

# **Structural Frame Analysis with the Finite Element Method (FEM)**

Bauhaus-Universität Weimar, Informatik im Bauwesen

2009

## Contents

1	Installation	3
2	Formulation of the Finite Element Solution	4
3	Implementation on the basis of the FEM Framework	6
3.1	Main programs for structural analysis:	8
3.2	Parsing persistent model information (input file)	10
3.3	Model information for the analysis	22
3.3.1	establishing element matrices	22
3.3.2	defining material properties	24
3.3.3	defining external influences on the system	25
3.3.4	defining support conditions	26
3.3.5	defining information for time integration	26
3.4	Output of system results	27
3.4.1	Alphanumeric output of results	27
3.4.2	Graphical output of results	29

# Analysis of Structural Frame Problems with the Finite Element Method (FEM)

## 1 Installation

Application software on the basis of the FEM-Framework may be downloaded as a zip-file that is provided under the name **FEM.zip**. Root directory for the installation is *FEM*. The zip-files contains:

- the source files (subdirectory *source*) for various FEM-applications which can either be compiled separately in an individual environment or imported into an IDE (e.g. Eclipse),
- the class files (subdirectory *binary*) for applications to be run directly without recompilation,
- some input files (subdirectory *input*) for example calculations to be performed,
- the Java-documentation (subdirectory *doc*) and
- some class notes (subdirectory *notes*) for explanation of the fundamental equations and solution strategies.

Currently, three different applications are provided

- linear elasticity analysis (subdirectory *source/elasticity*),
- linear stationary and transient heat transfer analysis (subdirectory *source/heat*) and
- linear static and dynamic analysis of structural beams and frames (subdirectory *source/structuralAnalysis*).

Finally, a 2D-CAD application is provided (subdirectory *cad*) for integration of analysis into an interactive CAD-environment. The CAD-software used for these purposes is openly available under the name *cademia* (<http://www.cademia.org/>).

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 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 Formulation of the Finite Element Solution

The analysis of problems in beam analysis consists of the Finite Element approximation of the system equations with a corresponding formulation of the element equations, the influences on the model (loads), the constitutive relations (material properties) and the external support conditions.

Furthermore, a specific element must be chosen that is appropriate for the problem to be solved defining the geometry of the problem (element geometry) and the element characteristics for the model.

The differential form of the **system equation** governing the behaviour of general beam elements in structural analysis is given as follows (see for example: [3]):

structural frameworks (1-dimensional) with axial deformation (truss), bending deformation (beams, frames) are commonly modelled by the	
differential equation for axial deformation	$EA \frac{d^2 u}{dx^2} = p_x$
and bending deformation	$EI \frac{d^4 w}{dx^4} = p_z$

The integral form of the beam equations for all elements is given by, s. [2]:

$$\sum_e \int_l \left( \frac{d(\delta u)}{dx} EA \frac{du}{dx} - \delta u p_x \right) dx + \sum_e \int_l \left( \frac{d^2(\delta w)}{dx^2} EI \frac{d^2 w}{dx^2} - \delta w p_z \right) dx = 0$$

The FEM approximation of the integral form is achieved by approximation of the deformation via form functions  $S(x)$

$$u(x) = S_D^T(x) \mathbf{u} = S_{1D}(x) u_a + S_{2D}(x) u_b$$

$$w(x) = S_B^T(x) \mathbf{w} = S_{1B}(x) w_a + S_{2B}(x) \varphi_a + S_{3B}(x) w_b + S_{4B}(x) \varphi_b$$

$\left( \sum_e \int_l \frac{\delta S_{De}(x)}{dx} (EA)_e \frac{dS_{De}^T(x)}{dx} dx \right) u_s$	element matrices for axial deformation (trusses)
$\left( \sum_e \int_l B^T(x) (EI)_e B(x) dx \right) w_s$	element matrices for bending deformation

Depending on the type of elements and interpolation functions chosen this will result in a different set of linear equations of the form:

$$\mathbf{K}_s \mathbf{u}_s = \mathbf{p}_s - \mathbf{q}_s$$

The system matrix  $\mathbf{K}_s$  is multiplied by the vector of nodal deformations  $\mathbf{u}_s$  referred to as primal system vector. The Right-Hand-Side (RHS) consists of the external influences on the system referred to as load vector  $\mathbf{p}_s$  and of the support conditions on the system  $\mathbf{q}_s$  referred to as dual system vector.

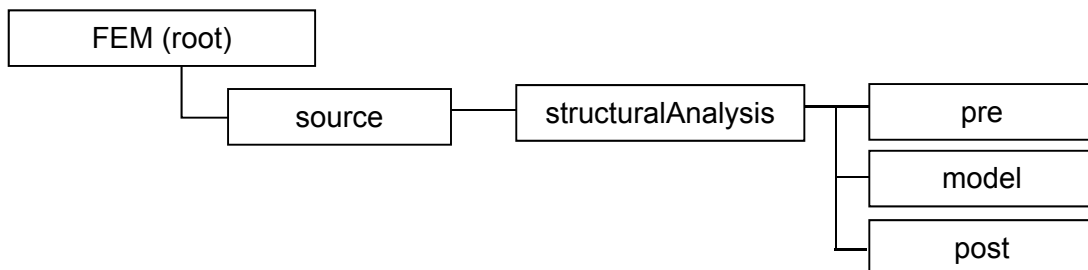
### 3 Implementation on the basis of the FEM Framework

Applications of the FEM for the solution of physical problems are embedded within the context of the general FEM Framework defined before.

In this context, all applications are stored in the hierarchical structure of the FEM Framework under the hierarchy *application*. In the case of **2-dimensional linear beam and frame analysis**, the corresponding functionality is stored under the hierarchy *source* in subdirectory *structuralAnalysis*.

Any application must define

- a **main program** for controlling a specific application analysis  
→ contained in the hierarchy structure of the specific application, i.e. *structuralAnalysis*
- a **file parser** for parsing the input file containing the persistently stored model data  
→ contained in the hierarchy under *structuralAnalysis.pre*
- several classes defining the complete **model** information, which in the case of 2-dimensional linear elasticity problems includes element, material, load and support information  
→ contained in the hierarchy under *structuralAnalysis.model*
- a class for **output** of information on the console and optionally on a graphical screen  
→ contained in the hierarchy under *structuralAnalysis.post*

