Linear Finite Element Method - soofeaM

Finite Element Method [304.007]

05.05.2022

Contents

Download the zip-folder you find at `Material - Unit 2` from the TeachCenter and unzip it. Copy the folder `soofeam_start` into your lecture folder. Go to the directory `soofeam_start`.
soofeaM (Software for Object Oriented Finite Element Analysis in Matlab) is the name of the Matlab program we will work with. Let's talk about the folder structure briefly:

- **src**: In the source folder, all classes of the program are found. They are packed into namespaces (folders which start with a '+'). This helps creating a hierarchical and overseeable structure. For more infos about namespaces, consult the Matlab documentation.
- **tests**: Here little scripts are stored, which test isolated parts of the program on their functionality - so called *unit tests* (they test units of the program).
- **examples**: In this folder, a subfolder is created for each example you want to calculate. An example is basically specified by geometry and boundary conditions. The input- and output files of the examples are found here.

1 Exercise 1 - Read a Model

We use the open-source program `Gmsh` for generating the finite element mesh. If you do not have this software already, you find it in the TeachCenter in the section "Software". Just download it and run the executable. If you encounter any problems, please contact me. The script `testModel` reads a model from an input file and displays it on the command window. An UML diagram of the model is given in Section 9.1.
If you want more background information about the model design, checkout this learning video[1].

Exercise

a) Create the input file. Therefore, open the file `examples/plateQuads/plateQuads.geo` with `Gmsh`. Mesh the geometry by clicking `Modules/Mesh/2D`. Save the mesh by clicking `File/Save Mesh`.

b) Read the contents of the script `testModel.m`. *Do not run the script directly!* Run the script `soofeam_unit2/testing.m` instead. Run all unit tests in the follwing exercises through this master script.

   Find out

   i) the coordinates of the node with number 89 (use `model.node_dict`),
   ii) which nodes belong to the element with number 23 (use `model.element_dict`),
   iii) the coordinates of these nodes (use `NodeContainer.getCoordinateArray`),
   iv) how many faces belong to the boundary with number 3 (use `model.boundary_dict`).

[1] `https://tube.tugraz.at/paella/ui/watch.html?id=676b8f10-99e2-4b93-a511-794547f49964`

2 Exercise 2 - Shape Functions

The base class for everything related to shape functions is `nsModel.nsShape.Shape`. All classes related to shape functions of concrete elements are derived from this base class. Examples for concrete elements are 1d-elements (`LinearShape`), triangle elements (`TriangleShape`) and quadrangle elements (`QuadShape`). Remember that all shape functions are defined on the *reference element*.
The only methods of this class you have to know are:

- `Shape.getArray(coordinates)`: Returns the interpolation matrix

$$N(\xi,\eta) = \begin{bmatrix} N_1(\xi,\eta) & N_2(\xi,\eta) & \cdots & N_n(\xi,\eta) \end{bmatrix}^T .$$

  This matrix contains the values of *all* shape functions of the element at position `coordinates = `$[\xi\ \eta]$ (natural coordinates on the reference element). You already saw this matrix in the learning video.

- `Shape.getDerivativeArray(coordinates)`: Returns the matrix `dN`. This matrix contains all partial derivatives of all shape functions. In 2D, this matrix is written as

$$dN(\xi,\eta) = \begin{bmatrix} \frac{\partial N_1}{\partial \xi} & \frac{\partial N_2}{\partial \xi} & \cdots & \frac{\partial N_n}{\partial \xi} \\ \frac{\partial N_1}{\partial \eta} & \frac{\partial N_2}{\partial \eta} & \cdots & \frac{\partial N_n}{\partial \eta} \end{bmatrix}^T .$$

  You already know this matrix from the learning video as well. This is the exact matrix we use twice when calculating the stiffness matrix, i.e. the terms ${N^l}_{,o}$ and ${N^j}_{,p}$ in Equation (1) on page 6.

