

# **Analysis of Heat Transfer Problems with the Finite Element Method (FEM)**

Bauhaus-Universität Weimar, Informatik im Bauwesen

2009

# Contents

1	Installation	3
2	Formulation of the FEM Solution	4
2.1	Finite Element Approximation for Heat Transfer	4
2.2	Linear 2-Node Elements for Stationary Heat Transfer	7
2.3	Linear 3-Node Elements for Stationary Heat Transfer	9
2.4	Linear 4-Node Elements for Stationary Heat Transfer	11
2.5	Linear 8-Node Elements in 3D for Stationary Heat Transfer	13
3	Implementation on the basis of the FEM Framework	15
3.1	Main programs for heat analysis:	17
3.2	Parsing persistent model information (input file)	19
3.3	Model information for the analysis	31
3.3.1	establishing element matrices	31
3.3.2	defining material properties	35
3.3.3	defining external influences on the system	36
3.3.4	setting up support conditions	40
3.3.5	defining information for time integration	40
3.4	Output of system results	41
3.4.1	Alphanumeric output of results	41
3.4.2	Graphical output of results	44
3.5	Integration into a CAD environment	48

# Analysis of Heat Transfer 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 FEM Solution

The analysis of problems in heat conduction 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.

### 2.1 Finite Element Approximation for Heat Transfer

The differential form of the **system equation** governing the stationary heat transfer of a two-dimensional continuum is given as follows (see for example: [1], [2]):

$$\frac{\partial f_0(\mathbf{x}_0)}{\partial \mathbf{x}_0} + \frac{\partial f_1(\mathbf{x}_1)}{\partial \mathbf{x}_1} + \frac{\partial f_2(\mathbf{x}_2)}{\partial \mathbf{x}_2} = \mathbf{q} \quad \text{for all } \mathbf{x} \in K$$

$$\text{or} \quad \text{div } \mathbf{f} = \mathbf{q} \quad \text{for all } \mathbf{x} \in K$$

where the heat balance in a body is described by vector **f** for the **heat state in a body** and by vector **q** for the **heat flow at the boundaries**.

**Temperature gradient g** can be calculated by the partial derivatives of the temperature T:

$$\mathbf{g} = \begin{pmatrix} \frac{\partial T}{\partial x_0} \\ \frac{\partial T}{\partial x_1} \\ \frac{\partial T}{\partial x_2} \end{pmatrix}$$

According to the mathematical definition of a gradient this equation can be written as:

$$\mathbf{g} = \text{grad } T \quad \text{for all } \mathbf{x} \in K$$

The relation between the heat state and the temperature gradient is defined by linear **constitutive conditions** with the definition that heat flows from regions with higher temperature to regions with lower temperature.

$$\mathbf{f} = -\mathbf{C} \mathbf{g} \quad \text{for all } \mathbf{x} \in K$$

$$\text{with: } \mathbf{C} = \begin{Bmatrix} \mathbf{c}_{00} & \mathbf{c}_{01} & \mathbf{c}_{02} \\ \mathbf{c}_{10} & \mathbf{c}_{11} & \mathbf{c}_{12} \\ \mathbf{c}_{20} & \mathbf{c}_{21} & \mathbf{c}_{22} \end{Bmatrix} \quad \text{heat conductivity matrix}$$

$$c_{ij} = c_{ij}(\mathbf{x}) \quad \text{in } \left[ \frac{\text{W}}{\text{m} \cdot \text{K}} \right] \quad \text{heat conductivity coefficient at position } \mathbf{x} \in K$$

The relation between heat state and heat flow through any cross section with normal  $\mathbf{n}$  at point  $\mathbf{x}$  is defined by:

$$\mathbf{q} = \mathbf{n}^T \mathbf{f} \quad \text{where } \mathbf{n} \text{ is the cross section normal vector}$$

The integral form of the **system equation** governing the stationary heat transfer of a two-dimensional continuum is given as follows (see for example: [3]):

$$\int_V -\delta \mathbf{g}^T \mathbf{f} dV = \int_V \delta \mathbf{t}^T \mathbf{q} dV - \sum_{K_u} \delta T_i q_i - \sum_{K_r} \delta T_i a_i$$

with:  $\mathbf{V}$  volume of the body,  $\delta \mathbf{g}$  variation of temperature gradient,  $\mathbf{f}$  internal heat state volumetric influences,  $\delta \mathbf{t}$  variation of temperature within an element,  $\mathbf{q}$  prescribed heat flow per unit of volume  
nodal influences,  $\mathbf{K}_u$  set of **unrestrained** nodes subjected to external influences (heat flow  $\mathbf{q}$ ),  $\delta \mathbf{T}$  is the variation of temperatures at the set of nodes  $\mathbf{K}_u$ ,  $q_i$  values of specified (known) external influences (heat flow) on set of nodes  $\mathbf{K}_u$   
system reactions (support),  $\mathbf{K}_r$  set of **restrained** nodes with prescribed system values (temperature  $\mathbf{T}$ ),  $a_i$  unknown support conditions (heat stream) resulting from prescribed temperatures at set of nodes  $\mathbf{K}_r$ .

Substituting the constitutive equations into the system equations above results in

$$\int_V \delta \mathbf{g}^T \mathbf{C} \mathbf{g} dV = \int_V \delta \mathbf{t}^T \mathbf{q} dV - \sum_{K_u} \delta T_i q_i - \sum_{K_r} \delta T_i a_i$$

**Discretization** over the volume results in two sums of element integrals  $\mathbf{V}_e$  over all elements:

$$\sum_e \int_{V_e} \delta \mathbf{g}_e^T \mathbf{C}_e \mathbf{g}_e dV_e = \sum_e \int_{V_e} \delta \mathbf{t}_e^T \mathbf{q}_e dV_e - \sum_{K_u} \delta T_i q_i - \sum_{K_r} \delta T_i a_i$$

Element temperatures and heat streams are conventionally approximated via **element shape function expressions**  $\mathbf{S}_e$  which may assume different forms depending on the choice of element:

$$\mathbf{t} \cong \sum_e \mathbf{S}_e \mathbf{T}_e \quad \text{with: } \mathbf{T}_e \text{ element vector of nodal temperatures}$$

$$\mathbf{q} \cong \sum_e \mathbf{S}_e \mathbf{q}_e \quad \text{with: } \mathbf{q}_e \text{ element vector of nodal heat flow}$$

The **temperature gradient** is defined by the equation

$$\mathbf{g} = \text{grad } \mathbf{T}$$

$$\text{with: } \frac{\partial T_e}{\partial x_i} = \frac{\partial}{\partial x_i} (\mathbf{S}_e^T \mathbf{T}_e)$$

$$\mathbf{g}_e = \mathbf{S}_{xe}^T \mathbf{T}_e$$

for linear **isotropic material** the heat conductivity matrix is defined by

$$\mathbf{C}_e = \begin{Bmatrix} c_e & 0 & 0 \\ 0 & c_e & 0 \\ 0 & 0 & c_e \end{Bmatrix} = c_e * \mathbf{I}$$

where:  $c_e$  is the element conductivity which is assumed to be constant and isotropic

Thus, the internal **heat flow** in a isotropic body is defined by

$$\mathbf{f}_e = -c_e * \mathbf{g}_e = -c_e * \mathbf{S}_{xe}^T \mathbf{T}_e$$

Substituting this information into the system equation above results in the following set of equations for a general formulation of stationary hat transfer:

$$\sum_e \delta \mathbf{T}_e^T \underbrace{\int_{V_e} c_e \mathbf{S}_{xe} \mathbf{S}_{xe}^T dV}_{\mathbf{k}_e} \mathbf{T}_e = \sum_e \delta \mathbf{T}_e^T \underbrace{\int_{V_e} \mathbf{S}_e^T \mathbf{q}_e dV}_{\mathbf{p}_{ve}} - \sum_{K_u} \delta T_i q_i - \sum_{K_r} \delta T_i a_i$$

$\mathbf{p}_{ue} \qquad \mathbf{q}_{re}$

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

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

The system matrix  $\mathbf{K}_s$  is multiplied by the vector of nodal temperatures  $\mathbf{T}_s$  referred to as primal vector of system unknowns or known (prescribed) values of system variables  $T$ .

The Right-Hand-Side (RHS) consists of the external influences on the system referred to as load vector  $\mathbf{p}_s$  with specified (known) values of heat streams inside the body and at specific nodes

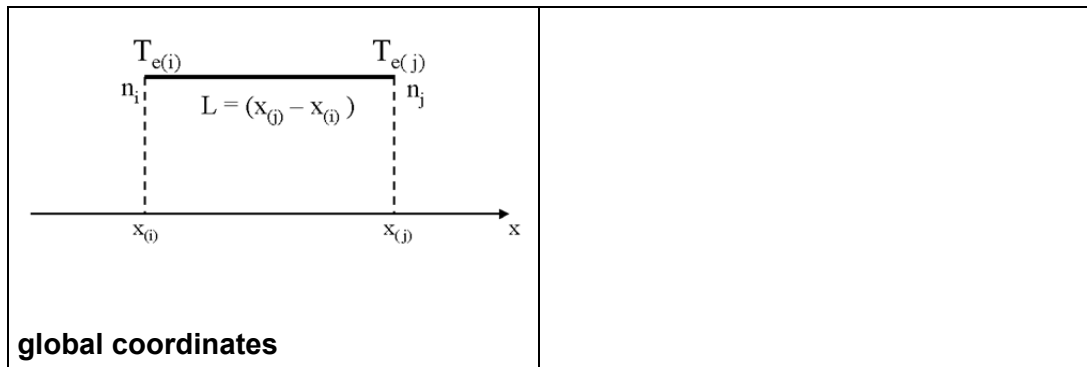
and of the support conditions on the system  $\mathbf{q}_s$  referred to as dual system vector with unknown values of heat streams resulting from predefined temperatures at restrained boundary nodes and all other values of the dual vector equal to 0.

## 2.2 Linear 2-Node Elements for Stationary Heat Transfer

Generally, a variety of elements and approximation functions may be chosen for approximating the geometry of the model and the functional behaviour of the model.

For the purposes of this text, linear 1-dimensional elements (heat rods) were chosen to approximate the element geometry and linear functions for the element characteristics in stationary heat transfer.

Physical problems of stationary heat transfer are defined by a single degree of freedom per node, i.e. the temperatures at the nodes ( $T_{e(i)}$ ,  $T_{e(j)}$ ) as primal unknowns).



The approximation of temperatures and heat streams within an element is accomplished by the following linear interpolation using the **shape function** in  $z$  for nodes (i,j):

$$\mathbf{t}(z)_e = \begin{bmatrix} 1-z & z \end{bmatrix} \begin{Bmatrix} T_{e(i)} \\ T_{e(j)} \end{Bmatrix}$$

$$\mathbf{t}(z)_e = \mathbf{s}_e(z)^T \mathbf{T}_e$$

The **topological mapping** of normalized onto corresponding global coordinates is defined by matrix  $\mathbf{R}_e$  which maps local field variables  $\mathbf{T}_e$  at nodes (i, j) to global field variables  $\mathbf{T}_s$ . This mapping is implicitly handled by the FEM Framework on the basis of the system indices generated for all nodal degrees of freedom. It does not need to be performed by the user. System indices refer to the global degrees of freedom associated with each node (e.g.  $T_i$ ) and elements with normalized coordinates are generated with reference to these nodes.

$$\begin{Bmatrix} T_{e(i)} \\ T_{e(j)} \end{Bmatrix} = \begin{Bmatrix} 0 & 0 & 0 & .. & 1 & .. & 0 & 0 & 0 & 0 \\ 0 & 0 & .. & 1 & .. & 0 & 0 & 0 & 0 & 0 \end{Bmatrix} \begin{Bmatrix} T_0 \\ .. \\ .. \\ T_i \\ .. \\ .. \\ T_n \end{Bmatrix}$$

$$\mathbf{T}_e = \mathbf{R}_e \mathbf{T}_s$$

Element matrices are determined by evaluating for each element the integrals:

$$\int_{L_e} c_e A_e \mathbf{S}_{xe} \mathbf{S}_{xe}^T dL$$

The shape function derivatives and the element geometry in general are totally independent of a specific application and are thus documented in the FEM-Framework documentation [4] and incorporated into the FEM-Framework implementation.

