RuntimeException
                    ("Node: number of coordinates not equal to dimension");
    this.coordinates = new double[spatialDimensions];
    for(int i = 0; i < spatialDimensions;i++)  this.coordinates[i] = crds[i];
}
```

Also, methods are defined for retrieving the number of nodal degrees of freedom (*getNumberOfNodalDOF*), the system indices of the nodal degrees of freedom (*getSystemIndex*), the predefined values of nodal states (*getNodalDOF*), the nodal coordinates (*getCoordinates*) and the nodal time history (*getNodalHistory)*.

```
public int getNumberOfNodalDOF()  {  return nodalDegreesOfFreedom;  }
public int[ ] getSystemIndex( )  {  return index;  }
public double[ ] getNodalDOF() throws AccessException {
    if (nodalDOF == null)
            throw new AccessException ("results must be analyzed first");
    return nodalDOF;
}
public double[ ] getCoordinates( )  {
    double[ ] _crds = new double[spatialDimensions];
    System.arraycopy(coordinates, 0, _crds, 0, spatialDimensions);
    return _crds;
}
public double[ ][ ] getNodalVariables() { return variables;  }
public double[ ][ ] getNodalDerivatives() { return derivatives;  }
```

Finally, methods are defined for output of nodal coordinates and nodal field variables

```
public void printNodalCoordinates() {
    if (spatialDimension == 1)
        System.out.printf(Locale.US,"%1$6s %2$10.2f \n", id, coordinates[0]);
    else if (spatialDimension == 2)
        System.out.printf(Locale.US,"%1$6s %2$10.2f %3$10.2f \n", id,
                                             coordinates[0], coordinates[1]);
    else if (spatialDimension == 3)
        System.out.printf(Locale.US,"%1$6s %2$10.2f %3$10.2f %4$10.2f \n",
                      id, coordinates[0], coordinates[1], coordinates[2]);
}
public void printNodalFieldVariables() {
    System.out.printf( "%1$6s", id);
    for (int i = 0; i < nodalDOF.length; i++)
        System.out.printf(Locale.US,"%1$ 10.4f", nodalDOF[i]);
    System.out.println();
}
```

It is to be noted that class *Node* is derived from superclass *AppObject* and thus further provides functionality defined in that context. This applies to a method for retrieving the id of a Node object (*getId*) and for a method for ordering Nodes in the natural order defined above (*compareTo*).

Abstract classes provide for the definition and functionality of other FEM components (element, material, loads and supports, time integration) that are separated into an application independent part implemented by *abstract classes* and into an application specific part that is implemented in separate classes that extend the abstract classes.

Class *AbstractElement* combines functionality required for administrational aspects of all elements, e.g. persistent identifiers and establishing references. It is part of the package *framework.model.abstractclasses*. It requires to import *model* and *application object* functionality of the Framework and to import interfaces defined for the FEM components *element*, *material* and *node*.

Abstract class element is derived from class *AppObject* and needs to implement the element interface. It has attributes for node and material identifiers, for node and material internal references and for the number of nodes per element and the number of degrees of freedom per element. The constructor of *AbstractElement* will pass the id to the superclass *AppObject*, it will evaluate the number of nodes per element based upon the number of nodal identifiers given in the persistent storage of the model for each element. It will declare the node vector and set the nodal identifiers, the material identifier and the number of degrees of freedom of an element.

```
public abstract class AbstractElement extends AppObject implements IElement  {
    protected String elementId;                         // element identifier
    protected String nodeId[ ];                         // node identifiers
    protected INode node[ ];                            // node references
    protected String materialId;                        // material identifier
    protected IMaterial material;                       // material reference
    protected String crossSectionId;                    // cross section identifier
    protected ICrossSection crossSection;               // cross section reference
    private int nodesPerElement,  elementDOF;

    public AbstractElement(String id, String[ ] eNodes, String m_materialId,
                                                                    int nDOF) {
        super(id);
        this.elementId=id;
        nodesPerElement = eNodes.length;
        node = new INode[nodesPerElement];
        this.nodeId=eNodes;
        this.materialId = m_materialId;
        this.crossSectionId = null;
        this.elementDOF = nDOF;
    }
    public AbstractElement(String id, String[ ] eNodes, String m_crossSectionId,
                                                String m_materialId, int nDOF)  {
        super(id);
        this.elementId=id;
        nodesPerElement = eNodes.length;
        node = new INode[nodesPerElement];
        this.nodeId=eNodes;
        this.materialId = m_materialId;
        this.crossSectionId = m_crossSectionId;
        this.elementDOF = nDOF;
    }
```

Class *AbstractElement* provides methods for retrieving information about an element. There is a method for retrieving the node identifiers (*getNodeId*) of all nodes that belong to a specific element. Another method will retrieve the internal references of all nodes (*getNodes*) that belong to a specific element and yet another one will retrieve the material identifier of the material that belongs to a specific element. The number of degrees of freedom for an element is retrieved by method *getElementDOF*.

Also, there is a method for retrieving the system indices of all element unknowns (*getSystemIndices*). The number of system indices is defined by the number of nodes per element multiplied by the number of element degrees of freedom. The contents is stored in vector *indices*.

```
public String getElementId() { return elementId; }
public String[ ] getNodeId( ) { return nodeId;  }
public INode[ ] getNodes( )  { return node;  }
public String getMaterialId( )  { return materialId;  }
public String getCrossSectionId( )  { return crossSectionId;  }
public int getElementDOF( )  { return elementDOF;  }
public int getNodesPerElement() {return nodesPerElement; }
public int[ ] getSystemIndices( )  {
    int[ ] indices = new int[nodesPerElement * elementDOF];
    int counter = 0;
    for (int i = 0; i < nodesPerElement; i++)  {
        for (int j = 0; j < elementDOF; j++)
            indices[counter++] = getNodes( )[i].getSystemIndex( )[j];
    }
    return indices;
}
```

AbstractElement provides a method that will store new node identifiers (*setNodeId*) and check if as many node identifiers are provided as needed by the element. Another method will store a material identifier for the element material. Since *AbstractElement* implements interface *IElement*, it is also required to implement method *setReferences* for establishing relations to the associated nodes and material.

```
        public void setNodeId(String[ ] nodeId)  {
            if(this.node.length != nodeId.length)
                throw new RuntimeException("*** Abstract Element: Element should" +
                            "have " + nodesPerElement + " nodes");
            this.nodeId = nodeId;
        }
        public void setMaterialId(String value)  {  materialId = value;  }
        public void setCrossSectionId(String value) { crossSectionId = value; }

        public void setReferences(Model m_model) throws FemException {
            // establish reference to each node of the element
            material = (IMaterial) m_model.getObject(materialId);
            if (crossSectionId != null) crossSection =
                                        (ICrossSection) m_model.getObject(crossSectionId);

            for (int i = 0; i < nodesPerElement; i++) {
                node[i] = (INode) m_model.getModel().getObject(nodeId[i]);
                if (node[i] == null) throw new FemException("*** Abstract Element: Node "
                            + nodeId[i] + " of Element " + getId() + " is not defined ***");
            }
            // establish reference to the element material
            if (material == null) throw new FemException("** Abstract Element: Material "
                            + materialId + " of Element " + getId() + " is not defined ***");
    }   }
```

Abstract element classes are derived from class *AbstractElement* and define the geometries for the corresponding elements corresponding to chapter 1.7.

AbstractLinear2Node2D defines static attributes for element length, angle, sine, cosine, shape function vector and rotation matrix. Methods are provided for computing the element geometry, length and shape function derivative vector Sx, for returning the element length and angle and an abstract method for computing the element end forces.

AbstractLinear3Node2D defines static attributes for computing precision, element determinant, shape function derivative matrix Sx and matrix Xzu. Methods are provided for computing the element geometry and the shape function derivative matrix Sx, for returning the element determinant and matrix Xzu and for specifying nodal identifiers.

AbstractLinear4Node2D defines static attributes for computing precision, element determinant, shape function derivative matrix Sx and matrix Xz. Methods are provided for computing the element geometry (determinant), shape function vector S and shape function derivative matrix Sx and for specifying nodal identifiers.

AbstractLinear8Node3D defines static attributes for computing precision, element determinant, shape function derivative matrices Sx and Sz and matrix Xz. Methods are provided for computing the element geometry (determinant) and shape function derivative matrix Sx.

Class *AbstractMaterial* is part of the package *framework.model.abstractclasses*. It requires importing general Java functionality for data structure *Map,* model functionality of the Framework for *AppObject* and an interface defined for the FEM component material.

Class *AbstractMaterial* consists of a constructor for storing the id and of two methods for writing and reading a string for the material property description.