The script `testShape.m` is a simple unit-test of the shape functions of a linear triangle element. Create a test for the bilinear quadrangle element in analogy:

Exercise The reference element for quad elements is the square $[0,1] \times [0,1]$. (see file `testShape.m`).

a) Check (in analogy to the triangle) if the 4 shape functions of the bilinear quad element take the value 1 in their own node and the value 0 in all other nodes.

b) Calculate all partial derivatives of all shape functions of the bilinear quad element at position (0.75/0.15). What is the value of the derivative $\partial N^3/\partial\eta = {N^3}_{,\eta}$ at this position? *Hint:* Use the method `Shape.getDerivativeArray`.

3 Exercise 3 - Numerical Integration

If you need a quick reminder on numerical integration, checkout this video[2].
In the following, all classes related to the numerical integration are explained briefly. They are depicted in the UML diagram of namespace `nsNumeric` in Section 9.2.

[2] `https://tube.tugraz.at/paella/ui/watch.html?id=fe32b621-16d9-4560-8d70-6226513971f9`

- `IPData`: Contains information about positions and weights of all integration points on the reference elements of 1d lines, triangles, quads, etc. It serves as a database.
- `NaturalIntegrationPoint`: The objects of this class are the integration points on the reference element. They have their weight and position as properties.
- `nsModel.RealIntegrationPoint`: The integration points on the 'real' element in the global x,y coordinate system. They have their x,y coordinates and their assigned `NaturalIntegrationPoint` as properties.
- `NumInt`: This class provides the following functions:
  - `getNaturalIntegrationPoints(shape_type, number_of_int_points)`: This method creates an array of type `nsNumeric.NaturalIntegrationPoint`. It contains the integration points of the chosen shape type (triangle, quad, ...) with the chosen amount of integration points.
  - `integrate(func, int_point_array)`: This method takes a function handle and a list of integration points as input. It performs the numerical integration according to the formula

    $$\int_{\Omega_{ref}} f(x)\,\mathrm{d}x \approx \sum_{i=0}^{n} f(x_i) w_i\,.$$

    This function is used in exercises only.
  - `methodIntegrate(func, int_point_array, element)`: This method is used to calculate all appearing integrals in the stiffness matrices and force vectors.

The script `testNumericalIntegration.m` demonstrates how the function `integrate` can be used for integrating a function numerically. The more important function `methodIntegrate` will be treated in Exercise 5 (Jacobian matrix).

Exercise Create a sketch of the reference quad element $[0, 1] \times [0, 1]$. Find out the positions of the integration points on the reference element and mark them into your sketch qualitatively. Use 3 integration points per coordinate direction. Additionaly find out the weight of each integration point and write it into the sketch.
Check if the sum of the weights of all integration points yields the area of the reference quad. Why must this be true?
Hint: Use the method `nsNumeric.NumInt.getNaturalIntegrationPoints.`

4 Exercise 4 - ElementType

The class `Type` combines the geometric element, the shape functions and the numerical integration (integration points). This is shown in the UML diagram of namespace `nsModel.nsType` in Section 9.3. The UML diagram of the namespace `nsModel` in Section

9.1 shows how the `Type` is connected to the geometric object (each NodeContainer has a Type).
Let's consider, for example, a quad element with bilinear shape functions and 2 integration points per coordinate direction. This Type would be created in the following way:

```
shape_order = 1;
shape_type = 'quad';
num_int_points = 2;

element_type = nsModel.nsType.ElementType(shape_order, shape_type,
    num_int_points);
```

Exercise Follow the instruction in the script `testElementType.m` in order to determine the positions of the integration points of a quad element with node coordinates (1/2), (4/1), (4/5) und (1/3). Two integration points per coordinate directions are used.
Hint: Use the property `NodeContainer.int_points` (elements are NodeContainers).