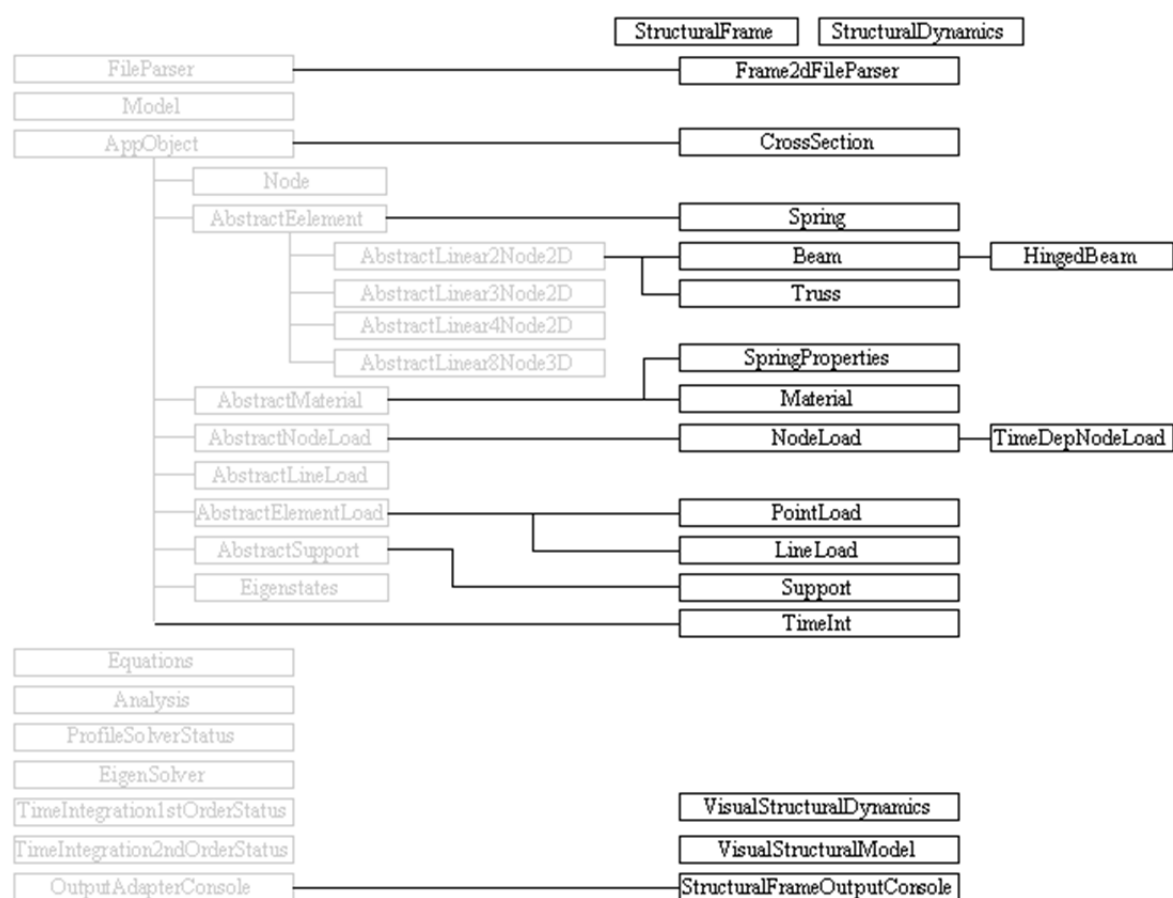


The implementation of classes defining a new application in case of structural analysis problems using FEM consists of a file parser for reading a new model, of the application specific element, material, load and support definitions, of the definitions for time integration of the model and of the respective functionality for output of system results.

It is important to note that the mathematical description of the physical problem is almost exclusively restricted to the classes in the model.

The new classes are embedded into the class hierarchy of the FEM Framework in the following way:

## Class Definitions (StructuralFrame) on the basis of the Framework:



### 3.1 Main programs for structural analysis:

There are two main program for the analysis of 2-dimensional linear structural analysis problems implemented in class **StructuralFrame** and **StructuralDynamics**. Both are contained in package *structuralAnalysis*.

Both applications need to import general Java functionality of File input and output (File IO). They further need to import functionality of the FEM Framework for the model and the analysis and functionality for file handling contained in the FEM utilities. Finally, functionality for parsing input information and displaying model definition and behavior in heat applications need to be imported.

**Class *StructuralFrame*** defines objects for the model, analysis and output as static attributes.

*Method main* starts by invoking method *getInputFile* from utility class *FileHandling*. The selection of applicable input files is accomplished via a file-chooser dialog box. Method *getInputFile* will require two arguments, one for specifying storage location relative to the root-directory, i.e. FEM, and a header definition for the corresponding dialog box.

The essential part of method *main* is defined by the part that controls the flow of the analysis enclosed in a try and catch-block for possible exceptions generated during analysis.

First, a new model object is generated on the basis of input data as a result from parsing the input file. A message will report the successful execution of file parsing.

Next, a new object *output* of class *StructuralFrameOutput* is generated for the specific model to be analyzed. Method *outModel* of this object will allow for checking model input.

Next, a new analysis object is generated for the specific model and method *computeSystemMatrix* of that object is initiated. Subsequently, the system vector is evaluated, the system equations are solved and the results are saved. Successful execution is acknowledged by a corresponding message.

Next, object *output* of class *StructuralFrameOutput* is used for output of results for the model analyzed.



	<pre> public class <b>StructuralFrame</b> {     private static Model model;     private static Analysis analysis;     private static StructuralFrameOutput output;      public static void main(String args[ ]) {         File file = FileHandling.getInputFile("/input/structuralFrame","FEM input files","inp");         try {             // setup and visualize model, analyze model and output results             model = new Frame2dFileParser(file).getModel();             analysis = new Analysis(model);             output = new StructuralFrameOutput(model);             output.outModel();              <b>analysis.computeSystemMatrix();</b>             <b>analysis.computeSystemVector();</b>             <b>analysis.solveEquations();</b>              output.outStationary(); </pre>
--	--

In case of a structural dynamics analysis, class **StructuralDynamics** is also contained in package *structuralAnalysis* and the analysis process only differs by a different definition of the analysis process, i.e. instead of setting up the system vector and solving the set of equations, the eigenstate of the system is evaluated and time integration performed.

A reference to the *analysis* object needs to be included in the *StructuralFrameOutput* constructor for enabling recalculation of time integration results in the *output* object with new parameters.

	<pre> // setup and visualize model, analyze model and output results model = new Frame2dFileParser(file).getModel(); analysis = new Analysis(model); output = new StructuralFrameOutput(model, <b>analysis</b>); output.outModel();  <b>analysis.computeSystemMatrix( );</b> <b>analysis.eigenstates( );</b> <b>analysis.timeIntegration2ndOrder( );</b>  output.outTransient(); </pre>
--	---

## 3.2 Parsing persistent model information (input file)

**Class *Frame2dFileParser*** is provided for parsing of model information that is persistently stored in an input file in Unicode format.

The functionality is contained in *package structuralAnalysis.pre*. It needs to import general Java functionality about Java file input and output (IO). It further needs to import functionality of the FEM Framework with respect to *Model* objects, the *FileParser* and corresponding exceptional conditions. It needs to import application specific functionality for FEM components as there are elements, loads, materials and supports. It finally needs to import application specific functionality for dynamic analysis as there are the eigensolution and time integration. 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 *Frame2dFileParser*.