The element shape function derivative  $\mathbf{S}_{xe}$  and the value of the determinant are independent of the normalized coordinates  $z$ . The element conductivity  $c_e$  and cross sectional area  $A_e$  were also assumed to be constant. The integrals can therefore be evaluated analytically for each element.

$$\mathbf{k}_e = c_e A_e L_e \mathbf{S}_{xe} \mathbf{S}_{xe}^T = \frac{c_e A_e}{L_e} \begin{Bmatrix} 1 & -1 \\ -1 & 1 \end{Bmatrix}$$

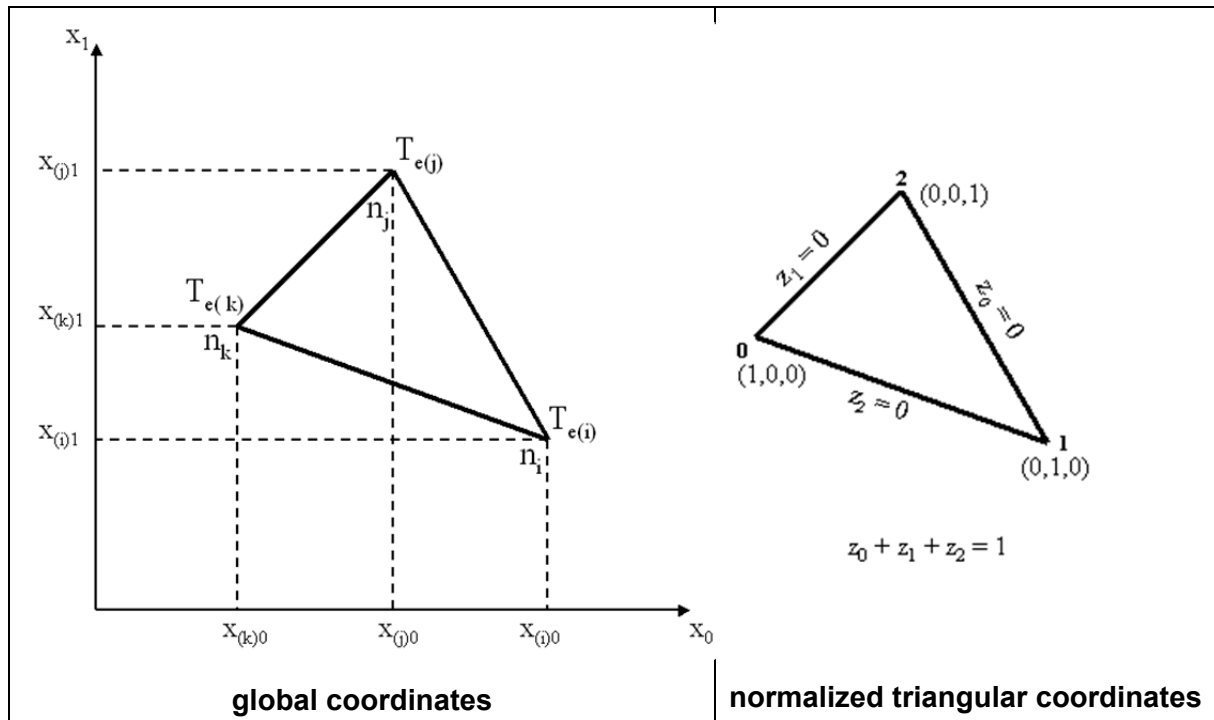


## 2.3 Linear 3-Node Elements for Stationary Heat Transfer

Generally, a variety of elements and approximation functions may be chosen for approximating the geometry of the model and the functional behaviour of the model.

For the purposes of this text, triangular elements were chosen to approximate the element geometry and linear functions for the element characteristics in stationary heat transfer. The element geometry is defined by deriving the implementation from the abstract implementation *AbstractLinear3Node2D*.

Physical problems of stationary heat transfer are defined by a single degree of freedom per node, i.e. the temperatures at the nodes ( $T_{e(i)}$ ,  $T_{e(j)}$ ,  $T_{e(k)}$ ) as primal unknowns).



The approximation of temperatures and heat streams within an element is accomplished by the following linear interpolation using the **shape functions** in triangular coordinates ( $z_0$ ,  $z_1$ ,  $z_2$ ) for all three nodes ( $i,j,k$ ):

$$\mathbf{t}(\mathbf{z})_e = \begin{Bmatrix} z_0 & z_1 & z_2 \end{Bmatrix} \begin{Bmatrix} T_{e(i)} \\ T_{e(j)} \\ T_{e(k)} \end{Bmatrix}$$

$$\mathbf{t}(\mathbf{z})_e = \mathbf{s}_e(\mathbf{z})^T \mathbf{T}_e$$

The **topological mapping** of normalized onto corresponding global coordinates is defined by matrix  $\mathbf{R}_e$  which maps local field variables  $\mathbf{T}_e$  at nodes ( $i, j, k$ ) to global field variables  $\mathbf{T}_s$ . This mapping is implicitly handled by the FEM Framework on the basis of the system indices generated for all nodal degrees of freedom. It does not need to be performed by the user. System indices refer to the global degrees of freedom associated with each node (e.g.  $T_i$ ) and elements with normalized coordinates are generated with reference to these nodes.

$$\begin{Bmatrix} T_{e(i)} \\ T_{e(j)} \\ T_{e(k)} \end{Bmatrix} = \begin{bmatrix} 0 & 0 & 0 & \dots & 1 & \dots & 0 & 0 & 0 & 0 \\ 0 & 0 & \dots & 1 & \dots & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \dots & 1 & \dots & 0 \end{bmatrix} \begin{Bmatrix} T_0 \\ \dots \\ \dots \\ T_i \\ \dots \\ \dots \\ \dots \\ T_n \end{Bmatrix}$$

$$\mathbf{T}_e = \mathbf{R}_e \mathbf{T}_s$$

Element matrices are determined by evaluating for each element the integrals:

$$\int_{\Lambda_e} c_e \mathbf{S}_{xe} \mathbf{S}_{xe}^T dA$$

In case of a linear triangle, the element shape function derivatives  $\mathbf{S}_{xe}$  and the value of the determinant are independent of the normalized coordinates  $z_i$  and thus constant at each point within the triangle. The element conductivity  $c_e$  was also assumed to be constant. The integrals can therefore be evaluated analytically for each element.

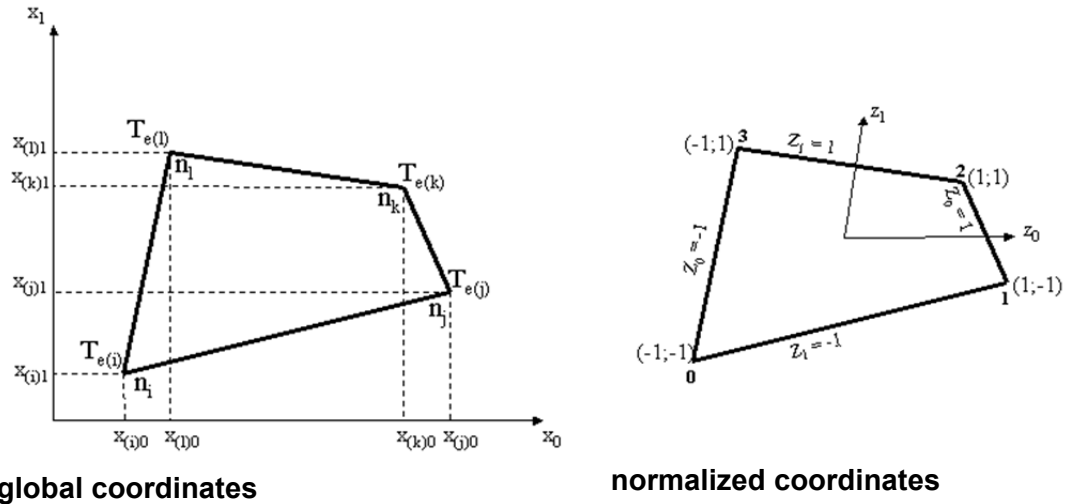
$$\mathbf{k}_e = c_e \mathbf{S}_{xe} \mathbf{S}_{xe}^T * \text{area}_e = 0.5 * \det \mathbf{X}_z c_e * \mathbf{S}_{xe} \mathbf{S}_{xe}^T$$

In general, integration of element matrices is accomplished numerically, i.e. by evaluating the individual contributions at each Gauss point, by summing up contributions for all Gauss points and by multiplying the results with the corresponding Gauss weights. Thus, evaluation at three Gauss point and application of Gauss weight 1/3 would result in the same constant value of the shape function derivatives and the determinant.

## 2.4 Linear 4-Node Elements for Stationary Heat Transfer

Elements with four nodes and a linear variation of coordinate functions may be chosen to approximate the element geometry and element characteristics in stationary heat transfer. The element geometry is defined by deriving the implementation from the abstract implementation *AbstractLinear4Node2D*.

Physical problems of stationary heat transfer are defined by a single degree of freedom per node, i.e. the temperatures at the nodes ( $T_{e(i)}$ ,  $T_{e(j)}$ ,  $T_{e(k)}$ ,  $T_{e(l)}$ ) as primal unknowns).



The approximation of temperatures and heat streams within an element is accomplished by linear interpolation using the **shape functions** in global coordinates ( $z_0$ ,  $z_1$ ) for all four nodes (0,1, 2, 3) in both directions (0,1):

$$\mathbf{t}(\mathbf{z})_e = 0,25 * \begin{Bmatrix} (1-z_0)(1-z_1) & (1+z_0)(1-z_1) & (1+z_0)(1+z_1) & (1-z_0)(1+z_1) \end{Bmatrix} \begin{Bmatrix} T_{e(i)} \\ T_{e(j)} \\ T_{e(k)} \\ T_{e(l)} \end{Bmatrix}$$

$$\mathbf{t}(\mathbf{z})_e = \mathbf{s}_e(\mathbf{z})^T \mathbf{T}_e$$

The **topological mapping** of normalized onto corresponding global coordinates is defined by matrix  $\mathbf{R}_e$  which maps local field variables  $\mathbf{T}_e$  at nodes (i, j, k) to global field variables  $\mathbf{T}_s$ . This mapping is implicitly handled by the FEM Framework on the basis of the system indices generated for all nodal degrees of freedom. It does not need to be performed by the user. System indices refer to the global degrees of freedom associated with each node (e.g.  $T_i$ ) and elements with normalized coordinates are generated with reference to these nodes.

$$\begin{Bmatrix} T_{e(i)} \\ T_{e(j)} \\ T_{e(k)} \\ T_{e(l)} \end{Bmatrix} = \begin{bmatrix} 0 & 0 & 0 & \dots & 1 & \dots & 0 & 0 & 0 & 0 \\ 0 & 0 & \dots & 1 & \dots & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \dots & 1 & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots & 1 & \dots & 0 & 0 & 0 \end{bmatrix} \begin{Bmatrix} T_0 \\ \dots \\ T_i \\ \dots \\ T_n \end{Bmatrix}$$

$$\mathbf{T}_e = \mathbf{R}_e \mathbf{T}_s$$

Element matrices are determined by evaluating for each element the integrals:

$$\int_{A_e} c_e \mathbf{S}_{xe} \mathbf{S}_{xe}^T dA$$

In case of a linear quadrilateral, the element shape function derivatives  $\mathbf{S}_{xe}$  and the value of the determinant are evaluated numerically by Gaussian integration. For this purpose, the values of matrix  $\mathbf{S}_{xe}$  are evaluated at corresponding Gauss points and multiplied by associated Gauss weights. Integration over the area of the elements is carried out in normalized coordinates:

$$dA = dx_0 dx_1 = \det \mathbf{X}_z dz_0 dz_1$$

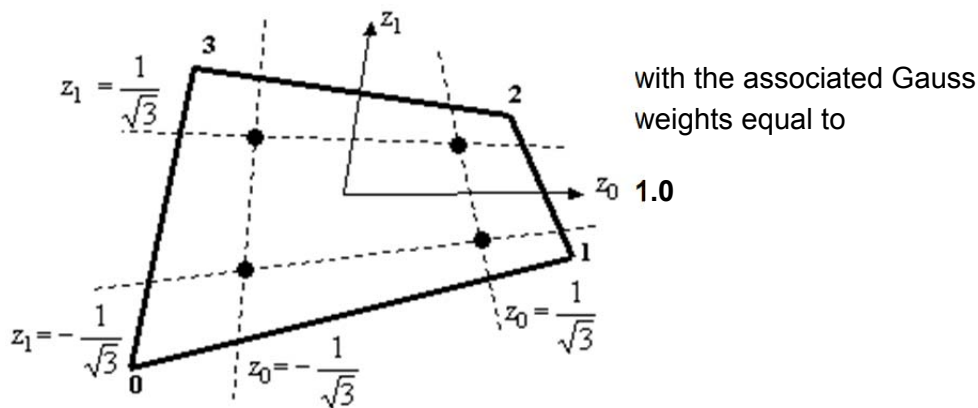
Thus, element matrices may be evaluated by the sum over the Gauss points for the product:

$$\mathbf{k}_e = \sum_{\text{Gauss points}} c_e * \mathbf{S}_{xe} * \mathbf{S}_{xe}^T * \det \mathbf{X}_z$$

where :  $c_e$  is constant over the element and

the values of  $\mathbf{S}_{xe}$  and  $\det \mathbf{X}_z$  are evaluated at each Gauss point