```
public abstract class AbstractMaterial extends AppObject implements IMaterial  {
    protected Map<String, Double> properties = new HashMap<String, Double> ( );

    public AbstractMaterial(String id)  { super(id); }

    // ....set()........................................................
    public void set(String property, double value)  {
         properties.put(property, new Double(value));
    }

    // ....get()........................................................
    public double get(String property)  {
         if(!properties.containsKey(property))  throw new RuntimeException
                         ("Material "+getId()+" does not contain property = "+property);
         return properties.get(property).doubleValue();
    }
}
```

Class *AbstractNodeLoad* is contained in package *framework.model.abstractclasses*. It requires to import model functionality of classes *Model* and *AppObject* with a corresponding exception *FEMException* and it further requires to import the interfaces for nodes and node loads.

Class *AbstractNodeLoad* consists of a constructor for storing the id and for setting the corresponding node id. It provides the method *setReferences()* discussed before and methods for retrieving the node id, the node object and the load vector.

```
public class AbstractNodeLoad extends AppObject implements ILoad  {

    protected int nodalDegreesOfFreedom;  // number of nodal degrees of freedom
    protected String nodeId;              // the load is applied at this node
    protected INode node;                 // reference of the node object
    protected double load[ ];             // loadvector
    protected Model model;

    public AbstractNodeLoad(String id, String nodeId)  {
         super(id);
         this.nodeId = m_nodeId;
    }
```

Since class *AbstractNodeLoad* implements interface *INodeLoad* is also needs to implement method *setReferences* which establishes relations to the corresponding node for the node load.

Three methods are provided for retrieving essential information for a specific node load. One method will return the node identifier of the corresponding node for the node load (*getNodeId*), another one return the internal reference of that node (*getNode*) and a final one will return the corresponding load vector for the node load (*getLoadVector*).

```
    public void setReferences( Model m_model) throws FemException  {
        this.model = m_model;
        node = (INode) Model.getModel().getObject(nodeId);
        if (node == null)
            throw new FemException ("*** Node " + nodeId + " of node load " +
                                                   getId() + " is not defined ***");
        nodalDegreesOfFreedom = node.getNumberOfNodalDOF( );
    }
    public String getNodeId( ) {  return nodeId; }
    public INode getNode( ) {  return node; }
    public int[ ] getSystemIndex() { return node.getSystemIndex(); }
    public double[ ] computeLoadVector() { return load; }
}
```

Class *AbstractLineLoad* is contained in package *framework.model.abstractclasses*. It requires to import model functionality of classes *Model* and *AppObject* with a corresponding exception *FEMException* and it further requires to import the interfaces for nodes and line loads.

Class AbstractLineLoad defines attributes for the start node and end node identifiers and node objects. It further defines attributes for the load definition and the load vector. It finally determines the number of the nodal degrees of freedom.

A constructor is provided for storing the id and for setting the corresponding identifiers for the start node and end node..

Since class AbstractLineLoad implements interface ILineLoad, it also needs to implement method setReferences which establishes relations to the corresponding nodes for the line load. Finally, it retrieves the number of degrees of freedom for the start and end node of the lineload.

Six methods are provided for retrieving essential information for a specific line load. Two methods will return the node identifiers of the corresponding start and end node of the line load (getStartNodeId, getEndNodeId), two more methods will return the internal reference of the nodes (getStartNode, getEndNode) and another one will return the corresponding load vector for the node load (getLoadVector).

The vital method of class *LineLoad* will compute the load vector for a given line load.

Class *AbstractElementLoad* is contained in package *framework.model.abstractclasses*. It requires to import model functionality of classes *Model* and *AppObject* with a corresponding exception *FEMException* and it further requires to import the interfaces for elements and element loads.

Class *AbstractElementLoad* extends class *AppObject* and implements interface *IElementLoad*. It defines attributes for the element identifier, the internal element reference and the load intensity vector. It defines of a constructor for storing the id and for setting the corresponding element id.

AbstractElementLoad provides a method for storing the element identifier and since it implements interface IElementLoad it also must implement method setReferences( ) for establishing relations to the corresponding element for the element load.

Finally, class AbstractElementLoad provides functionality for retrieving essential information associated with the element load. Method getElementId will return the identifier of an associated element, method getElement will return the internal reference of that element, method getSystemIndex will return the system indices associated with that element and method getIntensity will return the vector of element loads.

Class ***AbstractSupport*** is contained in package *framework.model.abstractclasses*. It needs to import model functionality of classes *Model* and *AppObject* with a corresponding exception *FEMException* and it needs to import the interfaces for nodes and supports.

Class *AbstractSupport* consists of a constructor for storing the id and for setting the corresponding node id, for retrieving the node object and for setting the reactions, prescribed state and the restraints.

```
public class AbstractSupport extends AppObject  {
    protected String nodeId;                  // identifier of support node
    protected INode node;                     // reference of support node
    protected int nodalDegreesOfFreedom;  // number of nodal degrees of freedom
    protected Model model;


    public AbstractSupport(String id, String m_nodeId)  {
         super(id);
         this.nodeId = m_nodeId;
    }
```

Class *AbstractSupport* provides methods for establishing references to an associated node, for defining reaction values, for retrieving an associated node and nodal identifier, for retrieving the reaction value, the boolean restraint value and the prescribed support value. Finally, a method is provided for printing support information.

```
    public void setReferences(Model m_model) throws FemException {
        this.model = m_model;
        node = (INode) Model.getModel().getObject(nodeId);
        if (node == null)
            throw new FemException("*** Node " + nodeId + " of support "
                                                  + getId() + " is not defined ***");
    }

    public int getNodalDegreesOfFreedom() { return nodalDegreesOfFreedom; }
    public INode getNode() { return node; }
    public String getNodeId() { return nodeId; }
    public String toString() { return "Support: " + getId() + ", node=" + nodeId; }
}
```

Class ***AbstractTimeInt*** is contained in package *framework.model.abstractclasses*. It needs to import model functionality of classes *AppObject* and it needs to import the interface for time integration objects. Time integration objects serve for storing all information required for a specific time integration analysis.

Class *AbstractTimeInt1st* and *TimeInt2nd* are the final classes in this context. Both are derived from class *AppObject* and implement interfaces *ITimeInt1st* and *ITimeInt2nd* respectively. They consist of a constructor for storing the object id in the model and for retrieving the object id of a specific time integration object.

8 Required Functionality for general Model Components

Functionality that is required to be implemented by classes in the FEM Framework is combined in package *framework.model.interfaces* under the concept of Java interfaces.

In Java an interface is defined by the keyword *interface* given instead of the keyword *class*. An interface allows to establish the “form“ of a class by specifying the method names, argument lists and return types but no implementations. A class implementing an interface must strictly follow this predefined “form” by implementing all functionality exactly as specified. In the context of the FEM Framework interfaces are used to determine functionality that is required for the correct functioning of applications within this framework and for combing functionality that is common to all components of a specific type (i.e. loads). Also, functionality that is common to a specific application (i.e. heat2d, elasticity2d) is combined under this concept.

Interfaces are sometimes described as a “contract” that must be fulfilled between separate interacting software components. In terms of FEM this means, for example, the “contract” that each separate element definition must fulfill in order to be used in the context of the framework. Adding new elements separate from the original implementation at a later time by others will thus have to strictly observe that “contract”.

The FEM Framework specifies interfaces *IAppObject, IElement, IReferencedObject, INode, IMaterial, ISupport, ILoad, INodeLoad, ILineLoad, IElementLoad* and *ITimeInt*. All interfaces are contained in *package framework.model.interfaces*.

Interface IAppObject only defines a single method *getId* which is required for returning the string identifier for any application object (FEM component).

Interface IElement defines a number of methods for retrieving node references and system indices of node objects. The final method defines the computation of the element matrix.

```
package framework.model.interfaces;

import framework.model.FemException;
import framework.model.AlgebraicException;

public interface IElement extends IReferencedObject {
    public INode[ ] getNodes();
    public int[ ] getSystemIndices();
    public double[ ][ ] computeMatrix() throws FemException, AlgebraicException;
    public double[ ]    computeDiagonalMatrix()
                            throws FemException, AlgebraicException;
}
```

Interface IReferencedObject defines method *setReferences* that is required by all elements that need to establish internal references to other elements based upon their persistent identifiers.

Interface INode defines methods for specifying system index, nodal degrees of freedom, coordinates and number of nodal degrees of freedom for a node object.

It defines methods for retrieving system indices, nodal degrees of freedom, coodinates and number of nodal degrees of freedom for a node object.

```
package framework.model.interfaces;

public interface INode extends IAppObject{
    public int  setSystemIndex(int k);
    public void setNodalDOF(double[ ] dof);
    public void setCoordinates(double[ ] crds);
    public void setNodalHistory(double[ ][ ] variables, double[ ][ ] derivatives);

    public int[ ] getSystemIndex();
    public double[ ] getNodalDOF();
    public double[ ] getCoordinates();
    public int getNumberOfNodalDOF();
}
```

Interface IMaterial defines a method for retrieving the string identifier of that material.