Class *Frame2dFileParser* is derived from class *FileParser* of the Framework and can thus utilize the keywords defined in that class:

```
// public void parseIdentifier() throws IOException, ParseException
//      identifier                                1 modelName

// public void parseDimensions() throws IOException, ParseException
//      dimensions                                2 spatialDimensions, nodalDegreesOfFreedom

// public void parseNodes() throws IOException, ParseException
//      nodes                                     1 number of nodal degrees of freedom
//                                                    2 name, x
//                                                    3 name, x, y
//                                                    4 name, 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

// public void parseEigenstates() throws IOException, ParseException
//      eigenstates                                2 name, numberOfStates
```

Class *Frame2dParser* defines **additional keywords for structural analysis**.

```
// trusses          5: name, node1, node2, crossSection, material
// beams           5: name, node1, node2, crossSection, material
// hingedBeams     6: name, node1, node2, crossSection, material,
//                   hinge type(f(irst),s(econd))

// crossSections   2: name, area
//                   3: name, area, moment of inertia(Ixx)

// materials       2: name, Young's modulus
//                   3: name, Young's modulus, Poisson ratio
//                   4: name, Young's modulus, Poisson ratio, specific mass
// springProperties 4: name, Kx, Ky, Kphi, discrete spring stiffness values

// nodeLoads       4: name, node, loadX, loadY
//                   5: name, node, loadX, loadY, moment

// pointLoads      5: name, element, pX, pY, offset

// lineLoads       6: name, element, p1X, p1Y, p2X, p2Y
//                   7: name, element, p1X, p1Y, p2X, p2Y, localCoord(true)

// supports        6: name, node, type(xyr), valueX, valueY, valueR
// springSupports 3: name, node, springPropertyId

// timeIntegration 6: id, tmax, dt, method, parameter1, parameter2
// damping         2: nodeId = "all", value - for uniform damping ratio for all dof
//                   2,3,4: nodeId, damping ratio at nodal dof 1,2,3
// initialDeformations 2: dof, d0
// initialVelocities 2: dof, v0
// forcingFunction 1: fileName
// timeDependentNodeLoad forcing function from time dependent node loads
//                   3: nodeLoadId, nodeId, nodal degree of freedom
//                   or for ground excitation
//                   4: nodeLoadId, nodeId, nodal degree of freedom, ground
//                   followed by a separate line for periodic excitation
//                   3: amplitude, frequency, phase angle
//                   or piece-wise linear excitation
//                   >3: startTime, startValue, [endTime, endValue]...
```

Class **Frame2dFileParser** provides a constructor of class *Frame2dFileParser* 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 keyword 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 keyword are available.

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 11 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 (N0 through N10), e.g.

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

Generate a 2-dimensional array of 3x3 nodes at a spacing of 1. unit starting from the origin (e.g. 0.;1.).

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 (n00, n01, n02, n10, n11, n12, n20, n21, n22).

Mesh generation will start with the second direction followed by the first direction, i.e. n00(0.;0.), n01(0.;1.), n02(0.;3.), n10(1.;0.), ..., n22(3.;3.)

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.	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.

New nodes will be generated with respect to the origin specified (0., 0., 0.) Each node will have a unique identifier based upon an initial identifier (N000, N001, N002, N003, N010 through N333).

Mesh generation will start with the third direction, followed by the second concluding with the first direction, i.e. N000(0.;0.;0.), N001(0.;0.;1.), N002(0.;0.;3.), N003(0.;0.;6.), N010(0.;1.;0.), ..., N333(6.;6.;6.).

**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.

### Application specific file parser:

Information that is specific for a particular application must be parsed via an application specific file parser.

Separate methods are defined for processing application specific input information. Each method starts by looping over the input file and keeps reading new lines until a respective keyword is encountered. Once a particular keyword is found, all subsequent lines are read in sequential order and each line is checked if the number of arguments is according to definitions, otherwise a corresponding exception is thrown. Parsing a keyword section is completed when an empty line is encountered.

This process is repeated for each method invoked and each keyword defined in the application specific file parser. While generally the names of the methods invoked remain unchanged, the contents – applicable keywords – must be adjusted for each application and the looping over the input file is carried out for each of the keywords defined.

**Method *parseElements*** is defined for parsing application elements of the FEM model. Keywords **trusses**, **beams** and **hingedBeams** are defined for that purpose. It throws an exception if a given element identifier already exists or if the number of arguments is not equal to 5 or 6.

Keyword **trusses** and **beams** require input of the element id, two node identifiers, one cross section identifier and one material identifier.

Keyword **hingedBeams** requires input of the element id, 2 node identifiers, 1 cross section identifier, 1 material identifier and 1 for the hinge location which is either “f” for first or “s” for second node.

**Method *parseCrossSections*** is defined for parsing cross section definitions for application elements of the FEM model. It throws an exception if a given cross section identifier already exists or if the number of arguments is not equal to 2 or 3.

Keyword **crossSections** requires input of the cross section id, the cross sectional area and optionally the moment of inertia  $I_{xx}$  of that cross section.

**Method *parseMaterials*** is defined for parsing material definitions for application elements of the FEM model. It throws an exception if a given material identifier already exists or if the number of arguments is not equal to 2, 3 or 4.

Keyword **material** requires input of the material id, the modulus of elasticity (Young’s modulus)  $E$  and optionally for Poisson’s ratio  $\nu$  and the specific mass  $m$ .

Keyword **springProperties** requires input of the spring property id, the elasticity in x-direction  $K_x$ , the elasticity in y-direction  $K_y$  and the rotational elasticity  $K_\phi$ .

**Method *parseNodeLoads*** is defined for parsing nodal load definitions for application elements of the FEM model. It throws an exception if a given node load identifier already exists or if the number of arguments is not equal to 4 or 5.

Keyword **nodeloads** requires input of the node load id, the id of the node the load is applied to, the force in x-direction  $F_x$  and the force in y-direction  $F_y$  and optionally for an additional rotational moment  $M$ .

**Method *parsePointLoads*** is defined for parsing point load definitions for beam elements of the FEM model. It throws an exception if a given point load identifier already exists or if the number of arguments is not equal to 5.

Keyword **pointLoads** requires input of the point load id, the identifier of the element a point load is applied to, the force in x-direction  $F_x$ , force in y-direction  $F_y$  and the offset in normalized coordinates from the start node.

**Method *parseLineLoads*** is defined for parsing line load definitions for beam elements of the FEM model. It throws an exception if a given line load identifier already exists or if the number of arguments is not equal to 6 or 7.

Keyword **lineLoads** requires input of the line load identifier, the identifier of the element the line load is applied to, the force at the start node in x-direction **p1x**, the force at the start node in y-direction **p1y**, the force at the end node in x-direction **p2x**, the force at end node in y-direction **p2y** and optionally a flag (true/false) for local coordinates.