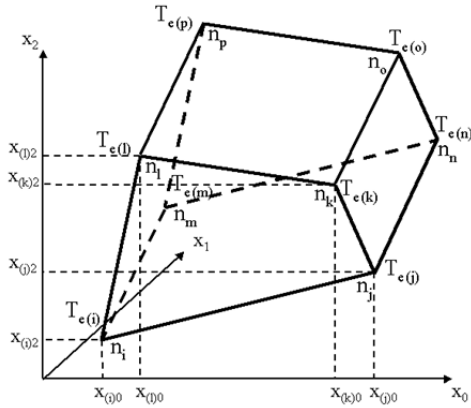
For the case of the linear element with 4 nodes, an order of integration of  $2 \times 2$  is sufficient. The corresponding Gauss points  $p(z_0, z_1)$  and Gauss weights are:



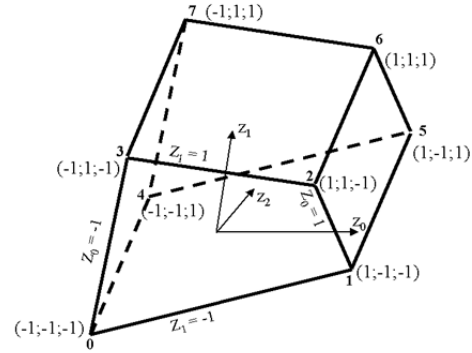
## 2.5 Linear 8-Node Elements in 3D for Stationary Heat Transfer

Elements with eight nodes and a linear variation of coordinate functions may be chosen to approximate the element geometry and element characteristics in 3-dimensional stationary heat transfer. The element geometry with the shape function definitions, the shape function derivatives and the Jacobian is defined by deriving the implementation from the abstract implementation *AbstractLinear8Node3D*.

Physical problems of stationary heat transfer are defined by a single degree of freedom per node, i.e. the temperatures at the nodes ( $T_{e(i)}$ , ....  $T_{e(p)}$ ) as primal unknowns).



global coordinates



normalized coordinates

The approximation of temperatures and heat streams within an element is accomplished by linear interpolation using the **shape functions** in global coordinates ( $z_0, z_1, z_2$ ) for all eight nodes (0 ... 7) in all three directions (0,1,2):

$$\mathbf{t}(z)_e = \mathbf{S}_e^T \mathbf{T}_e$$

where:  $\mathbf{T}_e$  is the vector of nodal temperatures at 8 nodes in 3 directions ( $t_{(i)0}, t_{(i)1}, t_{(i)2}$ ) and  $\mathbf{S}_e^T$  is the (3\*8) matrix of shape function vectors  $\mathbf{s}(z)$  in all 3 directions

The **topological mapping** of normalized onto corresponding global coordinates is defined by matrix  $\mathbf{R}_e$  which maps local field variables  $\mathbf{T}_e$  at the 8 nodes (i ... p) to global field variables  $\mathbf{T}_s$ . This mapping is implicitly handled by the FEM Framework on the basis of the system indices generated for all nodal degrees of freedom. It does not need to be performed by the user. System indices refer to the global degrees of freedom associated with each node (e.g.  $T_i$ ) and elements with normalized coordinates are generated with reference to these nodes.

$$\mathbf{T}_e = \mathbf{R}_e \mathbf{T}_s$$

Element matrices are determined by evaluating for each element the integrals:

$$\int_{V_e} \mathbf{S}_{xe} \mathbf{c}_e \mathbf{S}_{xe}^T dV$$

In case of a linear 8-node element, the element shape function derivatives  $\mathbf{S}_{xe}$  and the value of the determinant are evaluated numerically by Gaussian integration. For this purpose, the values of matrix  $\mathbf{S}_{xe}$  are evaluated at corresponding Gauss points and multiplied by associated Gauss weights. Integration over the area of the elements is carried out in normalized coordinates:

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

Thus, element matrices may be evaluated by the sum over the Gauss points for the product:

$$\mathbf{k}_e = \sum_{\text{Gauss points}} \mathbf{S}_{xe} * \mathbf{c}_e * \mathbf{S}_{xe}^T * \det \mathbf{X}_z$$

where :  $\mathbf{c}_e$  is a diagonal matrix with the value of conductivity for each direction  
and the values of  $\mathbf{S}_{xe}$  and  $\det \mathbf{X}_z$  are evaluated at each Gauss point

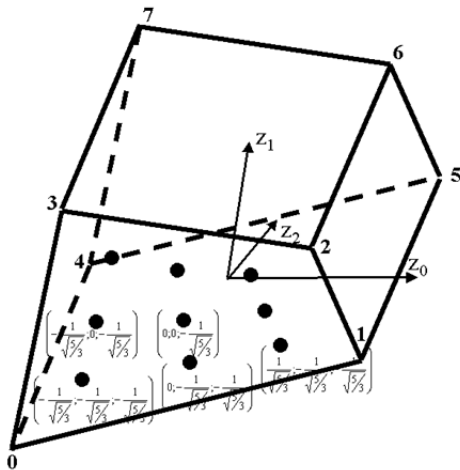
For the case of the linear element with 8 nodes, an order of integration of  $3 \times 3 \times 3$  is sufficient. The corresponding Gauss points  $p(z_0, z_1, z_2)$  are defined in all 3 directions by the 3 normalized coordinates:

$$\begin{matrix} -\frac{1}{\sqrt{5/3}} & 0 & \frac{1}{\sqrt{5/3}} \end{matrix}$$

and the corresponding Gauss weights are:

$$\begin{matrix} \frac{5}{9} & \frac{8}{9} & \frac{5}{9} \end{matrix}$$

This can be shown for the  $3 \times 3$  Gauss points in a plane in direction  $z_2 = -\frac{1}{\sqrt{5/3}}$  :



normalized coordinates

with the associated Gauss weights equal to

$$\begin{matrix} \frac{5}{9} & \frac{8}{9} & \frac{5}{9} \end{matrix}$$

in both directions

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

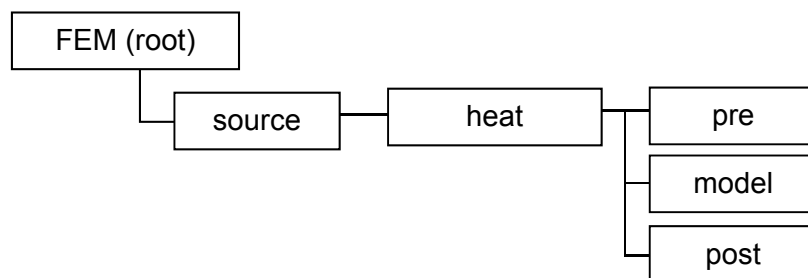
Applications of the FEM for the solution of physical problems are embedded in the file system usable as an Eclipse-Workspace. The root is named *FEM* with subdirectories for source code, binary data, input data and lecture notes. The FEM Framework is stored in Java archive format in the root directory as *FEM-Framework.jar* which can be used for compiling the application. Additional batch files are provided for for compiling and running the software without the IDE as simple console application.

Several applications could be stored under the hierarchy *source*. In case of this text, applications are provided for stationary and transient heat transfer in 2D or 3D.

Any application must define

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

Any new application is embedded into the hierarchical structure of the file system in the following way:

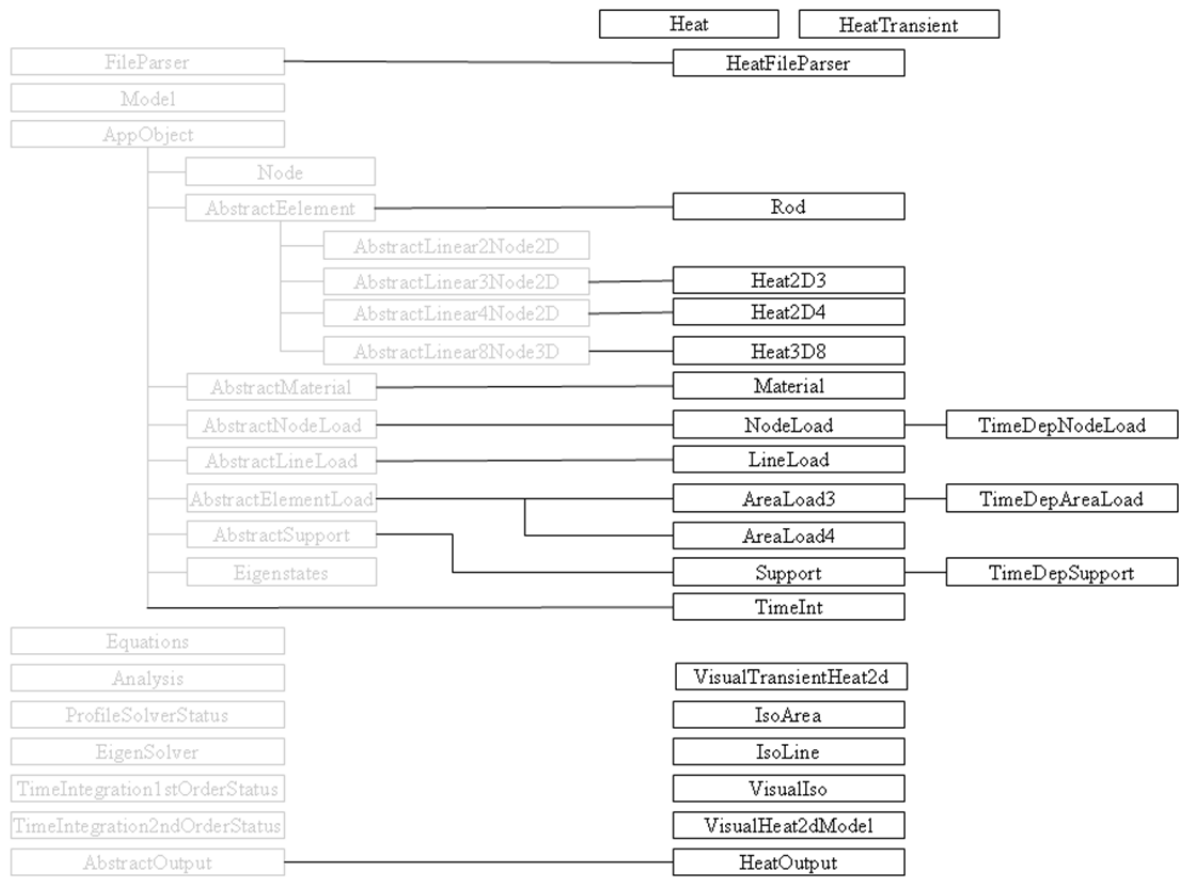


The implementation of classes defining a new application in case of heat transfer problems using FEM consists of a file parser for reading a new model, of the application specific element, material, load and support definitions, of the definitions for time integration of the model and of the respective 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 (Heat) on the basis of the Framework:





### 3.1 Main programs for heat analysis:

There are two main programs for the analysis of stationary heat transfer is implemented in class **Heat** and **HeatTransient**. Both are contained in package `heat`.

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 Heat** 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 *HeatOutput* is generated for the specific model to be analyzed. Method *output* of this object will allow for checking model input.

Next, a new analysis object is generated for the specific model and method *systemMatrix* 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 *HeatOutput* is used for output of results for the model analyzed.

```

public class Heat {
    private static Model model;
    private static Analysis analysis;
    private static HeatOutput output;

    public static void main(String args[]) {
        File file = FileHandling.getInputFile("./input/heat",
                                              "FEM input files", "inp");

        try {
            // setup and visualize model, analyze model and output results
            model = new HeatFileParser(file).getModel();
            analysis = new Analysis(model);
            output = new HeatOutput(model, analysis);
            output.outModel();

            // perform stationary analysis and output results
            analysis.computeSystemMatrix();
            analysis.computeSystemVector();
            analysis.solveEquations();
            output.outStationary();

        } catch (Exception e) {
            e.printStackTrace();
            System.err.println(e);
        }
    }
}

```

In case of a transient heat analysis, class **HeatTransient** is also contained in package *heat* 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 *HeatOutputConsole* constructor for enabling recalculation of time integration results in the *output* object with new parameters.

```

// perform transient analysis and output results
analysis.eigenstates();
analysis.setInitialStateFromStationarySolution();
analysis.timeIntegration1stOrder();
output.outTransient();

```

## 3.2 Parsing persistent model information (input file)

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

The functionality is contained in *package heat.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, influences, materials and supports and finally application specific functionality for transient heat analysis as there is 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 *HeatFileParser*.