5 Exercise 5 - Jacobian Matrix

The Jacobian matrix is essential for calculating the stiffness matrices and force vectors. It is represented by a class on its own. Each integration point gets an object of the class `nsAnalyzer.nsJacobian.Jacobian`. The class is shown in the UML diagrams `nsAnalyzer.nsJacobian.Jacobian` and `nsAnalyzer.nsImplementation.ElementImpl` (Section 9.4). The two most important methods of this class are:

- `Jacobian.getMatrix()`: Calculates the Jacobian matrix. This is needed for transforming the derivatives with respect to the global coordinates (x, y) into derivatives with respect to the natural coordinates (ξ, η).
- `Jacobian.getDetInv()`: Calculates the determinant of the inverse Jacobian matrix. This is needed for transforming the integral itself onto the reference element.

Exercise Complete the script `testJacobian.m` in order to calculate the area of the element from Exercise 4 using numerical integration.
Hint: The area of an element is $A = \int_\Omega 1 dV = \int_{\Omega_\text{ref}} \det J^{-1} dV$.

6 Exercise 6 - Element Stiffness Matrix

6.1 Exercise 6a - Material Law

The last missing building block of the element stiffness matrix is the material law (elasticity tensor). For the linear elastic analysis, we use the isotropic St.-Venant-Kirhhoff material, which is equal to Hooke's law. The elasticity tensor of this material reads

$$C_{ijkl} = \lambda\delta_{ij}\delta_{kl} + \mu\left(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}\right) .$$

The Lamé parameters λ and ν are connected to Young's modulus E and Poisson's ratio ν in the following way (for 2d this is only valid for plane strain, plane stress is not implemented):

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)} \qquad \mu = \frac{E}{2(1+\nu)}$$

The classes describing the material behaviour can be found in the namespace `nsModel.nsMaterial` and are depicted in the UML diagram in Section 9.5.

Exercise In the script `testMaterial` a St.-Venant-Kirchhoff material with $E = 210\,000\,\mathrm{N/mm^2}$ and $\nu = 0.3$ is created. Calculate the elasticity tensor of this material for the 3d case and check the two minor symmetries and the major symmetry:

$$C_{ijkl} = C_{jikl} \qquad C_{ijkl} = C_{ijlk} \qquad C_{ijkl} = C_{klij}$$

6.2 Exercise 6b - Calculation of Element Stiffness Matrix

Now we can finally calculate the element stiffness matrix. It reads

$$A_{ijkl} = \int_{\Omega^{ref}} C_{imkn} N^l{}_{,o} J_{on} N^j{}_{,p} J_{pm} \det J^{-1} \, \mathrm{d}V \, . \tag{1}$$

Exercise Complete the script `testStiffness.m` (see instructions therein) in order to calculate the stiffness matrix of the element from Exercise 4.
Hint: If you get stuck, have a look at the method `stiffnessMatrixIntegrator` of the class `nsAnalyzer.nsImplementation.LinearElementImpl`.

Solution: The stiffness matrix of this element reads

```
A =

   1.0e+05 *

    1.4481    0.3059   -0.5303    0.0245   -0.4120   -0.4283   -0.5058    0.0979
    0.3059    1.9478   -0.1774    0.4385   -0.4283   -0.7078    0.2998   -1.6786
   -0.5303   -0.1774    1.1422   -0.6119    0.2713    0.2080   -0.8832    0.5813
    0.0245    0.4385   -0.6119    1.1524    0.0061   -0.7485    0.5813   -0.8424
   -0.4120   -0.4283    0.2713    0.0061    0.6710    0.3977   -0.5303    0.0245
   -0.4283   -0.7078    0.2080   -0.7485    0.3977    1.0178   -0.1774    0.4385
   -0.5058    0.2998   -0.8832    0.5813   -0.5303   -0.1774    1.9193   -0.7037
    0.0979   -1.6786    0.5813   -0.8424    0.0245    0.4385   -0.7037    2.0825
```