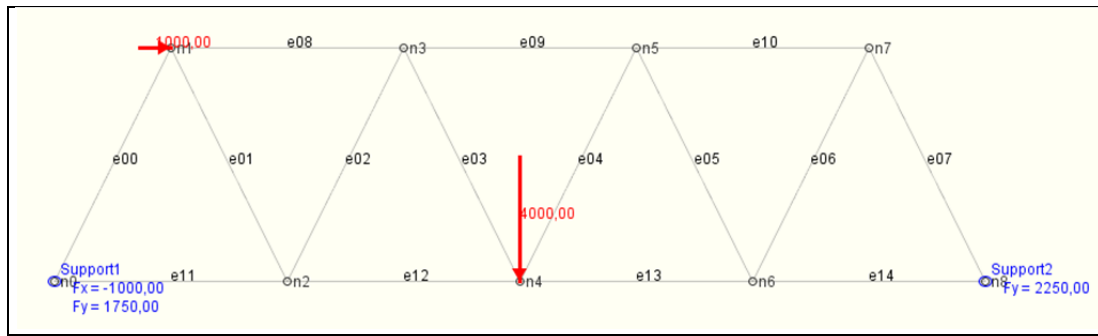
**Method *parseSupports*** is defined for parsing boundary (support) conditions of the FEM model. It throws an exception if a given support identifier already exists.

Keyword **supports** requires input of 6 arguments for the identifier of the support condition, the identifier of a supported node, a string defining predefined conditions in x-, y- or rotational direction ("xyr"), a predefined support value in x-direction, a predefined support value in y-direction and a predefined support value in rotational-direction.

Keyword **springSupports** requires input of 3 arguments for the identifier of the spring support, identifier of a supported node and the identifier for a spring property to be applied.

**Method *parseTimeInt*** is used for parsing information for time integration of an FEM model. Keywords **timeIntegration**, **damping**, **initialDeformations**, **initialVelocities**, **forcingFunction** and **timeDependentNodeLoad** are defined for these purposes.

This is demonstrated for an example of linear structural analysis of a truss.



e.g.	<b>identifier</b> truss bridge  <b>dimensions</b> 2      2  <b>nodes</b> n0    0.    0. n1    1.    2. n2    2.    0. n3    3.    2. n4    4.    0. n5    5.    2. n6    6.    0. n7    7.    2. n8    8.    0.	<b>trusses</b> e00   n0    n1    c1    iso e01   n1    n2    c1    iso e02   n2    n3    c1    iso e03   n3    n4    c1    iso e04   n4    n5    c1    iso e05   n5    n6    c1    iso e06   n6    n7    c1    iso e07   n7    n8    c1    iso e08   n1    n3    c1    iso e09   n3    n5    c1    iso e10   n5    n7    c1    iso e11   n0    n2    c1    iso e12   n2    n4    c1    iso e13   n4    n6    c1    iso e14   n6    n8    c1    iso  <b>crossSections</b> c1    0.001  <b>materials</b> iso   210000000.  <b>nodeLoads</b> load1 n1    1000.0    0.0 load2 n4        0.   -4000.0 <b>supports</b> Support1   n0    xy Support2   n8    y
------	--	---



First, the external Unicode-file must contain the keyword **identifier** followed by a unique string id describing the contents of the model.

In the example above, **dimensions** are specified with 2 spatial dimensions and 2 degrees of freedom per node, **nodes** are defined with the external identifiers *n0* to *n8* and corresponding 2-dimensional coordinates. Elements **trusses** are defined with the external identifiers *e00* to *e14* and material *iso*. **cross sections** are defined with the identifier *c1* and a double value for the specific property, e.g area. **materials** are defined with the identifier *iso* and a double value for the specific property, e.g conductivity. **nodeLoads** are defined with the identifier *load1* and *load2* and finally **supports** are defined with the identifier *Support1* and *Support2* an identifier for a node the load is applied to and a string for directions constrained (e.g. *xy*, *y*).

Relations to other objects are defined by referring to the corresponding external string identifiers. For the example, element *e00* is composed of nodes *n0* and *n1* and has cross section *c1* and material *iso*. Node loads are associated with node *n1* and *n4* and supports with nodes *n0* and *n8*.

Using external identifiers for establishing object relations has the considerable advantage that these relations may be established without the necessity that all objects referred to must have been generated before. Initially, all relations are established solely on the basis of the external string identifiers without any knowledge about internal storage allocation (internal references). Later, when starting the application, a link between external identifiers and internal references must be generated and used for referring to the actual internal addresses of the objects.

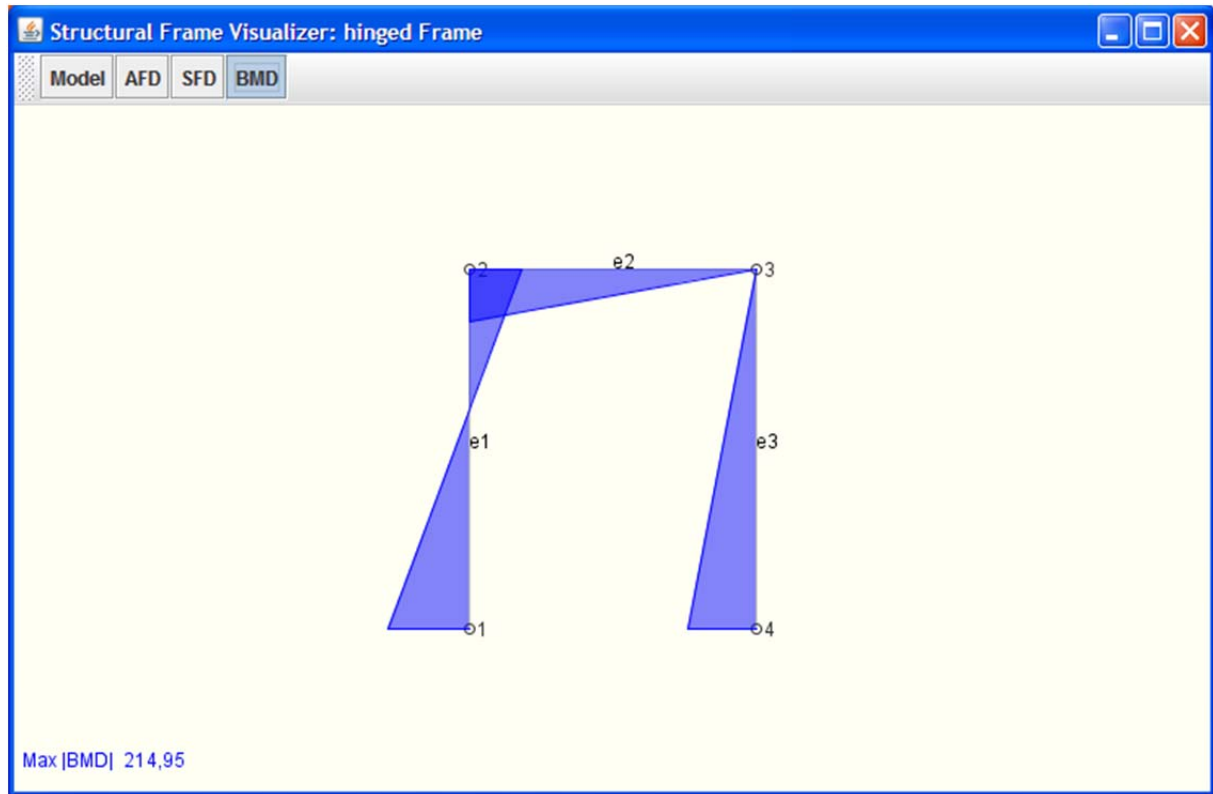
In the context of the FEM Framework this approach allows to strictly separate between topology and geometry. The topological model (elements referring to nodes that are only defined by name yet, loads, supports) may be defined completely separate from the geometrical model (nodes with coordinates).