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

```
//      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 1st 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 *HeatFileParser* defines additional keywords for heat analysis.

```
//      rods                rod element with 2 nodes
//                        4: name, node1, node2, material
//      heat2D3            triangular element in 2D with 3 nodes
//                        5: name, node1, node2, node3, material
//      heat2D4            quadrilateral element in 2D with 4 nodes
//                        6: name, node1, node2, node3, node4, material
//      heat3D8            quadrilateral element in 3D with 8 nodes
//                        10: name, node1, node2, node3, node4, node5, node6, node7,
//                             node8, material
//      3D8ElementArray    array of quadrilateral elements in 3D with 8 nodes
//                        4: elementInitial, nodeInitial, nIntervals, material

//      materials          2: name, conductivity
//                        3: name, conductivity, density(rho)*specificHeat(C)
//                        4: name, cx, cy, cz

//      nodeLoads          3: name, node, value
//      lineLoads          5: name, startNode, endNode, startValue, endValue
//      areaLoad3          area loads on triangular elements
//                        5: name, element, value1, value2, value3
//      areaLoad4          area loads on quadrilateral elements
//                        6: name, element, value1, value2, value3, value4

//      supports          3: name, node, value
//      supportFace        5: supportInitial, face, nodeInitial, nNodes, prescribed

//      timeIntegration    parameters for first order time integration
//                        4: id, tmax, dt, alfa
//      initialTemperatures initial temperatures at specified degrees of freedom
//                        2: dof, t0
//                        >2: initial temperatures at dof's starting from dof 0
//      temperatureBC      time varying temperature boundary conditions
//                        3: supportId, nodeId, file name with temperature variation
//                        5: supportId, nodeId, amplitude, frequency, phase angle
//                        >5: supportId, nodeId, startTime, startTemperature,
//                             [endTime, endTemperature]...
//      forcingFunction     read forcing function from specified file
//                        1: fileName
//      timeDependentAreaLoad constant influence over time on 2D triangular elements
//                        5: areaLoadId, elementId, value1, value2, value3
//      timeDependentNodeLoad forcing function from time dependent node loads
//                        3: nodeLoadId, nodeId, nodal degree of freedom
//                        4: nodeLoadId, nodeId, nodal degree of freedom, ground
//                        followed by a separate line of
//                        3: amplitude, frequency, phase angle or
//                        >3: startTime, startValue, [endTime, endValue]...
```

Class **HeatFileParser** provides a constructor 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 three 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 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.

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.

#### **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.

**Class *ElementParser*** is defined for parsing element components of the FEM model. It throws an exception if a given element identifier already exists or if the number of arguments is not in accordance with the keyword.

Keyword **rods** requires input of 4 arguments for the identifier of the rod element, the identifiers for the two nodes and the material identifier.

Keyword **heat2D3** requires input of 5 arguments for the identifier of the triangular element, the identifiers for the three nodes and the material identifier.

Keyword **heat2D4** requires input of 6 arguments for the identifier of the quadrilateral element, the identifiers for the four nodes and the material identifier.

Keyword **heat3D8** requires input of 10 arguments for the identifier of the box element, the identifiers for the eight nodes and the material identifier.

Elements may also be generated in form of an element array. For this purpose the keyword **3D8ElementArray** is available. It reads an initial identifier for the elements to be generated, an initial identifier for the corresponding nodes, the number of intervals and an identifier for the element material. The elements to be generated closely correspond to the keywords *uniformNodeArray* and *variableNodeArray* which will generate corresponding node definitions. The initial element identifier will be concatenated with interval counters for each direction. The initial identifier for the nodes will be concatenated with the corresponding identifiers for the node array.

e.g.      E   N   4   iso                      4 intervals of elements starting with E and nodes with N

**Class MaterialParser** is defined for parsing material definitions for the FEM model. It throws an exception if a given material identifier already exists or if the number of arguments is not in accordance with the keyword.

Keyword **materials** requires input of either two arguments for the material identifier and the conductivity or three arguments for an additional term for time integration (density  $\rho$  \* specific heat  $c$ ) or four arguments for the material identifier and the three conductivity terms for three directions,  $c_x$ ,  $c_y$ ,  $c_z$ .

**Class InfluenceParser** is defined for parsing nodal load definitions for heat 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 3.

Keyword **nodeloads** expects input of the node load id, the id of the node the load is applied to and the value of the nodal temperature.

It also is defined for parsing line load definitions for heat 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 5.

Keyword **lineLoads** expects input of the line load identifier, the identifier of the start node and the end node and the start and end values of the line load temperature.

It also is defined for parsing line load definitions for heat elements of the FEM model. It throws an exception if a given line load identifier already exists.

Keyword **areaLoad3** expects input of 5 arguments for the area load identifier, the element identifier and 3 temperature values at the nodes.

Keyword **areaLoad4** expects input of 6 arguments for the area load identifier, the element identifier and 4 temperature values at the nodes.

**Class BoundaryConditionParser** is defined for parsing boundary (support) conditions of the FEM model. It throws an exception if a given support identifier already exists or if the number of arguments is not in accordance with the keyword.

Keyword **supports** requires input of 3 arguments for the identifier of the support condition, the identifier of a supported node and a predefined boundary value.

Support definitions may also be generated for constraining a complete face in a 3D node array. Keyword **supportFace** requires input of 5 arguments for the initial identifier for the



support, the face identifier, the initial node identifier, the number of nodal supports to be generated and potentially a prescribed support value not equal to zero.

e.g. S X0 N 5 1. support initial, face, node initial, number of nodes, prescribed value

Support identifiers will be generated on the basis of the initial identifier, face identifier and corresponding indices, associated node identifiers on the basis of the initial node identifier, the face identifier and the indices.

**Class TransientParser** is defined for parsing information for time integration of the FEM model.

Keyword **timeIntegration** requires input of 4 arguments for identifier, maximum time, time step and integration parameter. Keyword **initialTemperatures** requires a set of pairs for the degree of freedom and the associated initial temperature. Keyword **temperatureBC** requires input of 3 arguments for the identifier, the name of a node and a file name containing temperature variation at that node over time, 5 arguments for harmonic temperature variation over time or more than 5 for piece-wise linear variation over time. Keyword **forcingFunction** requires input of a file name to be found in the input directory. Keyword **timeDependentAreaLoad** is defined for constant temperature over time on triangular elements. Keyword **timeDependentNodeLoad** requires 5 arguments for harmonic excitation or more than 5 for with a set of pairs for time and value with piece-wise linear excitation.

This is demonstrated for an example of linear heat transfer with triangular elements.

e.g.	<b>identifier</b> wall corner with 10 elements																																																					
	<b>dimensions</b> 2      1																																																					
	<b>nodes</b> <table><tr><td>n00</td><td>0.0</td><td>3.0</td></tr><tr><td>n01</td><td>0.0</td><td>2.0</td></tr><tr><td>n02</td><td>0.0</td><td>1.0</td></tr><tr><td>n03</td><td>1.0</td><td>3.0</td></tr><tr><td>n04</td><td>1.0</td><td>2.0</td></tr><tr><td>n05</td><td>1.0</td><td>1.0</td></tr><tr><td>n06</td><td>1.0</td><td>0.0</td></tr><tr><td>n07</td><td>2.0</td><td>3.0</td></tr><tr><td>n08</td><td>2.0</td><td>2.0</td></tr><tr><td>n09</td><td>2.0</td><td>1.0</td></tr><tr><td>n10</td><td>2.0</td><td>0.0</td></tr></table>				n00	0.0	3.0	n01	0.0	2.0	n02	0.0	1.0	n03	1.0	3.0	n04	1.0	2.0	n05	1.0	1.0	n06	1.0	0.0	n07	2.0	3.0	n08	2.0	2.0	n09	2.0	1.0	n10	2.0	0.0																	
n00	0.0	3.0																																																				
n01	0.0	2.0																																																				
n02	0.0	1.0																																																				
n03	1.0	3.0																																																				
n04	1.0	2.0																																																				
n05	1.0	1.0																																																				
n06	1.0	0.0																																																				
n07	2.0	3.0																																																				
n08	2.0	2.0																																																				
n09	2.0	1.0																																																				
n10	2.0	0.0																																																				
	<b>heat2D3</b> <table><tr><td>e00</td><td>n00</td><td>n01</td><td>n03</td><td>iso</td></tr><tr><td>e01</td><td>n01</td><td>n04</td><td>n03</td><td>iso</td></tr><tr><td>e02</td><td>n01</td><td>n02</td><td>n04</td><td>iso</td></tr><tr><td>e03</td><td>n02</td><td>n05</td><td>n04</td><td>iso</td></tr><tr><td>e04</td><td>n03</td><td>n04</td><td>n07</td><td>iso</td></tr><tr><td>e05</td><td>n04</td><td>n08</td><td>n07</td><td>iso</td></tr><tr><td>e06</td><td>n04</td><td>n05</td><td>n08</td><td>iso</td></tr><tr><td>e07</td><td>n05</td><td>n09</td><td>n08</td><td>iso</td></tr><tr><td>e08</td><td>n05</td><td>n06</td><td>n09</td><td>iso</td></tr><tr><td>e09</td><td>n06</td><td>n10</td><td>n09</td><td>iso</td></tr></table>				e00	n00	n01	n03	iso	e01	n01	n04	n03	iso	e02	n01	n02	n04	iso	e03	n02	n05	n04	iso	e04	n03	n04	n07	iso	e05	n04	n08	n07	iso	e06	n04	n05	n08	iso	e07	n05	n09	n08	iso	e08	n05	n06	n09	iso	e09	n06	n10	n09	iso
e00	n00	n01	n03	iso																																																		
e01	n01	n04	n03	iso																																																		
e02	n01	n02	n04	iso																																																		
e03	n02	n05	n04	iso																																																		
e04	n03	n04	n07	iso																																																		
e05	n04	n08	n07	iso																																																		
e06	n04	n05	n08	iso																																																		
e07	n05	n09	n08	iso																																																		
e08	n05	n06	n09	iso																																																		
e09	n06	n10	n09	iso																																																		
	<b>materials</b> iso      5.0																																																					
	<b>areaLoad3</b> <table><tr><td>a02</td><td>e02</td><td>30.0</td></tr><tr><td>a03</td><td>e03</td><td>30.0</td></tr></table>				a02	e02	30.0	a03	e03	30.0																																												
a02	e02	30.0																																																				
a03	e03	30.0																																																				
	<b>supports</b> <table><tr><td>s00</td><td>n00</td><td>10.0</td></tr><tr><td>s01</td><td>n01</td><td>10.0</td></tr><tr><td>s02</td><td>n02</td><td>10.0</td></tr><tr><td>s06</td><td>n06</td><td>20.0</td></tr><tr><td>s10</td><td>n10</td><td>20.0</td></tr></table>				s00	n00	10.0	s01	n01	10.0	s02	n02	10.0	s06	n06	20.0	s10	n10	20.0																																			
s00	n00	10.0																																																				
s01	n01	10.0																																																				
s02	n02	10.0																																																				
s06	n06	20.0																																																				
s10	n10	20.0																																																				

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 1 degree of freedom per node, **nodes** are defined with the external identifiers *n00* to *n10* and corresponding 2-dimensional coordinates. Elements **heat2D3** are defined with the external identifiers *e00* to *e09* and material *iso*. **materials** are defined with the identifier *iso* and a double value for the specific property, e.g conductivity. **areaLoad3** are defined with the identifier *a02* and *a03* and finally **supports** are defined with the identifier *s00*, *s01*, *s02*, *s06*, *s10* and a double value for the specific support value (e.g. predefined temperature).

Relations to other objects are defined by referring to the corresponding external string identifiers. For the example, element *e00* is composed of nodes *n00*, *n01* and *n03* and has material *iso*. Elements *e01* through *e09* are defined accordingly. Area load *a02* is associated with element *e02* and support *s00* is associated with node *n00*.

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.

In case of **transient heat conduction** additional keywords defined above are needed for entering the number of eigenstates to be considered, time integration parameters, definition of the specific heat matrix, initial temperatures 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 1<sup>st</sup> order differential equation. It throws an exception if a given support identifier already exists.

Keyword **timeIntegration** requires input of 4 arguments for the identifier of time integration, the maximum time for the integration, the time step and the parameter  $\alpha$  for the integration scheme.

Keyword **initialTemperatures** only needs to be entered for non-zero initial temperatures. In that case a sequence of lines may be entered defining the number of the corresponding degree of freedom and the associated temperature value.

Keyword **temperatureBC** distinguishes between different types of time varying temperature conditions. It expects an identifier for the support condition, the corresponding node identifier and either:

- a single value for a file name containing nodal temperature variation or
- piecewise linear temperature with a succession of time and temperature values at the end of each interval (number of intervals = number of token/2) or
- harmonic variation of temperature with amplitude and circular frequency  $\omega$ : frequency =  $2\pi\omega$  and a phase angle

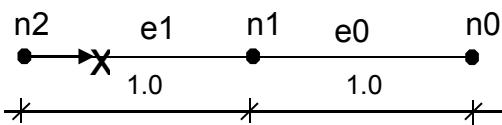
Keyword **forcingFunction** requires input of 1 argument for the name of the file containing the values for the force function.

Keyword **timeDependentAreaLoad** is defined for constant temperatures at triangular elements. It requires input of 5 arguments for the load id, element id, and 3 values at the nodes of triangular elements. The values of temperatures are kept constant over time.

Keyword **timeDependentNodeLoad** is defined for harmonic or piecewise-linear variation of temperatures at nodes. Arguments need to be defined for the load id and the node id and either

- amplitude, frequency and phase angle or
- a set of pairs of (time, temperature) values.

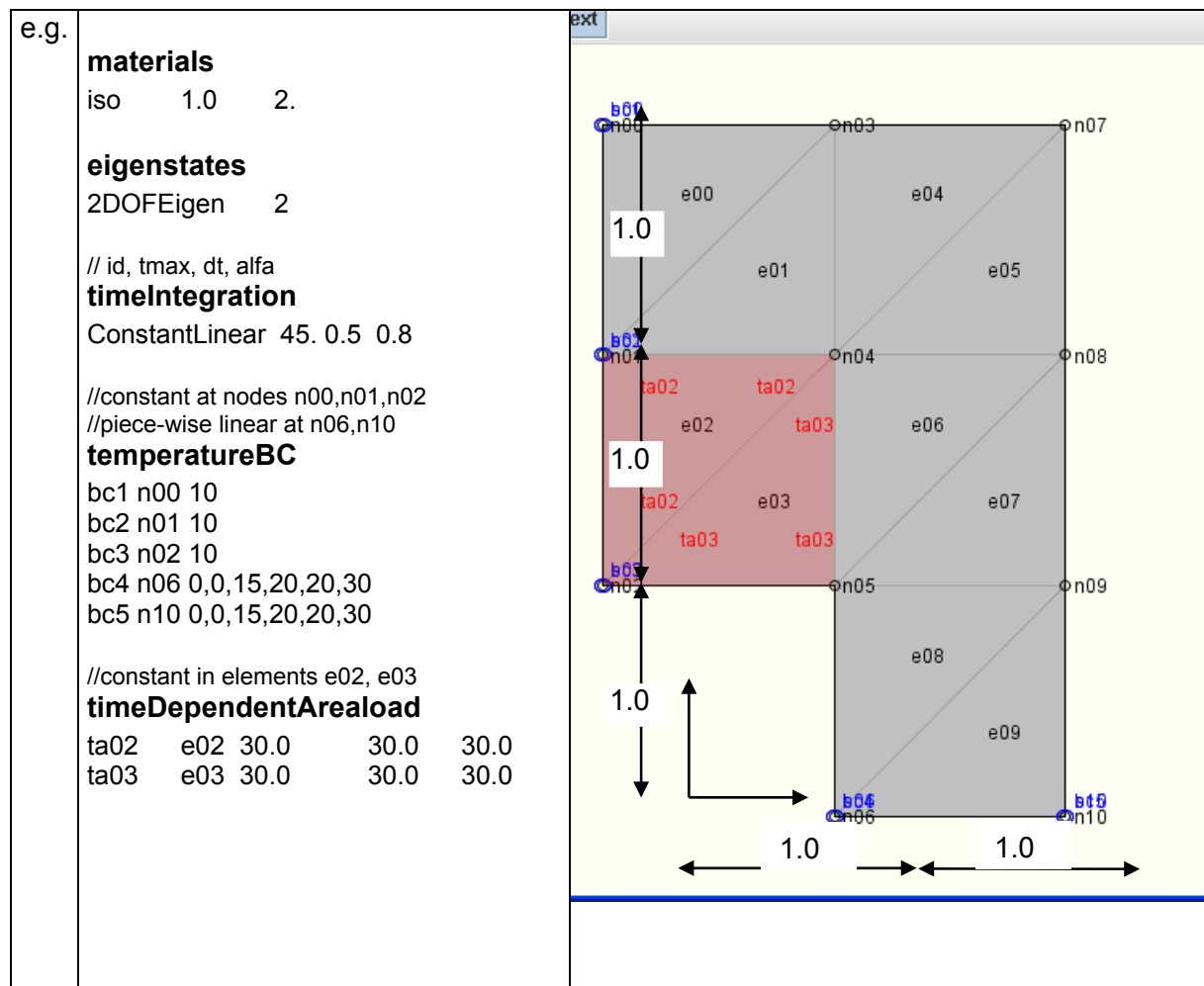
This is demonstrated for a simple rod consisting of 2 elements (e0, e1) with 2 different heat conductivities (c0, c1).

e.g.	<b>identifier</b> 3DOF Sine Function Excitation  <b>dimensions</b> 2      1  <b>nodes</b> n0    2.    0. n1    1.    0. n2    0.    0.  <b>rods</b> e0    n0    n1    c0 e1    n1    n2    c1  <b>materials</b> c0    1.    2. c1    0.01 0.  <b>supports</b> Support0    n2    0.  <b>eigenstates</b> 2DOFEigen    2  // id, tmax, dt, alfa <b>timeIntegration</b> 2DOFTimeSineForce 400. 0.99 0.  <b>ForcingFunction</b> ThreeDOFSine.for	<b>example:</b> Two one-dimensional elements (rods) e0 and e1, defined by three nodes (n0, n1, n2) and two material definitions (c0, c1). Material definition c0 is defined with conductivity=1. and specific heat=2.0. Material definition c1 is defined with conductivity=0.01 and specific heat=0.0. $\dot{x}_0 + x_0 - x_1 = f(t)$ $\dot{x}_1 - x_0 + 1.01 x_1 = 0$ or $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{Bmatrix} \dot{x}_0 \\ \dot{x}_1 \end{Bmatrix} + \begin{bmatrix} 1 & -1 \\ -1 & 1.01 \end{bmatrix} \begin{Bmatrix} x_0 \\ x_1 \end{Bmatrix} = \begin{Bmatrix} f(t) \\ 0 \end{Bmatrix}$ 
------	---	---

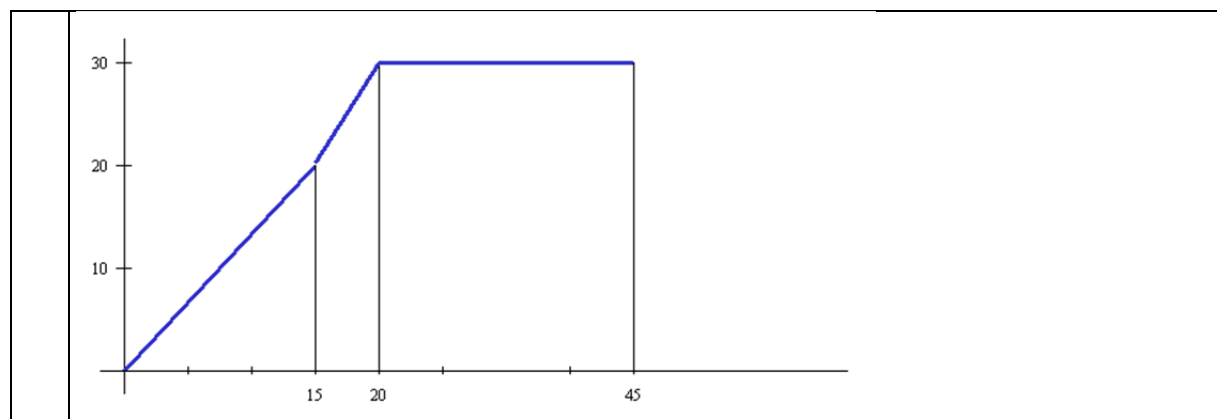
The number of eigenstates (eigenvalues, eigenvectors) to be computed is given by 2. The integration parameters consist of the maximum time for the integration, the time step and the integration parameter alfa.

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

The two-dimensional example from above with temperature kept constant at the left edge and within element e02 and e03 and a piece-wise linear increase at the bottom for transient heat analysis:



where the piecewise-linear increase of temperature at nodes n06 and n10 is defined by the pairs (0,0), (15,20) and (20,30) after which the temperature remains constant.



### 3.3 Model information for the analysis

Model information for the analysis of linear heat problems consists of the Finite Element approximation defining the element equations, the influences on the model (loads), the constitutive relations (material properties) and the external boundary conditions (supports).

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 triangular elements in elasticity analysis is collected in **class Heat2D3**.

**Class Heat2D3** is a linear triangular element for stationary heat flow in two dimensions. It is named *Heat2D3* as generally there could be other element types (e.g. four node or higher order) for the analysis of the same class of problems. It will be derived from *abstract class AbstractLinear3Node2D*.

It is contained in *package heat.model*. It requires to import the functionality of the model and the node implementation from the general FEM Framework. It throws an algebraic exception for some selected algebraic problems (e.g. area of triangle is zero).

The programming interface of class *Heat2D3* provides methods for defining the node identifiers of the element nodes, for computing the element matrix, for computing the heat state at the midpoint of the elements and for printing element properties and behaviour.

Class *Heat2D3* defines a number of static attributes for

- the number of degrees of freedom per node of the element
- the element matrix,
- the temperatures at the nodes of the elements,

It is not necessary to store all the element information for each individual object of class Heat2D3 as this information is only utilized to set up the corresponding system equations.

Each new element will thus use the identical memory space for evaluation of the information required for the analysis, e.g. calculating the element stiffness matrices. Element information will be regenerated when needed again.

The constructor will generate new elements on the basis of the string identifier for that element, the identifiers of the 3 element nodes and the material identifier.

Also, a method is defined for storing the identifiers of all element nodes.