```
package framework.model.interfaces;

public interface IMaterial extends IAppObject {
    public double get(String property);
}
```

Interface ISupport defines a method for specifying the nodal reactions. It further defines methods for retrieving internal references, string identifiers, restraint conditions and prescribed values for support object.

```
package framework.model.interfaces;

public interface ISupport extends IReferencedObject {
    public void setReactions(double[ ] support);
    public INode getNode();
    public boolean[ ] getRestraint();
    public double[ ] getPrescribed();
    public String getNodeId();
}
```

Interface ILoad combines the funcionality that is required by any system load. It defines methods for computing the element load vector and for retrieving the system indices of the load.

```
package framework.model.interfaces;

public interface ILoad extends IReferencedObject {
    public double[ ] computeLoadVector();
    public int[ ] getSystemIndex();
}
```

Interfaces ITimeInt1st and ***ITimeInt2nd*** define methods for retrieving maximum time, time step, the time integration parameters, the field variables (temperatures, deformations), derivatives, initial values and the forcing function. They also provide methods for setting integration parameters, field variables, derivatives and force function.

```
public interface ITimeInt1st {
    public double        getDt();
    public double        getTmax();
    public double        getParameter1();
    public double[ ][ ]  getFieldVariables();
    public double[]      getInitial();
    public double[ ][ ]  getForceFunction();
    public void          setParameters(double dt, double tmax, double parameter1);
    public void          setFieldVariables(double[][] fieldVariables);
    public void          setForceFunction(double[][] F);
}
```

```
public interface ITimeInt2nd {
    public double        getTmax();
    public double        getDt();
    public int           getMethod();
    public double        getParameter1();
    public double        getParameter2();
    public double[ ]     getDamping();
    public double[ ][ ]  getFieldVariables();
    public double[ ][ ]  getFieldDerivatives();
    public double[]      getInitialDeformation();
    public double[]      getInitialVelocity();
    public double[ ][ ]  getForceFunction();
    public void          setParameters(double dt, double tmax, int method, double
                                        parameter1, double parameter2);
    public void          setFieldVariables(double[][]  fieldVariables);
    public void          setFieldDerivatives(double[][] fieldDerivatives);
    public void          setForceFunction(double[][] F);
}
```

9 System Equations

Model information stored in the *componentSet* and in the *objectMap* is the basis for setting up the system equations.

The system equations are assembled from the individual contributions of each element matrix which will be added to the system matrix according to the system indices for each element. Correspondingly the right-hand-side vector (RHS, “load” vector) will be assembled from the individual contributions of each element load vector according to the system indices of the corresponding nodes.

For these purposes, ***package framework.model*** provides ***class Equations*** with the corresponding functionality needed in the context of the FEM Framework.

The programming interface of this class is defined by

- a constructor
  Equations (dimension)

- methods for retrieving information of the system equations

double[ ][ ]	getMatrix( )
double[ ][ ]	getDiagonalMatrix( )
double[ ]	getPrimal( )
double[ ]	getDual( )
double[ ]	getVector( )
boolean[ ]	getStatus( )
int[ ]	getProfile( )

- methods for setting the corresponding information

void	setProfile(int index[ ])
void	setStatus(boolean status, int rowIndex, double value)

- methods for allocating the system matrix and vector and adding information to it

void	allocateMatrix( )
void	initializeMatrix( )
public	double getValue(int i, int m)
void	setValue(int i, int m, double value)
void	addValue(int i, int m, double value)
void	addMatrix(int index[ ], double elementmatrix[ ][ ])
void	addDiagonalMatrix(int index[ ], double elementmatrix[ ][ ])
void	addVector(int index[ ], double subvector[ ])

The system equations are assembled in a profile structure of the form

A * u = w + q

where:

A is the system matrix

u is the primal solution vector for the system unknowns (temperatures, deformations, …)

w is the system vector of influences on the model (loads, heat, ...)

q is the dual solution vector for predescribed nodal states (support reactions, ….)

which internally use the additional information of

s the status vector, defining if a node has unknown (false) or predescribed (true) values in the primal solution vector

p the profile vector for storing the "profile" (non-zero values above diagonal) of the system matrix

For the example given above, the corresponding matrix and vectors are as follows:

matrix A							**u**	**w**	**q**	**status**	**profile**
0:	5.0						10.00	0.0	0.00	true	0
1:	-2.5	10.0					10.00	5.0	0.00	true	0
2:	-2.5	5.0					10.00	10.0	0.00	true	1
3:	-2.5	0.0	0.0	10.0			0.00	0.0	0.00	false	0
4:	-5.0,	0.0	-5.0	20.0			0.00	10.0	0.00	false	1
5:	-2.5	0.0	-5.0	15.0			0.00	5.0	0.00	false	2
6:	-2.5	5.0					20.00	0.0	0.00	true	5
7:	-2.5	0.0	0.0	0.0	5.0		0.00	0.0	0.00	false	3
8:	-5.0	0.0	0.0	-2.5	10.0		0.00	0.0	0.00	false	4
9:	-5.0	0.0	0.0	-2.5	10.0		0.00	0.0	0.00	false	5
10:	-2.5	0.0	0.0	-2.5	5.0		20.00	0.0	0.00	true	6

Matrix A is stored in zero-based format with each row allocated according to its profile structure (row number +1 – profile). The primal solution vector stores the predescribed nodal states and the load vector the system loads.

9.1 Implementation of Equation Solving

Equation solving is a separate topic that is beyond the scope of this document. In structural engineering different types of equation solvers have been developed taking into account the specific properties of system equations in FEM analysis.

The properties of system equations in FEM when correctly established result in a matrix that is

- quadratic of size (n*n),
- of rank n,
- symmetric and
- positive definite.
- Also, in FEM analysis the system matrix is rarely fully populated but rather sparsely populated often with a specific structure. The structure is often referred to as a “band structure” or a “profile structure”.

All of these properties are important in order to develop efficient implementations of equation solvers because the complexity of equation solving is of a cubic order **O(n^3).** For many years, it has been an important aspect of research to develop efficient solvers that will be able to solve very large systems of equations (n > 1 million) on computer systems with limited resources regarding storage capacity and processing power. With the enormous development of computer technology this area of research has lost some of its importance but it is still very relevant for very large systems.

For these reasons, many specific implementations were adopted taking into account as much of the specific properties as possible. In the context of this FEM Framework a profile solver with a status vector was used. The implementation used was taken from [4]. This solver can easily be replaced by other solvers with different properties.

For the example given above, the corresponding matrix and vectors after solving the equations are as follows:

matrix A							**u**	**w**	**q**	**status**	**profile**
0:	5.0						10.00	0.0	-6.41	true	0
1:	-2.5	10.0					10.00	5.0	-22.32	true	0
2:	-2.5	5.0					10.00	10.0	-22.99	true	1
3:	-2.5	0.0	0.0	3.16			12.56	0.0	0.0	false	0
4:	-5.0	0.0	-1.58	4.18			13.46	10.0	0.0	false	1
5:	-2.5	0.0	-1.20	3.68			15.19	5.0	0.0	false	2
6:	-2.5	5.0					20.00	0.0	12.01	true	5
7:	-0.79	-0.30	-0.10	0.0	2.07		13.33	0.0	0.0	false	3
8:	-1.20	-0.39	0.0	-1.40	2.54		14.09	0.0	0.0	false	4
9:	-1.36	0.0	-0.06	-1.23	2.58		16.12	0.0	0.0	false	5
10:	-2.5	0.0	0.0	-2.5	5.0		20.00	0.0	9.70	true	6

After decomposition the original contents of **matrix A** is lost and replaced by the values of the decomposed matrix. The system load vector (**w**) is preserved. The solution is stored in the primal (**u**) and dual (**q**) solution vectors.

The implementation of class **ProfileSolverStatus** class is contained in *package **framework.model***. Class *ProfileSolverStatus* provides

- three constructors
  ProfileSolverStatus(matrix, vector, primal, dual, status, profile)
  ProfileSolverStatus(matrix, vector, primal, status, profile)
  ProfileSolverStatus(matrix, status, profile)
- method *setRHS* for redefining only the RHS-vector if the coefficient matrix remains unchanged
- functionality for a triangular decomposition and for solving the equations
  void decompose() throws AlgebraicException
  void solve( )
  void solvePrimal( )
  void solveDual( )
  Method *solve* simply calls *solvePrimal* and *solveDual*. *solvePrimal* will first substitute the prescribed variables in the rows without prescribed primary variables : u = c1 + y1 - A12 * x2, next compute primal variables via forward sweep by rows followed by a backward sweep by rows. *solveDual* will compute the dual variables by substituting the primal variables in the rows with prescribed primal variables.

Another class is contained in ***package framework.model*** for capturing and throwing exceptional conditions in algebraic analysis:

10 General Eigenvalue Problem

Concept: Consider the general eigenvalue problem:

$$\mathbf{A}\mathbf{x}_i = \lambda_i \,\mathbf{B}\mathbf{x}_i$$

where:		
	$\mathbf{A}$	real regular symmetric coefficient matrix of dimension $\mathbf{n}$
	$\mathbf{B}$	real regular symmetric coefficient matrix of dimension $\mathbf{n}$
	λ_i	eigenvalue of the i-th eigenstate
	$\mathbf{x}_i$	eigenvector of the i-th eigenstate $\mathbf{x}_i^{\,T}\mathbf{B}\mathbf{x}_k = \delta_{ik}$

The eigenvalues $(\lambda_1 \ldots \lambda_n)$ are ordered in ascending order. Assume that the eigenvalue λ_1 with the smallest absolute value is to be determined by vector iteration. The start vector $\mathbf{u}_0 \neq \mathbf{0}$ of the iteration is chosen freely. In step s of the iteration, the vector $\mathbf{u}_{s-1}$ is used in the equation above to compute an auxiliary vector $\mathbf{y}_s$ which is then normalised with respect to $\mathbf{B}$ to yield the iterated vector $\mathbf{u}_s$. The iteration converges to the eigenstate $(\lambda_1, \mathbf{x}_1)$.

compute $\mathbf{y}_s$: $\quad \mathbf{A}\,\mathbf{y}_s = \mathbf{B}\,\mathbf{u}_{s-1}$

scale with λ_s: $\quad \mathbf{u}_s \quad = m_s\,\mathbf{y}_s \qquad$ so that $\qquad \mathbf{u}_s^{\,T}\,\mathbf{B}\mathbf{u}_\mathbf{s} = \delta_s \in \{-1, 1\}$

It can be shown that the **rate of convergence to the eigenvector** equals the ratio of the two eigenvalues with the smallest absolute value:

rate of convergence to the eigenvector: $\dfrac{\lambda_1}{\lambda_2}$

It can further be shown that the **rate of convergence to the eigenvalue** equals the square of the ratio of the eigenvalues with the smallest absolute value:

rate of convergence to the eigenvalue: $(\dfrac{\lambda_1}{\lambda_2})^2$

Higher eigenstates are evaluated on the basis of all lower eigenstates determined before. The eigenstate $(\lambda_2, \mathbf{x}_2)$ with the second smallest absolute value of the eigenvalue is to be determined by vector iteration. This is achieved by making the start vector $\mathbf{u}_0$ B-orthogonal to $\mathbf{x}_0$. Next eigenstates are similarly evaluated by making the start vector $\mathbf{u}_0$ B-orthogonal to **all** lower eigenvectors $\mathbf{x}_i$ previously determined.

10.1 Implementation of the Eigensolution

The implementation of class **EigenSolver** class is contained in *package **framework.model***. Class *EigenSolver* provides

- a constructor
  public EigenSolver(double[ ][ ] m_A, double[ ][ ] m_B,
  int m_profile[ ], boolean m_status[ ], int m_numberOfStates)
- methods for reading specific eigenstates
  public double getEigenValue(int index)
  public double[ ] getEigenVector(int index)
- functionality for solving for a specified number of eigenstates
  public void solveEigenstates
  This method will loop over the specified number of eigenstates, set a start vector for the iteration of the next eigenstate, perform the iteration until a specified error bound is met and will store the computed eigenstates.

Algorithm for the Vector Iteration: The iteration rules are not implemented directly but rather the following auxiliary vectors are introduced to avoid repetition of computations:

$$\mathbf{z}_s = \mathbf{B}\,\mathbf{y}_s$$

$$\mathbf{w}_s = m_s\,\mathbf{z}_\mathbf{s}$$

$$\mathbf{x}_s = m_s\,\mathbf{y}_s$$

The method converges to the smallest eigenstate. For each step of iteration, the start vector is B-orthogonalized with respect to ALL previous (smaller) eigenstates, the system $\mathbf{A}\,\mathbf{y}_s = \mathbf{w}_{s-1}$ is solved for $\mathbf{y}_s$, the intermediate vector $\mathbf{z}_s = \mathbf{B}\,\mathbf{y}_s$ is computed, the Rayleigh-quotient is computed with $r = \dfrac{\mathbf{y}_s^T \mathbf{w}_{s-1}}{\mathbf{y}_s \mathbf{z}_s}$

Assuming that matrix $\mathbf{A}$ has already been decomposed into the form $\mathbf{A}=\mathbf{L}\mathbf{L}^T$, the algorithm for the computation of a specified number of eigenstates consists of the following steps:

```
loop over the specified number of eigenstates $(0 \le state < numberOfStates)$
    set start vector $\mathbf{w}_0$
    start iteration for next eigenstate
    do while $|r_s - r_{s-1}| < \varepsilon = 0.001 * r_s$
        increment iteration counter s
        check if number of iterations $> s_{\max}$
        B-orthogonalization of $\mathbf{w}_{s-1}$ with respect to all smaller
        eigenvectors $\mathbf{x}_0 \dots \mathbf{x}_{state-1}$
        solve $\mathbf{A}\mathbf{y}_s = \mathbf{w}_{s-1}$ for $\mathbf{y}_s$
        compute $\mathbf{z}_s = \mathbf{B}\mathbf{y}_s$
        compute $m_s^2 = \frac{1}{\mathbf{y}_s^T \mathbf{z}_s}$
        compute Rayleigh-quotient $r_s = m_s^2 \mathbf{y}_s^T \mathbf{w}_{s-1}$ and the
        difference $(r_s - r_{s-1})$
        compute $\mathbf{w}_s = m_s \mathbf{z}_s$
    compute $\lambda_{state} = r_s$ and $\mathbf{x}_{state} = m_s \mathbf{y}_s$
    store the computed eigenstate $(\lambda_{state}, \mathbf{x}_{state})$
```

In this algorithm, the precision for evaluation of eigenstates is set to an error bound ε where the difference in the Rayleigh-quotient between successive steps of iteration is smaller than 1.0e-3 times the value of the Rayleigh-quotient at the current step.

Start vector w for safety reasons should be B-orthogonalized at the start of each successive step of iteration s due to round-off. This is accomplished by computing the product

$$c_i = \mathbf{w}_{s-1}^T \mathbf{x}_i \qquad \text{for each smaller eigenstate i } (0 <= i < \text{currentState})$$

and by subtracting the contribution of each smaller eigenstate from the start vector:

$$\mathbf{w}_{s-1} = \mathbf{w}_{s-1} - c_i \mathbf{p}_i \qquad \text{where: } \mathbf{p}_i = \mathbf{B}\mathbf{x}_i$$

11 Time Step Integration for 1st Order Differential Equations

problem: $\mathbf{C}\,\dot{\mathbf{T}}(t) + \mathbf{K}\,\mathbf{T}(t) = \mathbf{F}(t)$ and $\mathbf{T}(0) = \mathbf{T}_0$

where: $\mathbf{C}$ is the specific heat matrix

$\dot{\mathbf{T}}$ is the vector of temperature gradients

$\mathbf{K}$ is the thermal conductivity matrix

$\mathbf{T}$ is the temperature vector

$\mathbf{F}$ is forcing function vector of induced temperatures

$\mathbf{T}_0$ is the initial temperature vector at time=0

eigenproblem: $(\mathbf{K} - \beta\mathbf{C}) * \mathbf{q} = 0$

where: β is one of n-eigenvalues for which this equation is true

q is the corresponding eigenvector

solve: det $(\mathbf{K} - \beta\mathbf{C}) = 0$

time integration: $\dot{\mathbf{T}}_k$ chosen as primary dependent variable

$$(\mathbf{C} + \alpha\,\Delta t\,\mathbf{K})\,\dot{\mathbf{T}}_k = \mathbf{F}_k - \mathbf{K}\,(\mathbf{T}_{k-1} + \Delta t\,(1-\alpha)\,\dot{\mathbf{T}}_{k-1}) \tag{11.1}$$

The solution algorithm is defined by the following steps:

```
at time =0: calculate initial temperature gradients $\dot{\mathbf{T}}_0$
    $\mathbf{C}\,\dot{\mathbf{T}}_0 = \mathbf{F}_0 - \mathbf{K}\,\mathbf{T}_0$
evaluate constant coefficient matrix $\mathbf{CM} = (\mathbf{C} + \alpha\,\Delta t\,\mathbf{K})$
decompose constant coefficient matrix only once
for k=1  and  k < timeSteps
    time = time+Δt
    calculate temperature gradients at restrained nodes
        $\dot{\mathbf{T}}_k = (\mathbf{T}_k - \mathbf{T}_{k-1}) / \Delta t$
    calculate $\hat{\mathbf{T}} = \mathbf{T}_{k-1} + \Delta t\,(1-\alpha)\,\dot{\mathbf{T}}_{k-1}$
    modification of RHS: $\mathbf{R} = \mathbf{F}_k - \mathbf{K}\,\hat{\mathbf{T}}$
    backsubstitution of coefficient matrix  with new RHS
        $(\mathbf{C} + \alpha\,\Delta t\,\mathbf{K})\dot{\mathbf{T}}_k = \mathbf{R}$, solve for temperature gradient $\dot{\mathbf{T}}_k$
    compute temperature at next time step $\mathbf{T}_k = \hat{\mathbf{T}} + \alpha\,\Delta t\,\dot{\mathbf{T}}_k$
```

This algorithm has the advantage of being able to consider all values of α and Δt.

In particular, if α=0 then $\mathbf{T}_k = \hat{\mathbf{T}}$ and in case $\mathbf{C}$ is diagonal, no equations need to be solved. The case α=0 is called **explicit** and all other cases are called **implicit**.

The algorithm is conditionally stable for (0<= α < ½) and $\Delta t \leq \dfrac{2}{\beta_{\max}(1-2\alpha)}$ where $\beta_{\max}$ is the maximum eigenvalue of the corresponding eigenproblem. The algorithm is unconditionally stable for (½ <= α <= 1).

Special consideration needs to given to the case where specific boundary conditions are defined with prescribed values of the system variables, i.e. in heat conduction $\dot{\mathbf{T}}$ or $\mathbf{T}$ respectively.

In this case equation (11.1) needs to be expanded in the following form:

$$\begin{bmatrix} \mathbf{CM}_{ff} & \mathbf{CM}_{fp} \\ \mathbf{CM}_{pf} & \mathbf{CM}_{pp} \end{bmatrix} \begin{Bmatrix} \dot{\mathbf{T}}_f \\ \dot{\mathbf{T}}_p \end{Bmatrix}^k = \begin{Bmatrix} \mathbf{F}_f \\ \mathbf{F}_p \end{Bmatrix}^k - \begin{bmatrix} \mathbf{K}_{ff} & \mathbf{K}_{fp} \\ \mathbf{K}_{pf} & \mathbf{K}_{pp} \end{bmatrix} \begin{Bmatrix} \mathbf{T}_f + \Delta t(1-\alpha)\dot{\mathbf{T}}_f \\ \mathbf{T}_p + \Delta t(1-\alpha)\dot{\mathbf{T}}_p \end{Bmatrix}^{k-1}$$

where: superscripts (k) and (k+1) indicate the time step,

subscripts (f) indicates free boundary conditions for system variables and subscripts (p) indicates prescribed boundary conditions.

The actual storage scheme chosen does not reorder the system equations according to free and prescribed boundary conditions. It “keeps track” of these conditions via the concept of a status vector. Therefore, the equation above only serves as an explanation for the general problem to be taken into account.

Prescribed system variables in the primal vector $\dot{\mathbf{T}}_p$ do not need to be solved as they are prescribed. Therefore, the system only needs to be solved for the free system variables $\dot{\mathbf{T}}_f$.

$$\mathbf{CM}_{ff} * \dot{\mathbf{T}}_f{}^k = \mathbf{F}_f{}^k - \mathbf{CM}_{fp} * \dot{\mathbf{T}}_p{}^k - \left[\mathbf{K}_{ff}(\mathbf{T}_f + \Delta t(1-\alpha)\dot{\mathbf{T}}_f)^{k-1} + (\mathbf{K}_{fp}(\mathbf{T}_p + \Delta t(1-\alpha)\dot{\mathbf{T}}_p)^{k-1} \right]$$

$$\mathbf{CM}_{ff} * \dot{\mathbf{T}}_f{}^k = \mathbf{F}_f{}^k - \mathbf{CM}_{fp} * \dot{\mathbf{T}}_p{}^k - \begin{bmatrix} \mathbf{K}_{ff} & \mathbf{K}_{fp} \end{bmatrix} (\mathbf{T} + \Delta t(1-\alpha)\dot{\mathbf{T}})^{k-1}$$

In the above modification of the RHS-vector, the term with the product of the constant coefficient matrix with the prescribed temperature gradient at the restrained nodes is implicitly considered when solving the equations by the profile solver with status vector.

The other terms with the forcing function and the product of the unrestrained degrees of freedom of the system matrix with the temperature $\mathbf{T}$ and temperature gradient vector $\dot{\mathbf{T}}$ will be considered as a modification of the right-hand-side vector at each time step.

12 Time Step Integration for 2nd Order Differential Equations

problem: $\mathbf{M}\ddot{\mathbf{u}}(t)+\mathbf{C}\dot{\mathbf{u}}(t)+\mathbf{K}\,\mathbf{u}(t)=\mathbf{F}(t)$ and $\mathbf{u}(0)=\mathbf{d_0}$, $\dot{\mathbf{u}}(0)=\mathbf{v_0}$

where: $\mathbf{M}$ is the structural mass matrix

$\ddot{\mathbf{u}}$ is the acceleration vector

$\mathbf{C}$ is the structural damping matrix

$\dot{\mathbf{u}}$ is the velocity vector

$\mathbf{K}$ is the structural stiffness matrix

$\mathbf{u}$ is the displacement vector

$\mathbf{F}$ is forcing function vector of predefined deformations

$\mathbf{d_0}$ and $\mathbf{v_0}$ are the initial deformation and velocity vectors

eigenproblem: $(\mathbf{K} - \omega^2\,\mathbf{M})\,\mathbf{q} = 0$

where: ω is one of n-eigenvalues for which this equation is true

q is the corresponding eigenvector

solve: $\det(\mathbf{K} - \omega^2\,\mathbf{M}) = 0$

The original paper describing a general procedure for the solution of problems in structural dynamics is credited to Nathan M. Newmark [5] as a method of numerical integration particularly for use with digital computers. It was later called Newmark's method.

numerical integration: with accelerations $\mathbf{a}_{k+1}$ as primary dependent variables:

Newmark: $$(\mathbf{M}+\gamma\,\Delta\mathbf{t}\,\mathbf{C}+\beta\,\Delta\mathbf{t}^2\,\mathbf{K})\,\mathbf{a}_{k+1}=\mathbf{F}_{k+1}-\mathbf{C}\,\hat{\mathbf{v}}_{k+1}-\mathbf{K}\,\hat{\mathbf{d}}_{k+1}$$

where: $$\hat{\mathbf{d}}_{k+1}=\mathbf{d}_k+\Delta t\,\mathbf{v}_k+\Delta t^2\,(\tfrac{1}{2}-\beta)\,\mathbf{a}_k$$

$$\hat{\mathbf{v}}_{k+1}=\mathbf{v}_k+\Delta t\,(1-\gamma)\,\mathbf{a}_k$$ are called the predictors

and: $$\mathbf{d}_{k+1}=\hat{\mathbf{d}}_{k+1}+\beta\,\Delta t^2\,\mathbf{a}_{k+1}$$

$$\mathbf{v}_{k+1}=\hat{\mathbf{v}}_{k+1}+\gamma\,\Delta t\,\mathbf{a}_{k+1}$$ are called the correctors

The method is consistent for all values of γ and β. It is unconditionally stable for $(\gamma \geq \tfrac{1}{2})$ and $(\beta \geq \gamma/2)$. It is conditionally stable for $(\gamma \geq \tfrac{1}{2})$ and $(\beta < \gamma/2)$. The critical time step is:

$$\omega\,\Delta t_{cr}=\frac{\xi(\gamma-\tfrac{1}{2})+[\gamma/2-\beta+\xi^2(\gamma-\tfrac{1}{2})^2]^{1/2}}{(\gamma/2-\beta)}$$

Wilson's Θ: $(\mathbf{M}+\gamma\,\tau\,\mathbf{C}+\beta\,\tau^2\,\mathbf{K})\,\mathbf{a}_{k+\theta}=\mathbf{F}_{k+\theta}-\mathbf{C}\,\hat{\mathbf{v}}_{k+\theta}-\mathbf{K}\,\hat{\mathbf{d}}_{k+\theta}$ with: $\tau=\theta\,\Delta t$

where: $\mathbf{F}_{k+\theta}=(1-\theta)\,\mathbf{F}_k+\theta\,\mathbf{F}_{k+1}$ collocation of force

$$\hat{\mathbf{d}}_{k+\theta}=\mathbf{d}_k+\tau\,\mathbf{v}_k+\tau^2\,(\tfrac{1}{2}-\beta)\,\mathbf{a}_k$$

$$\hat{\mathbf{v}}_{k+\theta}=\mathbf{v}_k+\tau\,(1-\gamma)\,\mathbf{a}_k$$

the system is solved for $\mathbf{a}_{k+\theta}$ and from collocation of acceleration we get

$$\mathbf{a}_{k+\theta}=(1-\theta)\,\mathbf{a}_k+\theta\,\mathbf{a}_{k+1}$$

$$\mathbf{a}_{k+1}=\mathbf{a}_k+\frac{1}{\theta}(\mathbf{a}_{k+\theta}-\mathbf{a}_k)$$

For values of $\beta={}^1\!/_6$ and $\gamma={}^1\!/_2$ the method is referred to as Wilson's Θ-method. The value of $(\theta=1.420815)$ is recommended for this method with the additional requirement for $(\theta>1.366025)$.

Taylor's α: $(\mathbf{M}+\gamma\,\Delta\mathbf{t}\,\mathbf{C}+(1+\alpha)\beta\,\Delta\mathbf{t}^2\,\mathbf{K})\,\mathbf{a}_{k+1}=\mathbf{F}_{k+1}-\mathbf{C}\,\hat{\mathbf{v}}_{k+1}-\mathbf{K}\,[(1+\alpha)\,\hat{\mathbf{d}}_{k+1}-\alpha\,\mathbf{d}_k]$

where: $\hat{\mathbf{d}}_{k+1}=\mathbf{d}_k+\Delta t\,\mathbf{v}_k+\Delta t^2\,(\tfrac{1}{2}-\beta)\,\mathbf{a}_k$

$$\hat{\mathbf{v}}_{k+1}=\mathbf{v}_k+\Delta t\,(1-\gamma)\,\mathbf{a}_k$$

and: $\mathbf{d}_{k+1}=\hat{\mathbf{d}}_{k+1}+\beta\,\Delta t^2\,\mathbf{a}_{k+1}$

$$\mathbf{v}_{k+1}=\hat{\mathbf{v}}_{k+1}+\gamma\,\Delta t\,\mathbf{a}_{k+1}$$

Taylor's α-method is defined by

$\gamma={}^1\!/_2-\alpha$ and $\beta={}^1\!/_4(1-\alpha)^2$

with: $(-{}^1\!/_3\le\alpha\le 0)$

A discussion on consistency, stability and accuracy can be found in [6]. From this results:

- the minimum order of accuracy for all α,β,γ and ξ (real damping) is one.
- if $(\alpha+\gamma={}^1\!/_2)$ and either $\xi=0$ or $(\gamma={}^1\!/_2)$ the method is second order accurate
- if $\xi=0$, $(\alpha+\gamma={}^1\!/_2)$ and $(\beta={}^1\!/_{12}+\alpha^2)$ the method is third order accurate
- the methods are unconditionally stable for $(-{}^1\!/_2\le\alpha\le 0)$, $(\beta\ge{}^1\!/_4-\frac{\alpha}{2})$ and $(\alpha+\gamma={}^1\!/_2)$

Combination of Methods into a single algorithm:

$$[\mathbf{M}+\gamma\tau\mathbf{C}+(1+\alpha)\beta\tau^2\mathbf{K}]\mathbf{a}_{k+\theta}=(1-\theta)\mathbf{F}_k+\theta\mathbf{F}_{k+1}-\mathbf{C}\hat{\mathbf{v}}_{k+\theta}-\mathbf{K}[(\mathbf{1}+\alpha)\hat{\mathbf{d}}_{k+\theta}-\alpha\,\mathbf{d}_k] \qquad 12.1$$

with:

$$\hat{\mathbf{d}}_{k+\theta}=\mathbf{d}_k+\tau\,\mathbf{v}_k+\tau^2(½-\beta)\mathbf{a}_k$$

$$\hat{\mathbf{v}}_{k+\theta}=\mathbf{v}_k+\tau(1-\gamma)\mathbf{a}_k$$

and:

$$\mathbf{a}_{k+1}=\mathbf{a}_k+\frac{1}{\theta}(\mathbf{a}_{k+\theta}-\mathbf{a}_k)$$

$$\mathbf{d}_{k+1}=\mathbf{d}_k+\Delta t\,\mathbf{v}_k+\Delta t^2\,(½-\beta)\mathbf{a}_k+\beta\,\Delta t^2\,\mathbf{a}_{k+1}$$

$$\mathbf{v}_{k+1}=\mathbf{v}_k+\Delta t\,(1-\gamma)\,\mathbf{a}_k+\gamma\,\Delta t\,\mathbf{a}_{k+1}$$

Integration parameters are set depending on the method chosen:

Newmark's method:	$\alpha=0,\ \theta=1$ and β,γ will have to be defined
Wilson's Θ-method	$\alpha=0,\ \beta=\frac{1}{6},\ \gamma=\frac{1}{2}$ only θ will have to be defined
Taylor's α-method	$\beta=\frac{1}{4}(1-\alpha)^2,\ \ \gamma=\frac{1}{2},\ \theta=1$ only α will have to be defined

The solution algorithm is defined by the following steps:

```
at time =0: calculate initial accelerations $\mathbf{a_0}$ at unrestrained nodes
$\mathbf{M}\,\mathbf{a}_0=\mathbf{F}_0-\mathbf{C}\,\mathbf{v}_0-\mathbf{K}\,\mathbf{d}_0$

compute and decompose constant coefficient matrix
$\mathbf{CM}=\mathbf{M}+\gamma\tau\mathbf{C}+(1+\alpha)\beta\tau^2\,\mathbf{K}$ only once

for k=1 and   k < timeSteps
    k = k+1, time = time+ Δt

    calculate predictors
    $\hat{\mathbf{d}}_{k+\theta}=\mathbf{d}_{k-1}+\tau\,\mathbf{v}_{k-1}+\tau^2(½-\beta)\mathbf{a}_{k-1}$
    $\hat{\mathbf{v}}_{k+\theta}=\mathbf{v}_{k-1}+\tau(1-\gamma)\mathbf{a}_{k-1}$

    calculate new RHS
    $\mathbf{R}=(1-\theta)\mathbf{F}_{k-1}+\theta\mathbf{F}_k-\mathbf{C}\hat{\mathbf{v}}_{k+\theta}-\mathbf{K}[(\mathbf{1}+\alpha)\hat{\mathbf{d}}_{k+\theta}-\alpha\,\mathbf{d}_{k-1}]$

    backsubstitution of coefficient matrix with new RHS
    $(\mathbf{M}+\gamma\tau\mathbf{C}+(1+\alpha)\beta\tau^2\,\mathbf{K})\mathbf{a}_{k+\theta}=\mathbf{R}$ for acceleration $\mathbf{a}_{k+\theta}$