The persistent data structure is stored in an Unicode-file as outlined above. A file parser is provided for interpreting the contents of the persistent data structure and for generating the corresponding transient structure. The transient data structure is based upon Java container structures. This is accomplished in the following way.

Modelling hinges in frame analysis is accomplished following in two different ways.

First, a full hinge is modelled by reducing the nodal degrees of freedom for the particular node with a hinge and by attaching two hinged beams to that node.

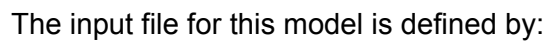
In the following example, node 3 is a fully hinged node:



The input file for this model is defined by:

<pre> dimensions 2      3  nodes 1      0.0    0.0 2      0.0    5.0 2 3      4.0    5.0 3 4      4.0    0.0  beams e1     1      2    c0 E I c  hingedBeams e2     2      3    c0 E I c    s e3     3      4    c0 E I c    f </pre>	<p>2 spatial dimensions, 3 nodal degrees of freedom</p> <p>number of nodal degrees of freedom reduced to 2 for node 3 number of nodal degrees of freedom set back to 3</p> <p>beam from node 1 to node 2</p> <p>hinged beam from node 2 to 3 with hinge at 2<sup>nd</sup> node hinged beam from node 3 to 4 with hinge at 1<sup>st</sup> node</p>
---	---

In the following example, a hinged beam (e4) is attached to a stiff node 3:

19

In case of **time integration problems** additional keywords are needed for entering the number of eigenstates to be considered, time integration parameters, initial displacements and velocities and an optional forcing function.

**Method *parseEigenstates*** is defined in the framework for parsing information for the solution of the eigenproblem of a system. It throws an exception if a given eigenstate identifier already exists or if the number of arguments is not equal to 2.

Keyword **eigenstates** requires input of an identifier for the specific eigenstate problem and an integer number for the number of eigenstates to be considered.

**Method *parseTimeInt*** is defined for parsing information for step-wise time integration of a structural FEM model defined by a 2<sup>nd</sup> order differential equation. It throws an exception if a given support identifier already exists.

Keyword **timeIntegration** requires input of 6 arguments for the identifier of time integration, the maximum time for the integration, the time step, the method to be used (Newmark, Alfa, Wilson-Theta), the parameter 1 for the specific method and the parameter 2 for the specific method.

Keyword **damping** requires input of an unspecified number of arguments for the values of a diagonal damping matrix. While this obviously only viable for small problems, it is mainly meant for specification of modal damping in mode superposition.

Keyword **initialDeformations** requires input of 2 arguments for the number of the applicable degree of freedom and its initial displacement  $\mathbf{d}_0$ . The initial displacement vector is initialized to zero meaning that no input is required for all degrees of freedom with zero initial displacement.

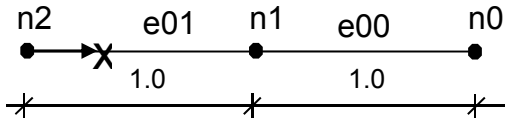
Keyword **initialVelocities** requires input of 2 arguments for the number of the applicable degree of freedom and its initial velocity  $\mathbf{v}_0$ . The initial velocity vector is initialized to zero meaning that no input is required for all degrees of freedom with zero initial velocity.

Keyword **forcingFunction** requires input of 1 argument for the name of the file containing the values for the force function matrix.

Keyword **timeDependentNodeLoad** requires input of 3 or 4 arguments for the node load id, the id of the associated node, the nodal degree of freedom where the load is applied to and optionally an indicator if the load is a ground load. Two different types of loading may then be specified in a following line for either harmonic excitation with amplitude, frequency and phase angle or piece-wise linear load variation defined for a set of pairs (time, value), where the values in structural dynamics typically contain accelerations.

The specific case of earthquake (ground) excitation is commonly handled by setting up a corresponding forcing function. The general problem of time-varying boundary conditions (accelerations) is restricted to the inertial forces  $-\mathbf{M}\mathbf{a}_g$  where  $\mathbf{M}$  is the mass matrix and  $\mathbf{a}_g$  is the ground acceleration at the corresponding degrees of freedom.

This is demonstrated for a simple truss consisting of 2 elements (e00, e01) with 2 different cross sections (c0, c1) and 2 different materials (iso0, iso1).

e.g.	<p><b>identifier</b> 6DOF Ground Excitation</p> <p><b>dimensions</b> 2      2</p> <p><b>nodes</b>  n0    2.    0.  n1    1.    0.  n2    0.    0.</p> <p><b>trusses</b>  e00   n0    n1    c0    iso0  e01   n1    n2    c1    iso1</p> <p><b>crossSections</b>  c0    0.01  c1    1.</p> <p><b>materials</b>  iso0   1.    0.    2.  iso1   1.    0.    1.98</p> <p><b>supports</b>  Support1    n2    xy  Support2    n1    y  Support3    n0    y</p> <p><b>eigenstates</b>  2DOFEigen   2</p> <p>// id, tmax, dt, method, par1, par2</p> <p><b>timeIntegration</b>  2DOFGround 125. 0.4 1 0.25 0.5</p> <p><b>ForcingFunction</b>  SixDOFGroundExcitation.for</p>	<p><b>example:</b></p> <p>Two 2-dimensional elements (trusses) e00 and e01, defined by three nodes (n0, n1, n2), two cross section definitions (c0, c1) and two material definitions (iso0, iso1).  The cross sectional area c0=0.01 and c1=1  Material definition iso0 is defined with elasticity=1., Poisson ratio=0. and specific mass=2.  Material definition iso1 is defined with elasticity=1., Poisson ratio=0. and specific mass=1.98</p> $.01 \ddot{x}_0 + .01 x_0 - .01 x_1 = f(t)$ $\ddot{x}_1 - .01 x_0 + 1.01 x_1 = 0$ <p>or</p> $\begin{bmatrix} .01 & 0 \\ 0 & 1. \end{bmatrix} \begin{Bmatrix} \ddot{x}_0 \\ \ddot{x}_1 \end{Bmatrix} + \begin{bmatrix} .01 & -.01 \\ -.01 & 1.01 \end{bmatrix} \begin{Bmatrix} x_0 \\ x_1 \end{Bmatrix} = \begin{Bmatrix} f(t) \\ 0 \end{Bmatrix}$ 
------	--	--

The number of eigenstates (eigenvalues, eigenvectors) to be computed is given by 2. The integration parameters consist of the maximum time for the integration, the time step, Newmark method (1), and the integration parameters  $\beta$  and  $\gamma$ .

The forcing function is defined in file "SixDOFGroundExcitation.for".

### 3.3 Model information for the analysis

Model information for the analysis of problems in structural analysis (beam theory) consists of the Finite Element approximation defining the element equations, the influences on the model (loads), the constitutive relations (material properties) and the external support conditions.