6.3 Exercise 6c - LinearElementImplementation

The calculation of the integrals appearing in the weak form is defined in the namespace `nsAnalyzer.nsImplementation`.
The class `nsAnalyzer.nsImplementation.LinearElementImplementation` consists of two methods. Both are used for the calculation of the element stiffness matrix. We already implemented the calculation of the element stiffness matrix in the previous Exercise 6b.
The script `testLinearElementImpl.m` tests the calculation of the element stiffness matrix using the class `LinearElementImplementation`.
Volume forces would also be implemented here, since the integration of volume forces is done over the inside domain of the element.
Surface forces are calculated with integrals over the face elements on the boundary and would be implemented in a class
`nsAnalyzer.nsImplementation.LinearBoundaryImplementation`.

Exercise Run the script `testLinearElementImpl.m`. Compare the results with the results from Exercise 6b.

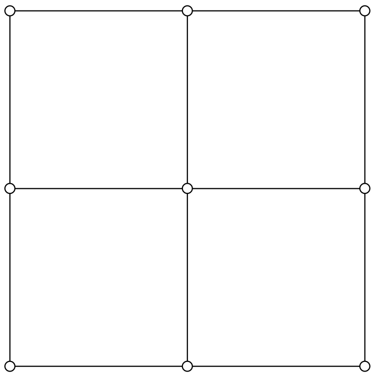
7 Exercise 7 - DOF-Degrees Of Freedom

Until now, we considered the following parts of the program: Geometry (Model), shape functions (Shape), numerical integration (NumInt, IntegrationPoint) and calculation of element stiffness matrices (ElementImplementation, Jacobian).
In order to implement the Dirichlet boundary conditions (= displacement boundary conditions) *and* in order to save the node displacements after solving the linear system of equations, we introduce the class `nsModel.nsDOF.DisplacementDOF`. It assigns the DOFs (**D**egree **O**f **F**reedom) to the nodes. Each node gets its own object of the class `DisplacementDOF`. The two properties of this class are (see UML diagram in Section 9.6):

- `displacement`: Value of the displacement
- `constraint`: `true` or `false`, depending on whether the DOF is defined through a Dirichlet boundary condition or not.

For setting the boundary conditions, we use the class `nsModel.BCHandler`. For each example/problem to solve, we define a concrete class `MyBCHandler` which is derived from the abstract class `nsModel.BCHandler` and overrides the abstract method `BCHandler.incorporateBC`. In this method, the boundary conditions of the system are defined (see Exercise 7).

Exercise In the script `testDOF`, a Model is created. Afterwards, the Dirichlet boundary conditions are defined and written into the DOFs of the model. Run the script `testDOF`. Below you find a sketch of the finite element mesh of this example. Write to each node whether it is constrained via a Dirichlet boundary condition or not. How would this body look like in a deformed state?



8 Exercise 8 - Analysis

Now that we implemented the DOFs, our model is ready. The class `nsAnalyzer.nsAnalysis.Analysis` performs the actual finite element analysis. It provides little helper functions for assembling the global stiffness matrix and force vector, and plugging in the Dirichlet boundary conditions into the linear system of equations. In this lecture, we don't mind how these helpers work.
The properties and methods of this class are shown in the UML diagram in Section 9.7 and are described below.
The analysis is started by calling the only public method `Analysis.run`. In this method, the following steps are processed:

1. The initial state of the model is exported in the `.vtk` format using the output handler. The VTK output handler is the counterpart of the Gmsh input handler and will be discussed in the next example.
2. The Dirichlet boundary conditions are written to the model's `DOF` objects using the BC handler.
3. The actual FE calculation is performed by calling the method `nsAnalyzer.nsAnalysis.Analysis.solveFESystem`. The individual steps are:
   a) Creating the empty global stiffness matrix and the empty global force vector
   b) Calculating the element stiffness matrices and assembling them into the global stiffness matrix
   c) Calculating the element force vectors and assembling them into the global force vector. The force vector contains Neumann boundary conditions (=surface forces) and possibly volume forces
   d) Plugging in the Dirichlet boundary conditions into the linear system of equations
   e) Solving the linear equation system with a Matlab-internal solver

   f) Saving the node displacements into the DOFs objects of each node of the model.