    compute accelerations, displacements and velocities at next time step
    $\mathbf{a}_k=\mathbf{a}_{k-1}+\frac{1}{\theta}(\mathbf{a}_{k+\theta}-\mathbf{a}_{k-1})$
    $\mathbf{d}_k=\mathbf{d}_{k-1}+\Delta t\,\mathbf{v}_{k-1}+\Delta t^2\,(½-\beta)\mathbf{a}_{k-1}+\beta\,\Delta t^2\,\mathbf{a}_k$
    $\mathbf{v}_k=\mathbf{v}_{k-1}+\Delta t\,(1-\gamma)\,\mathbf{a}_{k-1}+\gamma\,\Delta t\,\mathbf{a}_k$
```

Special consideration needs to given to the case where specific boundary conditions are defined with prescribed values of the system variables, i.e. in elasticity analysis the accelerations $\mathbf{a}$ or the deflections $\mathbf{d}$ respectively.

In this case equation (1.13.1) needs to be expanded in the following form:

$$\begin{bmatrix} \mathbf{CM}_{ff} & \mathbf{CM}_{fp} \\ \mathbf{CM}_{pf} & \mathbf{CM}_{pp} \end{bmatrix} \begin{Bmatrix} \mathbf{a}_f \\ \mathbf{a}_p \end{Bmatrix}^{k+1} = (1-\theta)\begin{Bmatrix} \mathbf{F}_f \\ \mathbf{F}_p \end{Bmatrix}^{k} + \theta\begin{Bmatrix} \mathbf{F}_f \\ \mathbf{F}_p \end{Bmatrix}^{k+1} - \begin{bmatrix} \mathbf{C}_{ff} & \mathbf{C}_{fp} \\ \mathbf{C}_{pf} & \mathbf{C}_{pp} \end{bmatrix} \begin{Bmatrix} \hat{\mathbf{v}}_f \\ \hat{\mathbf{v}}_p \end{Bmatrix}^{k+\theta}$$

$$- \begin{bmatrix} \mathbf{K}_{ff} & \mathbf{K}_{fp} \\ \mathbf{K}_{pf} & \mathbf{K}_{pp} \end{bmatrix} \left[(1+\alpha)\begin{Bmatrix} \hat{\mathbf{d}}_f \\ \hat{\mathbf{d}}_p \end{Bmatrix}^{k+\theta} + \alpha\begin{Bmatrix} \mathbf{d}_f \\ \mathbf{d}_p \end{Bmatrix}^{k} \right]$$

where: superscripts (*k*) and (*k+1*) indicate the time step,

subscripts (*f*) indicates **f**ree boundary conditions for system variables and subscripts (*p*) indicates **p**rescribed boundary conditions.

The actual storage scheme chosen does not reorder the system equations according to free and prescribed boundary conditions. It “keeps track” of these conditions via the concept of a status vector. Therefore, the equation above only serves as an explanation for the general problem to be taken into account.

Prescribed system variables in the primal vector $\mathbf{a}_p$ do not need to be solved as they are prescribed. Therefore, the system only needs to be solved for the free system variables $\mathbf{a}_f$.

$$\mathbf{CM}_{ff} * \mathbf{a}_f^{k+1} = (1-\theta)\mathbf{F}_f^k + \theta\mathbf{F}_f^{k+1} - \mathbf{CM}_{fp} * \mathbf{a}_p^{k+1} - \mathbf{C}_{ff}\, \hat{\mathbf{v}}_f^{\,k+\theta} - \mathbf{C}_{fp}\, \hat{\mathbf{v}}_p^{\,k+\theta}$$

$$- \mathbf{K}_{ff}[(1+\alpha)\hat{\mathbf{d}}_f^{\,k+\theta} + \alpha\,\mathbf{d}_f^k] - \mathbf{K}_{fp}[(1+\alpha)\hat{\mathbf{d}}_p^{\,k+\theta} + \alpha\,\mathbf{d}_p^k]$$

$$\mathbf{CM}_{ff} * \mathbf{a}_f^{k+1} = (1-\theta)\mathbf{F}_f^k + \theta\mathbf{F}_f^{k+1} - \mathbf{CM}_{fp} * \mathbf{a}_p^{k+1} - [\mathbf{C}_{ff} \quad \mathbf{C}_{fp}]\hat{\mathbf{v}}^{\,k+\theta}$$

$$- [\mathbf{K}_{ff} \quad \mathbf{K}_{fp}][(1+\alpha)\hat{\mathbf{d}}^{\,k+\theta} + \alpha\,\mathbf{d}^k]$$

In the above modification of the RHS-vector, the term with the product of the constant coefficient matrix with the prescribed accelerations at the restrained nodes is implicitly considered when solving the equations by the profile solver with status vector.

The other terms with the forcing function, the product of the damping matrix with the velocity vector $\hat{\mathbf{v}}$ and of the stiffness matrix with the deflection vectors $\hat{\mathbf{d}}$ and $\mathbf{d}$ at the unrestrained nodes may be evaluated at each time step as a modification of the right-hand-side vector.

13 A general Processor for the FEM Analysis

The control of the analysis for a system with a profile storage scheme for the system equations is implemented in class *Analysis* contained in *package framework.model.* Class *Analysis* provides the following functionality

- a constructor *Analysis(Model m)*
  which will establish all element references of a particular model by invoking *m_model.setReferences()*
- a method for establishing the system matrix
  public void systemMatrix() throws FemException, AlgebraicException
- methods for setting the system indices of the nodes, for determining the system dimension and setting the profile vector, for computing and recomputing the system matrix, for computing the system vector, setting the status vector, solving the equations, finding the eigenstates of the system, computing the diagonal system matrix (specific mass or heat), for defining time dependent forcing function and boundary conditions, performing 1st or 2nd order time integration of the system equations and for printing the results.

private void setNodeSystemIndices()
private void determineDimension()
private void setProfile( )

private void computeSystemMatrix() throws FemException, AlgebraicException
private void reComputeSystemMatrix() throws FemException, AlgebraicException
***public** void computeSystemVector()*
***public** void setStatusVector()*

***public** void solveEquations() throws AlgebraicException*

// eigensolution
***public** void eigenstates() throws FemException, AlgebraicException*
private void computeDiagonalMatrix( ) throws FemException

// time integration analysis
private void compute1stOrderForcingFunction(ITimeInt1st timeInt)
private void compute1stOrderBoundaryConditions(ITimeInt1st timeInt)
private void piecewiseLinear(double dt, double[] interval, double[] function)
private void periodic(dt, amplitude, frequency, phaseAngle, function)
***public** void timeIntegration1stOrder( ) throws FemException, AlgebraicException*
private void compute2ndOrderForcingFunction(ITimeInt2nd timeInt)
***public** void timeIntegration2ndOrder( ) throws FemException, AlgebraicException*

***public** void printSystemMatrix( )*
***public** void printDiagonalMatrix( )*
***public** void printProfileVector( )*
***public** void printStatusVector( )*

Class Analysis is contained in *package framework.model*. It requires to import the standard Java functionality for an iterator object on container classes.

It further requires to import functionality of the general FEM Framework for the interfaces to the FEM components *INode*, *IElement*, *ILoad*, *ISupport*, *IEigenstates, ITimeDepAreaLoad, ITimeDepNodeLoad, ItimeDepSupport, ITimeInt1st* and *ITimeInt2nd*.

Class Analysis defines attributes for an iterator object for traversing container classes, for a model object for managing model data, for a systems equations object for setting up the systems equations, for an equation solver object implementing the solution of a profile matrix with status information and for the dimension of the systems equations.

The constructor of class Analysis establishes a reference to object m_model. The reference is passed to the constructor of class Analysis as a result of parsing the external data file. Next, a message with the identifier for the model to be analyzed is given. Before the start of the analysis, all internal references for FEM components must be established on the basis of the persistent identifiers by invoking method setReferences for a specific model.

Method systemMatrix will set up the system matrix for the analysis. For this purpose, the corresponding system indices for each node are evaluated, all individual element matrices are determined and added to the system matrix and the status vector is computed based upon the support conditions.

```
public class Analysis  {
    private Iterator<?> iter;
    private Model m_model;
    private Equations m_systemEquations;
    private ProfileSolverStatus m_profileSolver;
    private int dimension;
    private boolean decomposed = fals, nodeSystemIndices = false,
                     setDimension = false, setProfile = false, diagonalMatrix = false;

    public Analysis(Model m)  {
         m_model = m;
         System.out.println("Model '" + m_model.getId() + "' will be evaluated.");
         m_model.setReferences();
    }

    public void systemMatrix() throws FemException, AlgebraicException  {
        if (!nodeSystemIndices) setNodeSystemIndices();
        computeSystemMatrix();
        setStatusVector();
    }
```

Method *setNodeSystemIndices* will compute system indices by traversing all node objects and by incrementing a counter according to the node numbers and to the number of nodal degrees of freedoms. The result is stored in vector *index* of class *Node*.

Method ***computeSystemMatrix*** is defined to assemble the system matrix by first determining the dimension of the system equations. The dimension is computed by traversing all nodes and incrementing the dimension according to each individual number of nodal degrees of freedom. The system matrix is then allocated accordingly.

In the next step, the profile of the system equations is computed by traversing all elements and invoking method *setProfile* of class *SystemEquations*.

Finally, all elements are traversed and method *computeMatrix* is invoked for each individual element. The resulting element matrix is then added to the system matrix.

```
private void computeSystemMatrix() throws FemException  {
    IElement element;
    double elementMatrix[ ][ ];
    int index[ ];
    if (!setDimension) determineDimension();
    if (!setProfile) setProfile();

    // traverse the elements to assemble the element coefficients
    iter = m_model.iterator(IElement.class);
    while (iter.hasNext()) {
        element = (IElement) iter.next();
        index = element.getSystemIndices();
        elementMatrix = element.computeMatrix();
        m_systemEquations.addMatrix(index, elementMatrix);
}    }
```

Method ***computeSystemVector*** defines attributes for a node object, a node load object, an element load object, the index vector for system indices and the load vector.

It traverses all node load objects in order to compute the corresponding element load vector and adds it to the system load vector by observing the system indices of the nodes.

It finally traverses all element loads, computes the element load vector and adds it to the system load vector by observing the system indices of the element.

```
public void computeSystemVector() {
    ILoad load;
    int index[ ];
    double loadVector[ ];
    // traverse the loads
    iter = m_model.iterator(ILoad.class);
    while (iter.hasNext()) {
        load = (ILoad) iter.next();
        index = load.getSystemIndex();
        loadVector = load.computeLoadVector();
        m_systemEquations.addVector(index, loadVector);
}    }
```

Method setStatusVector is used to indicate if a certain node is restricted to prescribed values of nodal degrees of freedom. Most often this is a support condition restricting the corresponding nodal degree of freedom to be equal to 0.

This is achieved by traversing all support objects, by identifying the corresponding nodes and system indices and by retrieving prescribed nodal values and restraint conditions. Method *setStatusVector* is concluded by invoking method *setStatus* of class *SystemEquations* for each restraint condition.