Furthermore, a specific element must be chosen that is appropriate for the problem to be solved defining the geometry of the problem (element geometry) and the element characteristics for the model.

Each application that is defined in the context of the FEM Framework simply needs to implement the additional application specific functionality for

- the element matrices,
- the material properties,
- the external influences on the model,
- the support conditions for the model and
- the time integration information.

These classes will be used by the parser in order to generate application specific objects.

#### 3.3.1 establishing element matrices

Functionality that is required for 2-dimensional linear truss elements in structural analysis is collected in **class Truss**.

Class **Truss** defines static attributes for the number of nodal degrees of freedom, the stiffness matrix, the mass matrix, the element deformations and the vector of the truss end forces.

The corresponding classes will be used by the parser in order to generate application specific objects.

A constructor is defined for generating new truss elements on the basis of identifiers for a new truss element, element nodes, cross section, material and the number of nodal degrees of freedom.

Class Truss is a simple element for linear truss analysis in two dimensions. It will be derived from *abstract class AbstractLinear2Node2D*.

Evaluation of the element matrix starts with the computation of the geometry, evaluation of the factor  $\left(\frac{EA}{L}\right)_e$  and the shape function vector for each truss element. This is followed by the computation of the corresponding element matrix:

$$\int_l S_e(x)(EA)_e S_e^T(x) dx$$

The diagonal mass matrix is stored in vector format. It is defined by the product of the specific mass with half the volume of the element for each node in each direction:

$$\text{diagonal terms of } M_e = m_e * \frac{1}{2} (A_e * L_e)$$

The behaviour of truss elements is defined by the computation of end forces and element deformations. Evaluation of end forces starts with re-computation of element geometries, computation of element displacements in local coordinates and evaluation of:

$$F_e = \left( \frac{EA}{L} \right)_e (u_0 - u_1)_e$$

Functionality that is required for 2-dimensional linear beam elements in structural analysis is collected in **class Beam** and **class HingedBeam**.

Class **Beam** defines static attributes for the stiffness matrix, the mass matrix, the vector of the beam end forces, the element deformation vector, the shape function vector, the load vector and the value of the Gauss integration point.

The corresponding classes will be used by the parser in order to generate application specific objects.

A constructor is defined for generating new beam elements on the basis of identifiers for a new beam element, element nodes, cross section, material and the number of nodal degrees of freedom.

Class Beam is defined for linear beam analysis in two dimensions. It will be derived from *abstract class AbstractLinear2Node2D*.

Evaluation of the element matrix starts with the computation of the local matrix

$$\int_l B_e^T(x)(EI)_e B_e(x) dx$$

which is subsequently transformed to global coordinates.

The diagonal mass matrix is stored in vector format. It is defined by the product of the specific mass with half the volume of the element for each node in each direction:

$$\text{diagonal terms of } M_e = m_e * \frac{1}{2} (A_e * L_e)$$

The element load vector is evaluated for element point loads or line loads depending on the type of abstract element load given as parameter. Loads may either be defined in global or in local element coordinates which is indicated by a flag in the parameter list. Interpolation of loads is carried out on the basis of the shape function vector at a specific point.

The behaviour of beam elements is defined by the computation of end forces and element deformations. Evaluation of end forces starts with computation of the local element matrix followed by computation of element deformations in local coordinates. End forces are then computed with contributions from nodal element deformations and contributions from point and line loads for a beam element.

Class **HingedBeam** is derived from class **Beam**. It distinguishes between a hinge at the start (*FIRST*) or at the end node (*SECOND*) and computes the corresponding element matrix via static condensation of the rotational degree of freedom released from the system. For this purpose a “type” attribute is added to the constructor as an integer value.

### 3.3.2 defining material properties

The material description in structural analysis is given in general by the elasticity matrix  $E$ . In beam theory material properties required for evaluation of the element matrix are:

$E$	Young’s modulus of elasticity and
$\nu$	Poisson’s ratio

For evaluation of the element mass matrix the additional value required is:

$m$	specific mass
-----	---------------

**Class *Material*** is contained in package *structuralAnalysis.model* and imports class *model.implementation.AbstractMaterial*. It is derived from class *AbstractMaterial*.

It consists of three constructors for passing the material identifier to the model and for storing the material property under names “*emodulus*”, “*poisson*” and “*mass*” and it provides methods for retrieving the material properties (*getModulusOfElasticity*, *getPoissonRatio* and *getMass*).



	<pre> package structuralAnalysis.model;  import model.implementation.AbstractMaterial;  public class Material extends AbstractMaterial {     public Material(String id, double _emodulus, double _poisson, double _mass ) {         super(id);         set("emodulus",_emodulus);         set("poisson",_poisson);         set("mass",_mass);     }     public Material(String id, double _emodulus, double _poisson ) {         super(id);         set("emodulus",_emodulus);         set("poisson",_poisson);     }     public Material(String id, double _emodulus) {         super(id);         set("emodulus",_emodulus);     } } </pre>
--	---

### 3.3.3 defining external influences on the system

External influences on the model can either be defined as

- **nodal influences**  
are concentrated loads at the nodes of the element mesh. The intensity of the load is given. The intensity is positive in positive coordinate directions.
- **line influences**  
are line loads between two neighbouring nodes of the element mesh. The line load intensities are defined. If the line load intensity varies linearly along the line, the intensities at the end nodes are given.

Class **NodeLoad** contains the implementation of **nodal load influences**. It is derived from class *AbstractNodeLoad*. Constructors are defined by specifying an identifier for the node load and for the node the load is applied to and by specifying the values for load intensities at the node in x-and y-direction (**p<sub>x</sub>**, **p<sub>y</sub>**) and optionally the rotational moment **M**.

Class **LineLoad** contains the implementation of **line load influences**. It is derived from class *AbstractElementLoad*. Constructors are defined by specifying an identifier for the node load and for the element the load is applied to and by specifying the values for load intensities at the start and end node in x-and y-direction (**p<sub>x</sub>**, **p<sub>y</sub>**) either in local or global coordinates

### 3.3.4 defining support conditions

Support conditions for the model can be defined for each node. Since nodes in structural analysis have three degrees of freedom, the prescribed values at a node need to be defined and the system status vector needs to be adjusted.

**Class Support** is contained in *package structuralAnalysis.model*. It is derived from class *AbstractSupport* and needs to implement interface *ISupport*.

Each support object provides attributes for the types of support conditions (x-direction, y-direction or rotation restrained), the vector of reaction forces, the vector of prescribed deformations, the vector of time varying deflections and a boolean vector indicating locations of element restraints.

The constructor is defined by 2 string identifiers for the support and the associated node, by an integer value for the support type and by a vector of predefined nodal deformations.

The support type is defined for x-direction, y-direction and rotation restraint with

1 **x**(direction) fixed, 2 **y**(direction) fixed, 3 **xy** fixed, 4 **r**(otation) fixed,  
5 **xr** fixed, 6 **yr** fixed and 7 **xyr** fixed.