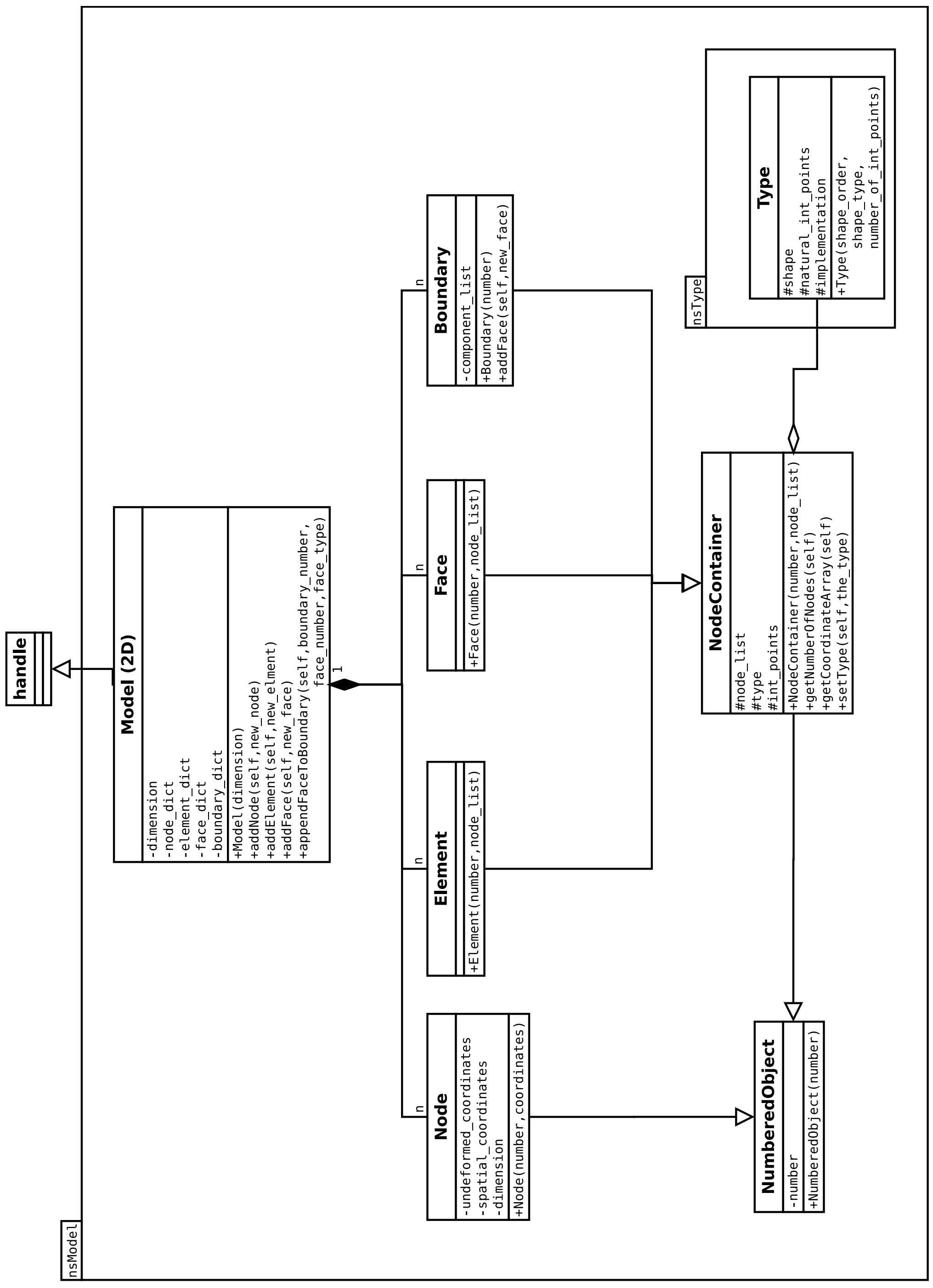
4. After the nodal displacements are saved to the DOF objects of the model in the method `solveFESystem`, the model is exported again in `.vtk` format.