```
public void setStatusVector( ) {
    // loop on the supports
    iter = m_model.iterator(ISupport.class);
    while (iter.hasNext()) {
        ISupport support = (ISupport) iter.next();
        INode node = support.getNode();
        int[ ] index = node.getSystemIndex();
        double[ ] prescribed = support.getPrescribed();
        boolean[ ] restraint = support.getRestraint();
        for (int i = 0; i < restraint.length; i++)
            if (restraint[i])
                    m_systemEquations.setStatus(true, index[i], prescribed[i]);
}   }
```

Method *solveEquations* throws an algebraic exception in case an exceptional condition occurs. It will invoke the constructor of class *ProfileSolverStatus* with the system matrix, the load vector, the primal unknowns, the dual unknowns, the system status vector and the profile vector as parameters. It will subsequently perform the triangular decomposition and the forward and backward solution of the system equations.

After decomposition the original matrix is lost and therefore a flag is checked if the matrix is already decomposed or not.

```
private void solveEquations( ) throws AlgebraicException {
    if (!decomposed) {
        m_profileSolver = new ProfileSolverStatus(
            m_systemEquations.getMatrix(), m_systemEquations.getVector(),
            m_systemEquations.getPrimal(), m_systemEquations.getDual(),
            m_systemEquations.getStatus(), m_systemEquations.getProfile());
        m_profileSolver.decompose( );
        decomposed = true;
    }
    m_profileSolver.solve( );
}
```

Method *eigenstates* will evaluate the eigensolution (eigenvalues and eigenvectors) of a given problem. It will next traverse all objects of class Eigenstates, read the number of eigenstates to be considered, determine the profile of the system equations, read the general B-matrix and perform the eigensolution.

```
public void eigenstates() throws FemException, AlgebraicException {
    EigenSolver m_eigenSolver;
    int numberOfStates;
    double[ ] bDiag,eigenvalues;
    double[ ][ ] aMatrix, bMatrix, eigenvectors;

    iter = m_model.iterator(IEigenstates.class);
    while(iter.hasNext()){
        IEigenstates eigenstates = (IEigenstates)iter.next();
        numberOfStates = eigenstates.getNumberOfStates();
        aMatrix = m_systemEquations.getMatrix();
        if (!diagonalMatrix) computeDiagonalMatrix( );
        bDiag = eigenstates.getGeneralBMatrix();
        // general B-Matrix is expanded to the same structure as A
        bMatrix = new double[dimension][ ];
        int row, col;
        for (row = 0; row < aMatrix.length; row++) {
            bMatrix[row] = new double[aMatrix[row].length];
            for (col = 0; col < bMatrix[row].length-1; col++)
                    bMatrix[row][col] = 0.;
            bMatrix[row][col] = bDiag[row];
        }
```

In case the system equations are not decomposed yet, a decomposition of the system equations is performed.

Finally, a new object of class Eigensolver is instantiated, the eigensolution is performed, the solution found is saved and a flag for successful completion is set.

```
            if (!decomposed) {
                m_profileSolver = new ProfileSolverStatus(
                                                        m_systemEquations.getMatrix(),
                                                        m_systemEquations.getStatus(),
                                                       m_systemEquations.getProfile());
                m_profileSolver.decompose();
                decomposed = true;
            }

            m_eigenSolver = new EigenSolver(m_systemEquations.getMatrix(),
                                        bMatrix, m_systemEquations.getProfile(),
                                m_systemEquations.getStatus(),numberOfStates);
            m_eigenSolver.solveEigenstates();

            // save eigenvalues and eigenvectors
            eigenvalues  = new double[numberOfStates];
            eigenvectors = new double[numberOfStates][dimension];
            for(int i=0; i < numberOfStates; i++) {
                 eigenvalues[i]  = m_eigenSolver.getEigenValue(i);
                 eigenvectors[i] = m_eigenSolver.getEigenVector(i);
            }
            eigenstates.setEigenvalues(eigenvalues);
            eigenstates.setEigenvectors(eigenvectors);
            eigenstates.setStatus(m_systemEquations.getStatus());
        }
        m_model.setEigen(true);
        System.out.println("eigensolution successfully performed");
    }
```

Method *timeIntegration1stOrder* will perform the time step integration of first order differential equations.
It will traverse all objects of class ITimeInt1st, read the time step, parameter alfa, the specific heat matrix, the initial temperatures and the forcing function.

```
    public void timeIntegration1stOrder() throws FemException, AlgebraicException {
        TimeIntegration1stOrder m_timeIntegration;
        double dt, alfa;
        double[ ] specificHeat;
        double[ ][ ] temperature, forceFunction;
        // compute specific heat matrix
        if (!diagonalMatrix) computeDiagonalMatrix();
        specificHeat = m_systemEquations.getDiagonalMatrix();
        iter = m_model.iterator(ITimeInt1st.class);
        while(iter.hasNext()){
            ITimeInt1st timeInt = (ITimeInt1st)iter.next();

            // ... compute time dependent forcing function and boundary condition
            compute1stOrderForcingFunction(timeInt);
            compute1stOrderBoundaryConditions(timeInt);

            dt                  = timeInt.getDt();
            alfa                = timeInt.getParameter1();
            temperature         = timeInt.getFieldVariables();
            temperature[0]      = timeInt.getInitial();
            forceFunction       = timeInt.getForceFunction();

            if (decomposed){
                reComputeSystemMatrix();
                decomposed = false;
            }
            m_timeIntegration = new TimeIntegration1stOrderStatus(
                specificHeat, m_systemEquations.getMatrix(), forceFunction,
                m_systemEquations.getPrimal(), m_systemEquations.getDual(),
                m_systemEquations.getProfile(), m_systemEquations.getStatus(),
                dt, alfa, temperature);
            m_timeIntegration.perform();

            // save nodal time histories
            iter = m_model.iterator(INode.class);
            while (iter.hasNext()) {
                INode node = (INode) iter.next();
                int[ ] index = node.getSystemIndex();
                double[ ][ ] nodalTemperatures = new double [1][temperature.length];
                double[ ][ ] nodalGradients      = new double [1][temperature.length];
                for (int k = 0; k < temperature.length; k++) {
                        nodalTemperatures[0][k] = temperature[k][index[0]];
                        nodalGradients[0][k]    = m_timeIntegration.getTDot()
                                                                        [k][index[0]];
                }
                node.setNodalHistory(nodalTemperatures, nodalGradients);
        }   }
        m_model.setTimeint(true);
        System.out.println("1st order time integration successfully performed");
    }
```

Method *timeIntegration2ndOrder* performs the time step integration of second order differrential equations. It traverses all objects of class ITimeInt2nd, reads the time step, parameters, the mass and damping matrix, initial displacements, velocities and the forcing function.

14 Output of System Results

The general, application independent functionality for output of results is contained in *package framework.post* which contains an *interface IOutput* and an abstract class *OutputAdapterConsole*.

Interface IOutput simply defines functionality for setting the model and for output of results in general.

Abstract class OutputAdapterConsole defines the basic alphanumerical output functionality for the general FEM analysis. It controls the output of the model, the influences and the system behaviour.

```
package framework.post;

import framework.model.FemException;
import framework.model.Model;

public abstract class OutputAdapterConsole implements IOutput {
    protected Model model;

    public void OutputAdapterConsole (Model _model)  { model = _model; }

    protected abstract void printModel();
    protected abstract void printInfluences();
    protected abstract void printBehaviour() throws FemException;}
```

Bibliography

[1] Ray W. Clough: “The Finite Element Method in Plane Stress Analysis”, Conference papers / American Society of Civil Engineers, Conference on Electronic Computation : Pittsburgh, Pa., September 8-9, 1960

[2] Sun Microsystems, Inc., Copyright 1995-2006: Java™ Standard Edition 6, API Specification, http://java.sun.com/javase/6/.

[3] Sun Microsystems, Inc., Copyright 2006: Download The Java Tutorial, http://java.sun.com/docs/books/tutorial/.

[4] Pahl, P. J.: “Finite Element Method, Stationary Heat Transfer”, TU Berlin, 2001

[5] Newmark, Nathan M.: “A Method of Computation for Structural Dynamics”, Journal of the Engineering Mechanics Division, Proceedings of the American Society of Civil Engineers, July 1959

[6] Hilber, H.M.: “Analysis and Design of Numerical Integration Methods in Structural Dynamics”, EERC Report No 76-29, Berkeley, Nov 1976