Interface *ISupport* requires implementation of methods *setReactions*, *getNode*, *getRestraint*, *getPrescribed*, *getTimeDependent*, *getTimeVariation* and *getNodeId*.

### 3.3.5 defining information for time integration

Time integration information for the model can be defined for different time integration objects. Time integration objects are defined to store all the relevant information for a specific time integration problem in the model.

**Class TimeInt** is contained in *package structuralAnalysis.model*. It is derived from class *AbstractTimeInt2nd* which needs to be imported from *package framework.model.abstractclasses*.

It defines attributes for the time step, the integration parameters depending on the method used, the diagonal damping matrix, the displacement and velocity matrix, an indicator for the method to be used ( 1=Newmark, 2=Wilson-Theta, 3=Alpha) and the forcing function.

```
package structuralAnalysis.model;

import java.util.Scanner;
import framework.model.AppObject;
import framework.model.interfaces.ITimeInt2nd;

public class TimeInt extends AbstractTimeInt2nd {
    private double    tmax, dt, parameter1, parameter2;
    private double[]   damping;
    private double[] [] displacement, velocity;
    private int        method;
    private double[] [] forceFunction;
    Scanner tastatur = new Scanner(System.in);
```

	<pre> // ....Constructor diagonal mass and no damping public TimeInt(String id, double _tmax, double _dt, int _method, double _parameter1,                 double _parameter2, double[ ][ ] _displacement,                 double[ ][ ] _velocity, double[ ][ ] _forceFunction)  // ....Constructor diagonal mass and damping public TimeInt(String id, double _tmax, double _dt, int _method, double _parameter1,                 double _parameter2, double[ ] damping, double[ ][ ] _displacement,                 double[ ][ ] _velocity, double[ ][ ] _forceFunction) </pre>
--	---

Time integration information for the model can be defined for different time integration objects each having a specific identifier (id). Time integration objects are defined to store all relevant information for a specific time integration problem in the model.

Two constructors are provided, one defining the time integration id, the time step, the method to be used, parameter1 and parameter2 for the specific method, the displacement matrix, the velocity matrix and the forcing function matrix. The second constructor will define an additional diagonal damping matrix.

Displacement, velocity and forcing function matrices will store nodal values for all nodes at each time step while the velocity matrix only stores values at the current and the immediately preceding step.

### 3.4 Output of system results

Output functionality as a result of the analysis of 2-dimensional linear problems in structural analysis is implemented in three different classes, one for alphanumeric console output (*StructuralFrameOutputConsole*) and two other ones (*VisualStructuralModel*) and (*VisualStructuralDynamics*) for output in a graphical frame (window Structural Frame Visualizer) created with Java Swing technology

#### 3.4.1 Alphanumeric output of results

**Class *StructuralFrameOutputConsole*** is contained in package *structuralAnalysis.post*. It requires to import general Java functionality (*interface Iterator* of the Java Collections Framework) for iterating Java collections.

It further needs to import functionality of the FEM Framework for the model, i.e. *FEMException*, *Node* and interface *OutputAdapterConsole*. It also needs to import general utility functionalities for input and output of information.

It finally, needs to import the specific FEM components of the application model, i.e. *element*, *load*, *material*, *support*, *eigensolution* and *time integration*.

Class *StructuralFrameOutputConsole* only has attributes object *iter* of interface *Iterator* and flags for the type of analysis performed. Method *output* will control the flow of output of information interactively requested by the user.

If the model has not been analyzed yet, method *output* will present five different options for the output of the **m**(odel), the **i**(nfluences), the **v**(isualization) of the model, for **a**(nalyzing) the model and for **q**(uitting) operation. Each option is selected by entering the first character of the option on the console. If the models has been analyzed already, method *printBehaviour* is called directly.

Method ***printModel*** will print out the node, element, support and material information of the FEM model analyzed. For these purposes it will instantiate objects of those classes.

It will present the user with the corresponding choices for printing node, element, cross section, material or support information or for terminating model output.

According to the choice selected it will print a header for the respective choice.

It will subsequently first traverse all nodes of the model and print out the nodal identifier and coordinates.

It will next traverse all elements of the model and print out the element identifier and the identifiers for the corresponding nodes, materials and cross sections.

It will next traverse all support conditions of the model and print out the support identifier, the identifier for the corresponding nodes and prescribed support values.

It will next traverse all cross sections defined in the model and print out the cross section identifier and the cross section values, i.e. area **A** and moment of Inertia **I<sub>xx</sub>**.

It will next traverse all materials defined in the model and print out the material identifier and the material values, i.e. modulus of elasticity, Poisson's ratio and specific mass.

Finally, the option for terminating model output is presented.

Method ***printInfluences*** will print out the nodal, line and element loads of the FEM model analyzed.