Exercise Since all classes of the program have been treated, a 'real' example can be calculated. Run the script `soofeam` in order to get familiar with the class `nsAnalyzer.nsAnalysis.Analysis`.
To comprehend the flow of the programm better, you can use the Matlab debugger. If you have any questions, consult your teacher or the Matlab documentation: `https://www.mathworks.com/help/matlab/matlab_prog/debugging-process-and-features.html`
The output files are written into the folder `examples/plateAnalysis/results` and can be viewed with `ParaView` (`https://www.paraview.org/download/`).
For an introduction to ParaView, see this video (starting at 07:20)[3].

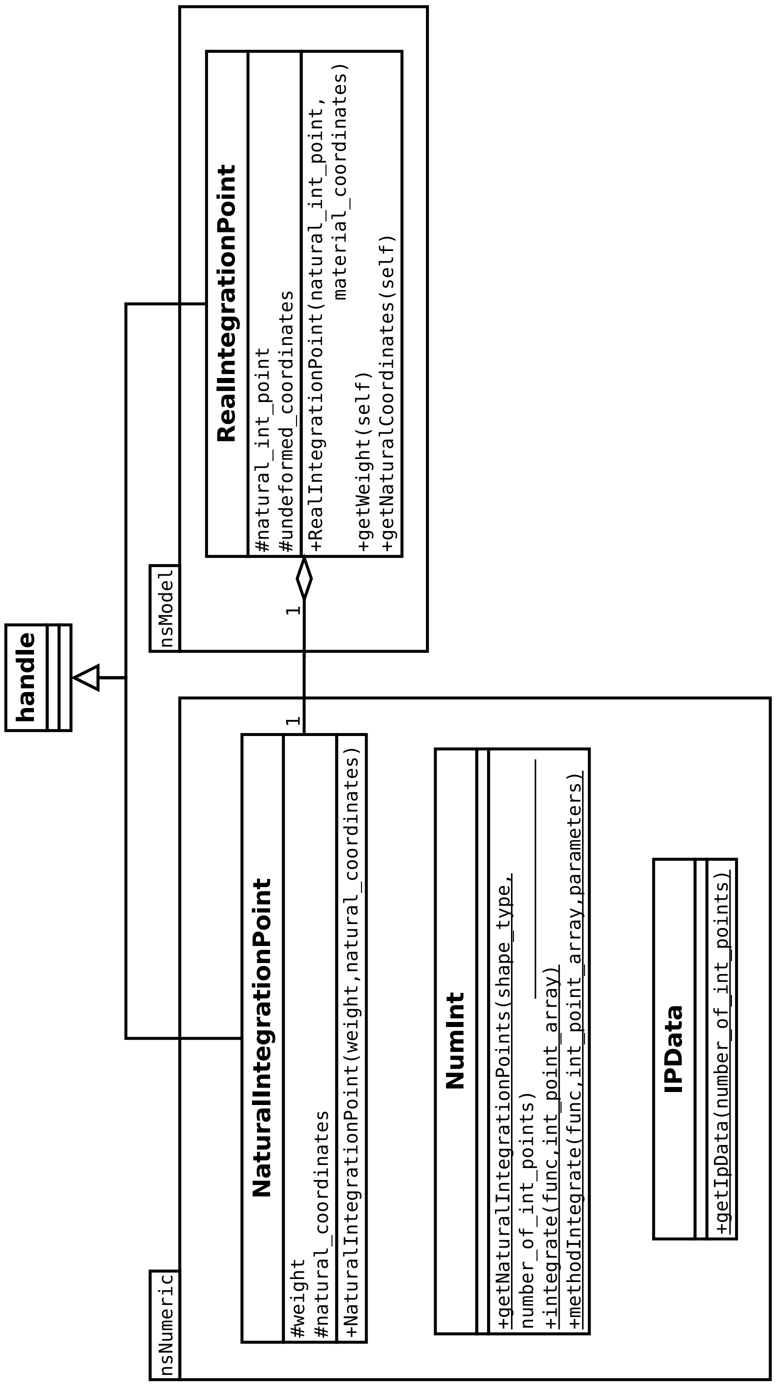
[3]`https://tube.tugraz.at/paella/ui/watch.html?id=be005a1b-b8fa-497b-b5fa-b931ad29a85b`