```
public class Heat2D3 extends AbstractLinear3Node2D {
    private static final long serialVersionUID = 1L;
    private static final int ELEMENT_DOF_PER_NODE = 1;
    private static double elementMatrix[][] = new double[3][3];
    private static double specificHeatMatrix[] = new double[3];
    private static double elementTemperatures[] = new double[3];

    // ....Constructor.....
    public Heat2D3(String id, String[] eNodes, String materialId) {
        super(id, eNodes, materialId, ELEMENT_DOF_PER_NODE);
    }
}
```

At the heart of the element implementation is a method for computing the element matrix (*computeElementMatrix*). It starts by evaluation of the element geometry (*computeGeometry*) and by evaluating the global derivatives of the element shape functions (*computeSx*). It next determines the material property for the current element.

The element matrix is then computed by evaluating the product  $K_e = area * c * S_x * S_x^T$ .

```
// ....Compute element Matrix.....
public double[][] computeMatrix() throws AlgebraicException {
    computeGeometry();
    Sx = computeSx();
    double conductivity = ((Material) material).getConductivity();
    // Ke = area*c*Sx*SxT
    elementMatrix = MatrixAlgebra.multMatrixTransposed
        (0.5*determinant*conductivity, Sx, Sx);
    return elementMatrix;
}
```

For time integration of heat transfer problems, a method for computing the specific heat matrix (*computeDiagonalMatrix*) is provided. It starts by evaluation of the element geometry (*computeGeometry*). The specific heat matrix is assumed to be diagonal with diagonal terms defined by the product of material density with conductivity and element area divided by 3:

The diagonal matrix is thus computed by evaluating the products  $C_e = \rho * c * area/3$ .

```
// ....Compute diagonal Specific Heat Matrix.....
public double[] computeDiagonalMatrix() throws AlgebraicException {
    computeGeometry();
    // Me = density * conductivity * 0.5*determinant / 3 (area/3)
    specificHeatMatrix[0]=specificHeatMatrix[1]=specificHeatMatrix[2]
        = ((Material) material).getRhoC() * determinant/6.;
    return specificHeatMatrix;
}
```



The internal **heat flow within an element** is computed as the element behavior from the partial derivatives of the temperature **T** with respect to the global variables  $x_i$  as given above:

$$\mathbf{f}_e = -c_e * \mathbf{g}_e = -c_e * \mathbf{S}_{xe}^T \mathbf{T}_e \quad \text{internal heat flow}$$

Method *computeHeatState* uses attributes *node*, *conductivity* and a vector for the heat flow (*midPointHeatState*) at the center of each element.

The element geometry and global derivatives of the element shape functions are regenerated. The temperature values for each node are retrieved and stored in vector *elementTemperatures*. The heat gradients in both directions within the element are evaluated by computing the product  $\mathbf{S}_x^T * \mathbf{T}$  for each element. Finally, the heat gradients are multiplied by the value of conductivity in order to get the element heat flow.