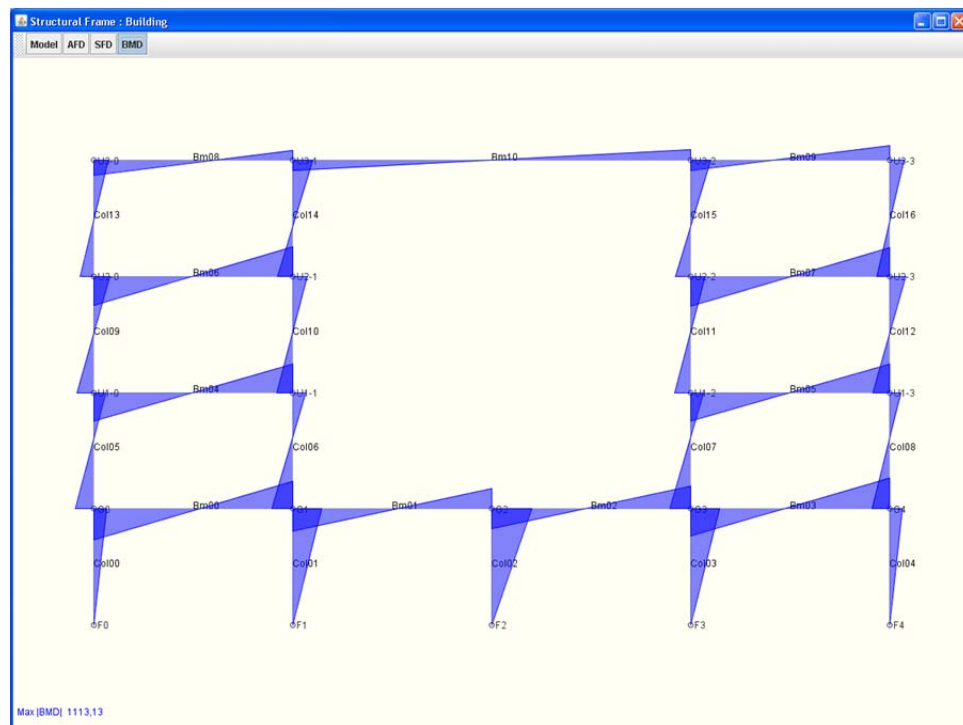
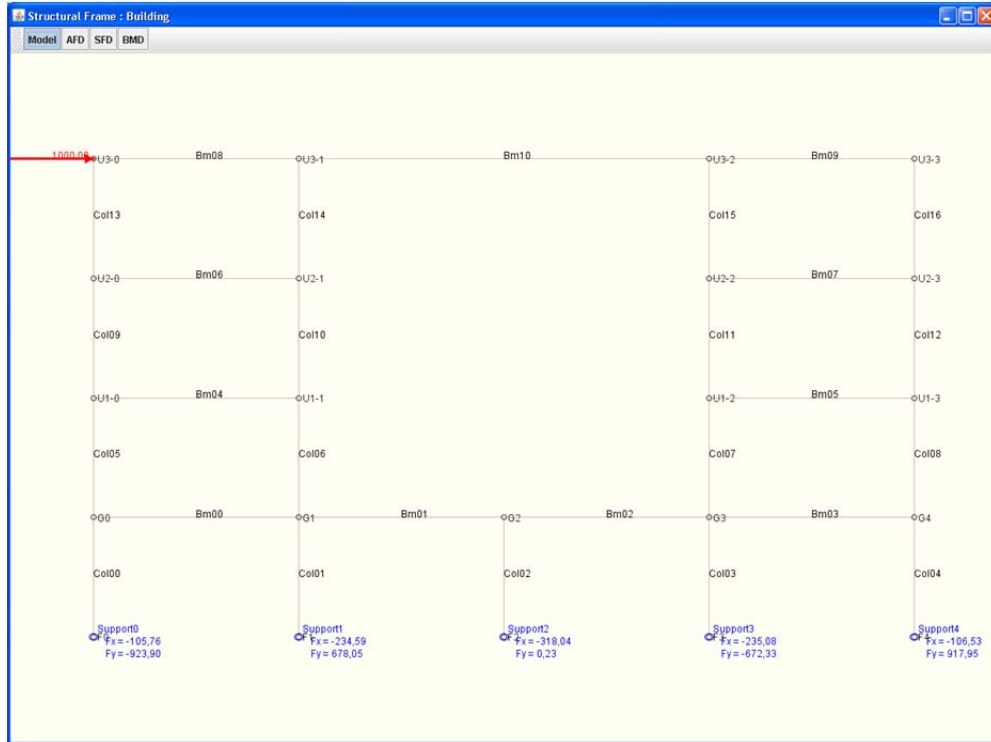
Method ***printBehaviour*** will present the user with the corresponding choices for printing nodal deformations, element forces, support reactions and for terminating output of results analyzed.

If the model has been solved, options are presented for displaying **n**(odal deformations), **f**(orces in elements), **r**(eaction) forces and **v**(isualization) of results. If in addition or alternatively, an eigensolution has been performed it presents option **e**(igensolution) for display of eigenvalues and eigenvectors. If finally time integration has been performed, it presents options **t**(ime history) and **p**(lot time history) for displaying the time history of nodal variables.

According to the choice selected it will print a header for the respective choice.

### 3.4.2 Graphical output of results

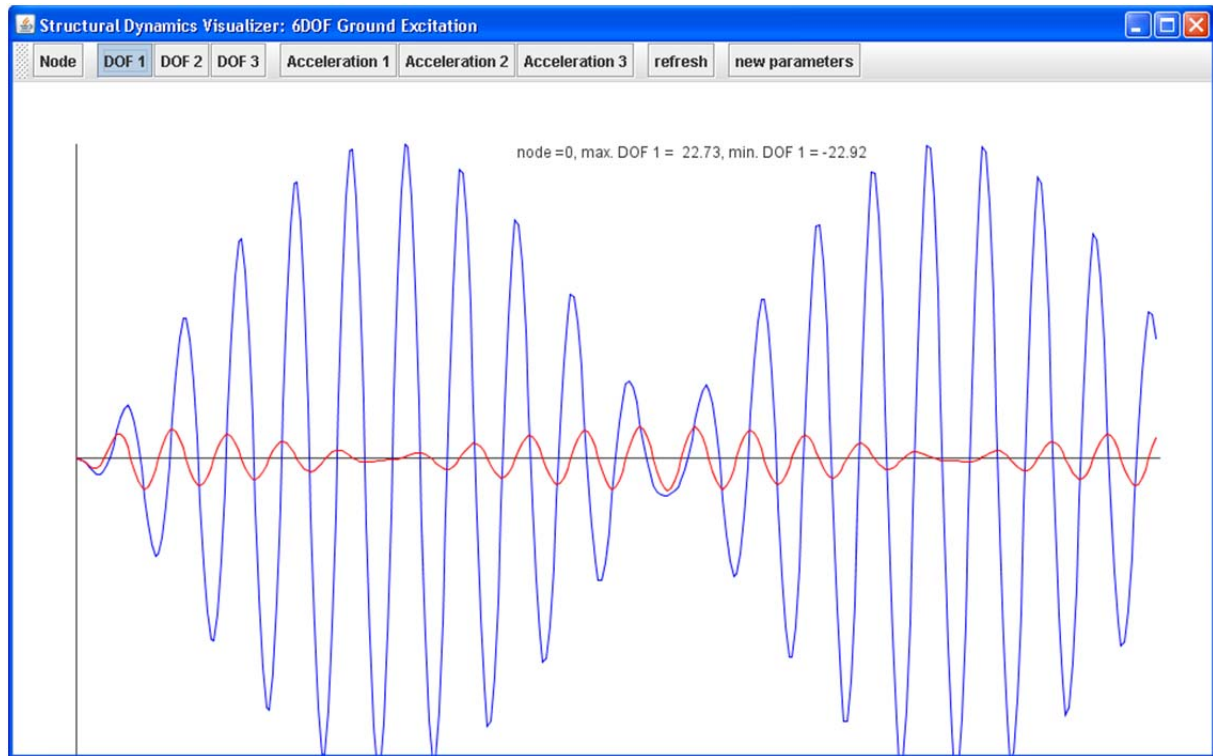
Class *VisualStructuralModel* is also contained in package structuralAnalysis.output. It is implemented exclusively on the basis of Java Swing technology. For the purposes of this text only the visualization of the input data will be demonstrated.



Class *VisualStructuralDynamics* is also contained in package structuralAnalysis.output.

Example output for the motion of two different nodes (DOF1 and DOF2):

Repeated evaluation and output of results is plotted on the same graph at the scale of the first curve. The “refresh” button will clear the curves and the “new parameters” button will allow for resetting of parameters for new time integration analysis.



## Literaturverzeichnis

- [1] The Finite Element Method, Its Basis & Fundamentals; O. C. Zienkiewicz, R. L. Taylor, J.Z. Zhu, 6<sup>th</sup> Edition, Elsevier 2005.
- [2] W. Wunderlich, G. Kiener, Statik der Stabtragwerke, Teubner Verlag 2004
- [3] Sun Microsystems, Inc., Copyright 2006: Download The Java Tutorial,  
<http://java.sun.com/docs/books/tutorial/>.
- [4] <http://www.cademia.org>, © 2006, Bauhaus-Universität Weimar | Lombego Systems
- [5] <http://www.opensource.org>