```
// ....Compute the heat state at the midpoint of the element.....
public double[] computeHeatState(double z0, double z1) {
    double midpointHeatState[] = new double[2]; // in element
    computeGeometry();
    Sx = computeSx();
    for (int i = 0; i < 3; i++)
        elementTemperatures[i] = node[i].getNodalDOF()[0];
    double conductivity = ((Material) material).getConductivity();
    midpointHeatState = MatrixAlgebra.multTransposed
        (-conductivity, Sx, elementTemperatures);
    return midpointHeatState;
}
```

**Class Heat2D4** is a linear rectangular element for stationary heat flow in two dimensions. It will be derived from class **AbstractLinear4Node2D**. It is contained in *package heat.model*. It requires to import the functionality of the model and the node implementation from the general FEM Framework. It throws an algebraic exception for some selected algebraic problems.

The programming interface of class *Element4* provides methods for defining the node identifiers of the element nodes, for computing the element matrix, for computing element geometry (area), for computing the element shape functions and for computing the heat state at the midpoint of the elements.

Class *Heat2D4* defines a number of static attributes for

- the number of degrees of freedom per node of the element,
- the element matrix,
- the temperatures at the nodes of the elements,

It is not necessary to store all the element information for each individual object of class *Heat2D4* as this information is only utilized to set up the corresponding system equations.

Each new element will thus use the identical memory space for evaluation of the information required for the analysis, e.g. calculating the element stiffness matrices. Element information will be regenerated when needed again.

The constructor will generate new elements on the basis of the string identifier for that element, the identifiers of the 4 element nodes and the material identifier.

At the heart of the element implementation is a method for computing the element matrix (*computeMatrix*). Method *computeMatrix* requires evaluation of the element geometry (*computeGeometry*) and of the global derivatives of the element shape functions (*computeSx*). Interpolation of loads is performed using the element shape functions. These are determined in method *computeS*.

The element matrix is then computed by first retrieving the material property, setting up the sum over the Gauss points, by evaluating the determinant and shape function derivatives at each Gauss point and evaluating the product

$$\mathbf{k}_e = c * \mathbf{S}_x * \mathbf{S}_x^T * \text{determinant}$$

at each Gauss point and summing up the contributions for each Gauss point.

```
// ....Compute element Matrix.....
public double[][] computeMatrix() {
    double z0, z1;
    double gaussCoord[] = {-1./Math.sqrt(3), 1./Math.sqrt(3)};
    double conductivity = ((Material)material).getConductivity();
    MatrixAlgebra.clear(elementMatrix);
    for (int i = 0; i < gaussCoord.length; i++) {
        for (int j = 0; j < gaussCoord.length; j++) {
            z0 = gaussCoord[i];
            z1 = gaussCoord[j];
            computeGeometry(z0, z1);
            Sx = computeSx(z0, z1);
            // Ke = C*Sx*SxT*determinant
            MatrixAlgebra.multAddMatrixTransposed
                (elementMatrix, determinant*conductivity, Sx, Sx);
        }
    }
    return elementMatrix;
}
```

The internal **heat flow within an element** is computed as the element behavior from the partial derivatives of the temperature **T** with respect to the global variables  $x_i$  as given above:

$$\mathbf{f}_e = -c_e * \mathbf{g}_e = -c_e * \mathbf{S}_{xe}^T \mathbf{T}_e \quad \text{internal heat flow}$$

Method *computeHeatState* uses attributes *node*, *conductivity* and a vector for the heat flow (*midPointHeatState*) at the center of each element.

The element geometry and global derivatives of the element shape functions are regenerated. The temperature values for each node are retrieved and stored in vector *elementTemperatures*. The heat gradients in both directions within the element are evaluated by computing the product  $\mathbf{S}_x^T * \mathbf{T}$  for each element. Finally, the heat gradients are multiplied by the value of conductivity in order to get the element heat flow.

```

// ....Compute the heat state at the (z0,z1) of the element.....
public double[] computeHeatState(double z0, double z1) {
    double midpointHeatState[] = new double[2]; // in element
    computeGeometry(z0, z1);
    Sx = computeSx(z0, z1);
    for (int i = 0; i < getNodesPerElement(); i++)
        elementTemperatures[i] = node[i].getNodalDOF()[0];
    double conductivity = ((Material)material).getConductivity();
    midpointHeatState = MatrixAlgebra.multTransposed
        (-conductivity, Sx, elementTemperatures);
    return midpointHeatState;
}

```

Functionality that is required for 3-dimensional linear triangular elements in elasticity analysis is collected in **class Heat3D8**.

**Class Heat3D8** is a linear triangular element for stationary heat flow in two dimensions. It is named *Heat3D8* as generally there could be other element types (e.g. four node or higher order) for the analysis of the same class of problems. It will be derived from *abstract class AbstractLinear8Node3D*.

It is contained in *package heat.model*. It requires to import the functionality of the model and the node implementation from the general FEM Framework. It throws an algebraic exception for some selected algebraic problems (e.g. area of triangle is zero).

The programming interface of class *Heat3d3* provides methods for defining the node identifiers of the element nodes, for computing the element matrix, for computing the heat state at the midpoint of the elements and for printing element properties and behaviour.

### 3.3.2 defining material properties

The material description in heat transfer problems is given in general by the **heat conductivity matrix  $\mathbf{C}$** . The heat conductivity matrix is then needed for the evaluation of the element matrices as described before:

The material description chosen for the purposes of this text is an isotropic material with a constant rate of conductivity in all directions. Therefore matrix  **$\mathbf{C}$**  can be replaced by:

$$\mathbf{C} = c \mathbf{I} \quad \text{i.e. a constant } c \text{ multiplied by the identity matrix}$$

Therefore, the conductivity  $c$  is a multiplier that is constant for a given element. We can, however, associate different values of conductivity  $c$  with different elements assigning different material identifiers to it.

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

It simply consists of a constructor for passing the material identifier to the model and for storing the material property under name "Conductivity" and it finally provides a method for retrieving the material property (*getConductivity*).

### 3.3.3 defining external influences on the system

External influences on the model can either be defined as

- **nodal influences**  
are concentrated heat loads at the nodes of the element mesh. The intensity of the load is given. The intensity is positive if the heat flow is directed out of the body.
- **line influences**  
are line heat loads between two neighboring nodes of the element mesh. The line heat intensity is defined. If the line load intensity is constant, the value is given. If it varies linearly along the line, the intensities at the end nodes are given. The intensity is positive if the heat flow is directed out of the body.
- **area influences**  
are surface heat sources with a given heat flow intensity. Heat flow into an element is by definition positive. If the heat load intensity over an element is constant, the constant value is given. If the heat load intensity varies linearly across the element, its value is given at each of the three nodes of the element.

Class **NodeLoad** contains the implementation of **nodal influences**. These are simply defined by specifying the load intensity at a specific node.

**Line influences** are specified in general by the relation between heat input at boundary  $q_{R(i)}$  and the concentrated heat transfer at node  $i$ :  $Q_{Ri}$

$$\begin{pmatrix} Q_{R1} \\ Q_{R2} \end{pmatrix} = \frac{L}{6} \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} q_{R(1)} \\ q_{R(2)} \end{pmatrix} \quad (3.55)$$

The implementation of the line load is contained in *package heat.model*. It needs to import class *AbstractLineLoad* from *package model.implementation* from *package model.interfaces* of the general FEM Framework. It will be derived from *abstract class AbstractLineLoad*.

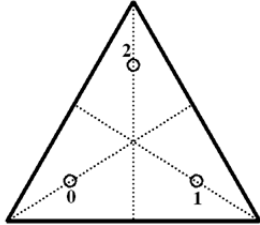
**Class LineLoad** defines a constructor for a linearly varying line load. The method for specifying the intensity of the line load (*setIntensity*) will expect the nodal values of the load intensity as input parameters.

The method for computing the load vector (*computeLoadVector*) will evaluate the element load vector based upon equation 3.55 given above. For this purpose, the two nodes of the line load need to be retrieved, the corresponding coordinates need to be evaluated, the length of the edge between the two nodes will be computed and the element load vector will be returned.

**Area influences** in 2 dimensions are specified by

$$\mathbf{w}_e = \int_{A_e} \mathbf{S}_e q_e dA \quad : \quad \text{element vector} \quad (3.58)$$

In case of **triangular elements** with a constant load  $q_e$  the integral could easily be evaluated analytically. In general, however, the integral is integrated numerically by Gaussian integration where the integral is replaced by a sum over the function values at selected Gauss-points multiplied by corresponding weight factors (Gauss-weights). For the simple triangular element chosen three Gauss-points and weights were chosen and are defined as follows:

	points $p_i$	Coordinates $z_0, z_1, z_2$	weight $g_i$
	0	2/3 1/6 1/6	1/3
	1	1/6 2/3 1/6	1/3
	2	1/6 1/6 2/3	1/3

Load values are given as discrete values at the nodes. Interpolation of load values at corresponding Gaussian coordinates is accomplished via element shape functions. Thus, load values at Gaussian coordinates are determined by:

$$q(p_i) = \sum_{i=0}^2 \begin{Bmatrix} z_i(p_0) * q_i \\ z_i(p_1) * q_i \\ z_i(p_2) * q_i \end{Bmatrix}$$

Finally, the integral can be replaced by the sum of element shape functions and element load values at Gaussian coordinates multiplied by the corresponding Gaussian weights. The multiplication by the Jacobian takes care of the local to global coordinate mapping as integration is carried out over the normalized local coordinates.

$$\mathbf{w}_e = \sum_{i=0}^2 \begin{Bmatrix} z_i(p_0) \\ z_i(p_1) \\ z_i(p_2) \end{Bmatrix} * q(p_i) * g_i * \text{determinant}$$

**Class AreaLoad3** specifies a static attribute for the load vector and attributes for the vector of the load intensities at the nodes, for the matrix for mapping coordinate increments to independent normalized coordinate increments and for the element area (determinant).

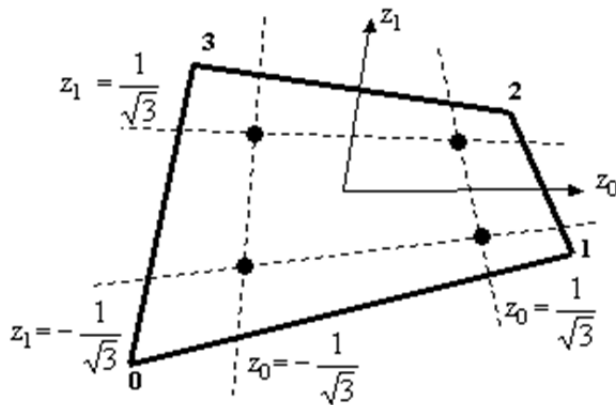
A constructor is defined for an element load defined by different values at each node.

Two methods are provided for defining and retrieving the load intensities at the nodes.

The element load vector can be computed (*computeLoadVector*) and the results printed. This method will retrieve the nodes of the element and the coordinates of each node. The determinant) will be evaluated and the load vector determined by Gaussian integration.

```
// .....Compute the element load vector.....
public double[] computeLoadVector() {
    Heat2D3 element3 = (Heat2D3)element;
    element3.computeGeometry();
    double area = 0.5*element3.getDeterminant();
    double gaussWeight = 1./3.;
    double gc0 = 2./3., gc12 = 1./6.;
    double qp0, qp1, qp2;
    double vector[] = new double[3];
    p0 = ( gc0*load[0] + gc12*load[1] + gc12*load[2]);
    qp1 = (gc12*load[0] + gc0*load[1] + gc12*load[2]);
    qp2 = (gc12*load[0] + gc12*load[1] + gc0*load[2]);
    vector[0] = (gc0*qp0 + gc12*qp1 + gc12*qp2) * gaussWeight * area;
    vector[1] = (gc12*qp0 + gc0*qp1 + gc12*qp2) * gaussWeight * area;
    vector[2] = (gc12*qp0 + gc12*qp1 + gc0*qp2) * gaussWeight * area;
    return vector;
}
```

In case of **4-node elements** the integral is integrated numerically by Gaussian integration where the integral is replaced by a sum over the function values at selected Gauss-points multiplied by corresponding weight factors (Gauss-weights). For the quadrilateral element chosen four Gauss-points and weights were chosen and are defined as follows:



with the associated Gauss weights equal to

**1.0**

**normalized coordinates**

Load values are given as discrete values at the nodes. Interpolation of load values at corresponding Gaussian coordinates is accomplished via element shape functions. Thus, load values at Gaussian coordinates are determined by:

$$q(p_i) = \sum_{i=0}^3 \begin{Bmatrix} z_i(p_0) * q_i \\ z_i(p_1) * q_i \\ z_i(p_2) * q_i \\ z_i(p_3) * q_i \end{Bmatrix}$$

Finally, the integral can be replaced by the sum of element shape functions and element load values at Gaussian coordinates multiplied by the corresponding Gaussian weights. The multiplication by the Jacobian takes care of the local to global coordinate mapping as integration is carried out over the normalized local coordinates.

$$\mathbf{w}_e = \sum_{i=0}^3 \begin{Bmatrix} z_i(p_0) \\ z_i(p_1) \\ z_i(p_2) \\ z_i(p_3) \end{Bmatrix} * q(p_i) * g_i * \text{determinant}$$

**Class AreaLoad4** specifies a static attribute for the load vector and attributes for the vector of the load intensities at the nodes, for the matrix for mapping coordinate increments to independent normalized coordinate increments and for the element area (determinant).

Two constructors are defined for an element load defined by a single or by different values at each node.

Two methods are provided for defining and retrieving the load intensities at the nodes.

Finally, the element load vector can be computed (*computeLoadVector*). This method will retrieve the nodes of the element and the coordinates of each node. In case of the 4-node element in 2-dimensions, the determinant will be evaluated and the load vector determined by Gaussian integration.

The results are printed using method (*printAreaLoads*).

```

// .....Compute the element load vector.....
public double[] computeLoadVector() {
    Heat2D4 element4 = (Heat2D4)element;
    int row, col;
    double z0, z1;
    double [][] ssT = new double[4][4];
    double [] s = new double[4];
    double[] gaussCoord = {-1./Math.sqrt(3),1./Math.sqrt(3)};
    double vector[] = new double[4];

    for ( int i=0; i<gaussCoord.length; i++ ) {
        for( int j=0; j<gaussCoord.length; j++ ) {
            z0 = gaussCoord[i];
            z1 = gaussCoord[j];
            element4.computeGeometry(z0, z1);
            s = element4.computeS(z0, z1);

            for (row=0; row < 4; row++)
                for (col=0; col < 4; col++)
                    ssT[col][row] = ssT[col][row] + element4.getDeterminant()
                        * s[row] * s[col];
        }
    }
    for (row=0; row < 4; row++) {
        vector[row] = 0.;
        for (col=0; col < 4; col++)
            vector[row] = vector[row] + ssT[row][col] * load[row];
    }
    return vector;
}

```

### 3.3.4 setting up support conditions

Support conditions for the model can be defined for each node. Since nodes in heat flow analysis have only a single degree of freedom, only the prescribed value at a node needs to be defined and the system status vector needs to be adjusted.

**Class Support** is contained in *package heat.model*. It is derived from class *AbstractSupport* which needs to be imported from *package model.implementation*. It also needs to implement interface *model.interfaces.ISupport*.

The constructor will expect the support identifier, the identifier of the corresponding node and the prescribed temperature value as input parameters thus defining the support to be restrained to the specified value. Methods *getPrescribed*, *getRestraint* and *getReaction* will return the status of the node (restraint or not), the prescribed value of the nodal temperature and the heat stream at the node as the nodal reaction. Method *setReaction* will save the nodal reactions according to the parameters specified.

### 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 heat.model*. It is derived from class *AbstractTimeInt1st* which needs to be imported from *package model.abstractclasses*.



	<pre> package heat.model;  import framework.model.AppObject; import framework.model.interfaces.ITimeInt1st;  public class TimeInt extends AppObject implements ITimeInt1st {     private static final long serialVersionUID = 1L;     private int nSteps;     private double tmax, dt, alfa;     private double[][] temperature, forceFunction;      // ....Constructor.....     public TimeInt(String id, double _tmax, double _dt, double _alfa,                     double[] initial, double [][] _forceFunction) {         super(id);         this.tmax = _tmax;         this.dt = _dt;         nSteps = (int)(tmax/dt)+1;         this.alfa = _alfa;         temperature = new double[nSteps][initial.length];         temperature[0] = initial;         this.forceFunction = _forceFunction;     } </pre>
	<pre> // ..get()..... public double getTmax() { return tmax; } public double getDt() { return dt; } public double getParameter1() { return alfa; } public double[][] getFieldVariables() { return temperature; } public double[] getInitial() { return temperature[0]; } public double[][] getForceFunction() { return forceFunction; } </pre>

Class *TimeInt* also provides some functionality for output of information in the context of time integration of instationary heat conduction. This includes output of time integration parameters *id* and *alfa* for the analysis

### 3.4 Output of system results

Output functionality as a result of the analysis of 2-dimensional stationary heat transfer is implemented in two different classes, one for alphanumeric console output (HeatOutputConsole) and another one (VisualHeatModel) for output in a graphical frame (window Heat Visualizer) created with Java Swing technology.

#### 3.4.1 Alphanumeric output of results

Class *HeatOutputConsole* is contained in package *heat.output*. 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 and support.

Class *HeatOutputConsole* 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, 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 materials defined in the model and print out the material identifier and the material values, i.e. conductivity and the product of density\*specific heat.

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 temperatures, heat streams, reactions, visualization of results and for terminating output of results analyzed.

If the model has been solved, options are presented for displaying **n**(odal temperatures), **h**(eat streams in elements), **r**(eactions) 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 *VisualHeat2DModel*** is contained in package `heat.output`. It is implemented solely on the basis of Java Swing technology. The implementation of this class depends upon knowledge of the general concepts of graphical user interfaces for engineering problem solving and upon a fundamental understanding of the concepts of Java Swing technology. For the purposes of this text, only the results of graphical visualization will be demonstrated. The other subjects would have to be addressed separately.

A graphical visualization will only be able to visualize information of a model. Generally, there will be no provision for user interaction with information presented in a graphical form. The presentation is what is sometimes called “passive graphics” without any provisions for interaction between the presentation and the model.

A simple visualization will generally be sufficient for checking the validity of input data and for illustrating the results of the analysis. It is however often not sufficient for the purposes of engineering project work.

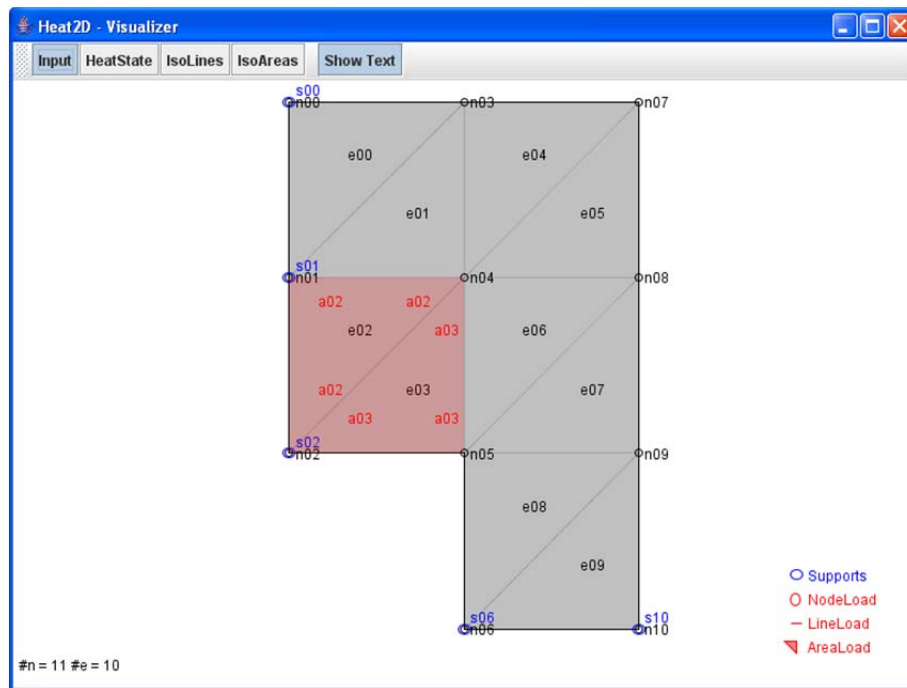
Also, a graphical presentation cannot be used easily as a basis for further detailing of the graphical presentation in the context of an engineering project (e.g. dimensioning, annotations, etc.). Therefore, it is not a suitable basis for integration of information in construction processes.

Visualization of information is primarily a tool for verifying the validity of input data and for visualizing results of an analysis.

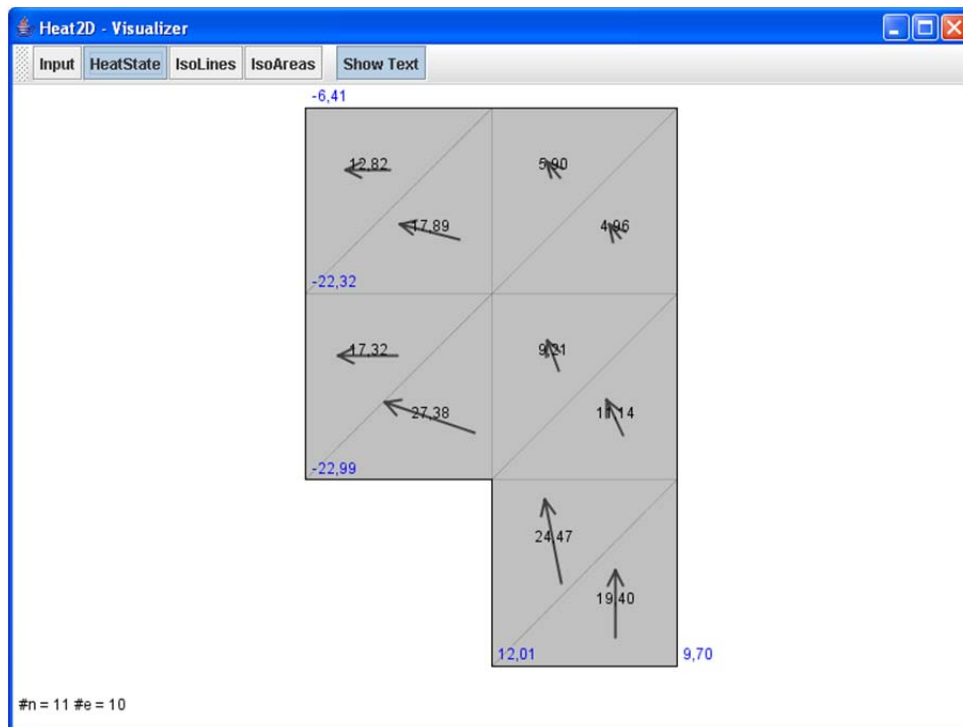
For these reasons, a visualization is generally a simple pixel image

- which can only be edited to a very limited extend,
- which is not drawn to scale in a coordinate system,
- which is difficult to use for integration purposes with other engineers working on the same project.

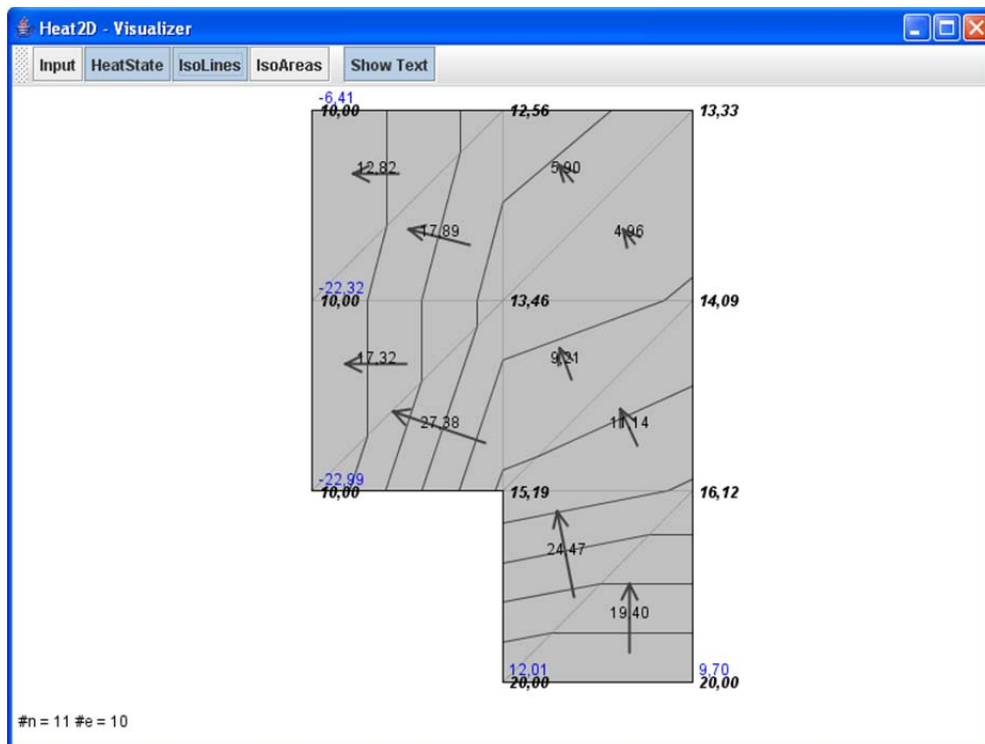
The first frame visualizes the input data in a graphical representation.



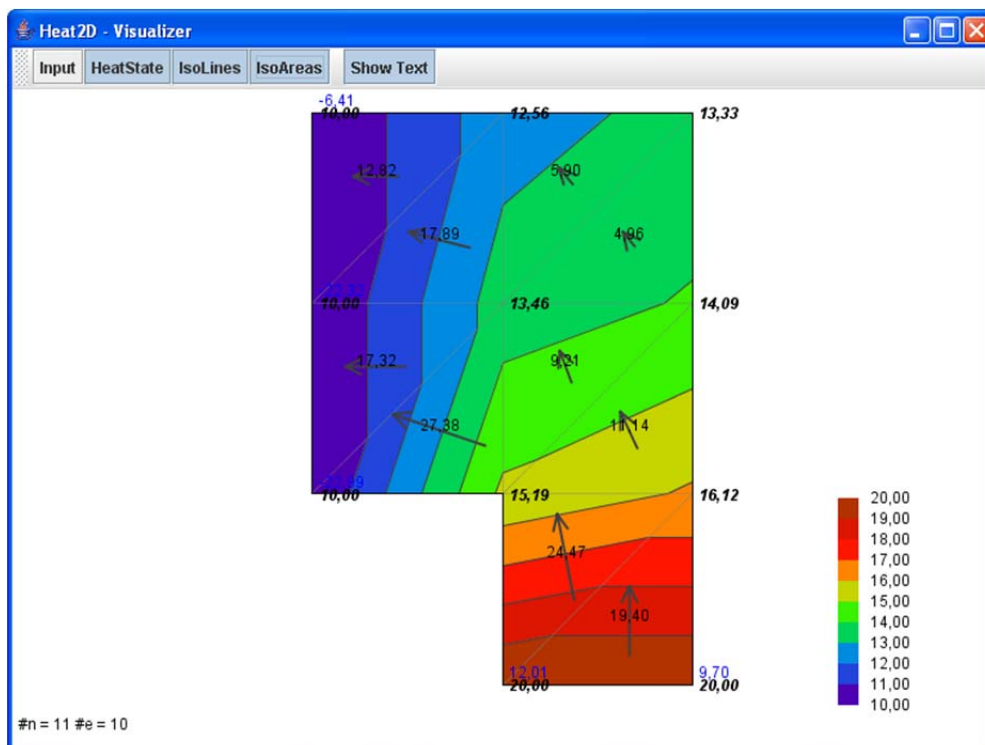
The next frame visualizes the heat gradients in the midpoints of the elements.



The next frame visualizes isolines for levels of the same heat states in the system.



The final frame for stationary analysis visualizes isochromatic regions for levels of the same heat states in the system.



**Class *VisualTransientHeat2D*** is contained in package `heat.output`. It is also implemented solely on the basis of Java Swing technology.

A simple visualization will generally be sufficient for checking the validity of input data and for illustrating the results of the analysis. It is however often not sufficient for the purposes of engineering project work.

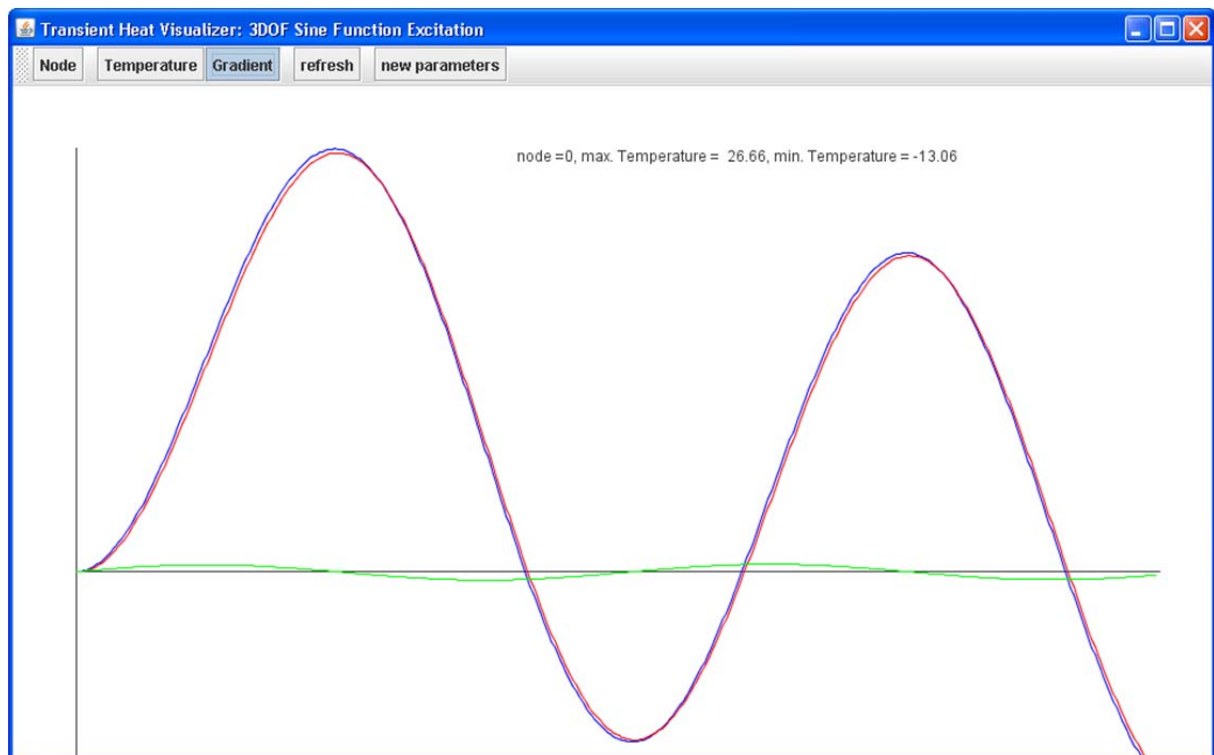
Visualization of information is primarily a tool for verifying the validity of input data and for visualizing results of an analysis.

For these reasons, a visualization is provided which allows for

- entering the node for plotting the nodal time history,
- the selection of the temperature or temperature gradient at the selected node and
- a plot of the selected curve with a print-out of the maximum and minimum time history values.

Several curves may be superimposed until the screen is renewed with the refresh-button.

The influence of specific parameter settings may be investigated using the “new parameters” button.



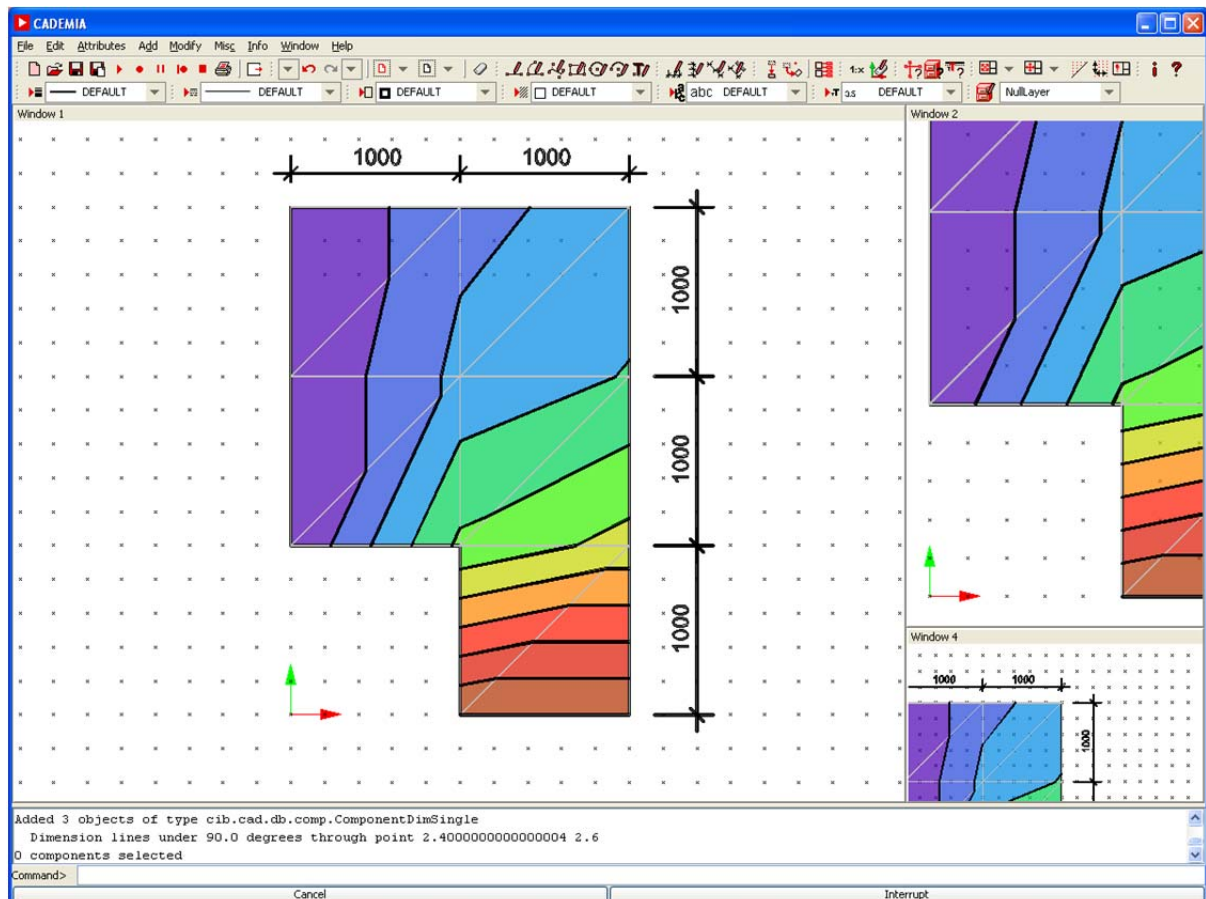
### 3.5 Integration into a CAD environment

Requirements on an integrated use of geometrical information in engineering processes generally go far beyond the capabilities of a simple visualization.

If it is required to edit and enhance a geometrical model, if it is further required to identify individual components of an engineering analysis and if it is required to access and manage information in the context of a coordinate system to a specific scale, technology that is generally referred to as CAD (Computer Aided Drafting) will have to be used.

Normally, CAD technology is only available in commercial products with limited provision for integrating custom solutions and even for openly understanding internal concepts. It is normally a “black box” system that can only be used for purposes of integration via data interfaces or proprietary application programming interfaces (API). In addition, the cost for such products is not negligible.

For these reasons, CAD technology under the name **cademia** was used that is publically available [4] and that is completely open for any developer even to the source code level. The technology was developed under the provisions of the Open Source Initiative (OSI) [5]. It was built upon Java Swing technology and is – thus far – limited to 2-dimensional drafting of technical drawings. A detailed discussion of the implementation is beyond the scope of this text and will have to be treated separately.





## Bibliography

- [1] Handouts, Project of Civil Engineering, Software-Design for Numerical Calculation of Physical Processes (stationary plane heat transfer), Beucke, Könke, Richter, Luther, Weimar 10/2005.
- [2] Pahl, P. J.: Finite Element Method, Stationary Heat Flow, TU Berlin, November 2000
- [3] Scriptum, Framework for the Analysis of Physical Behaviour with the Finite Element Method (FEM), Bauhaus-Universität Weimar, Informatik im Bauwesen, 2009
- [4] <http://www.cademia.org>, © 2006, Bauhaus-Universität Weimar | Lombego Systems
- [5] <http://www.opensource.org/>

additional reading:

- [5] Sun Microsystems, Inc., Copyright 2006: Download The Java Tutorial, <http://java.sun.com/docs/books/tutorial/>.
- [3] Pahl, P. J.: Finite Element Methoden für Physikalische Aufgaben, TU Berlin, Oktober 1991
- [8] Zienkiewicz, O. C.: The Finite Element Method, McGraw Hill, ISBN 0-07-084072-5
- [9] Bathe, Klaus-Jürgen: Finite Element Procedures, Prentice Hall, ISBN 0-13-301458-4