9 Diagrams

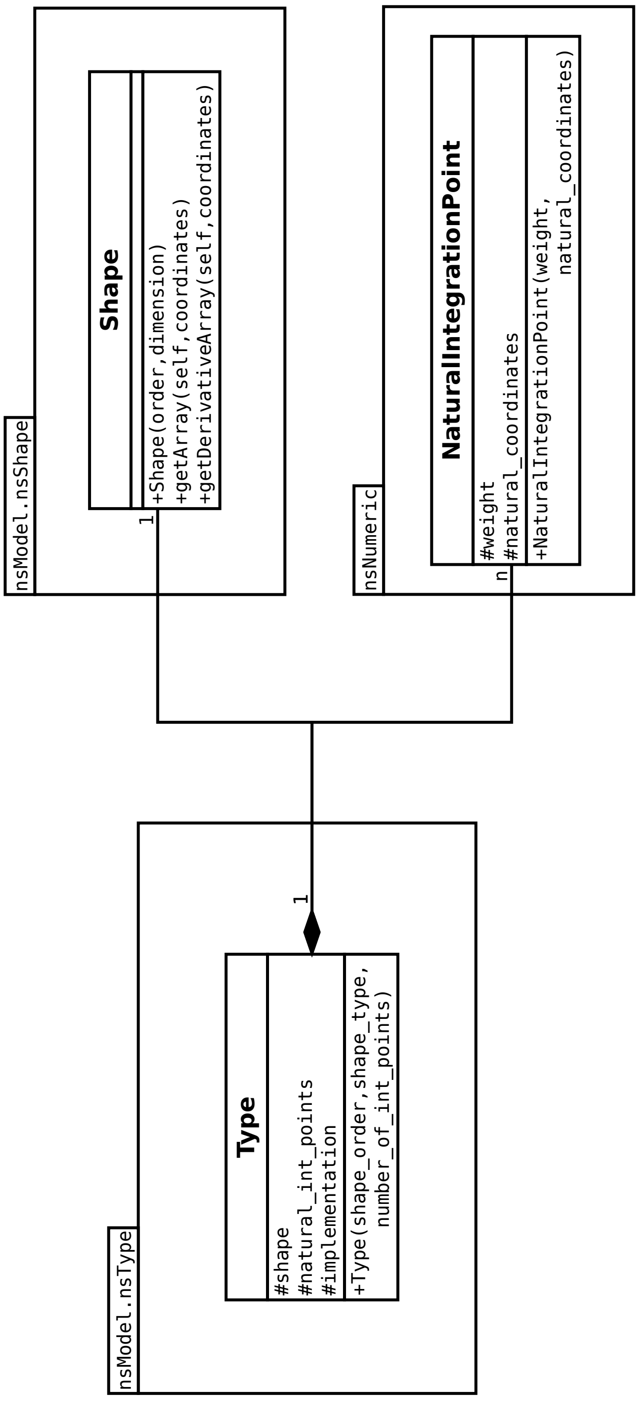
9.1 UML-Diagram nsModel



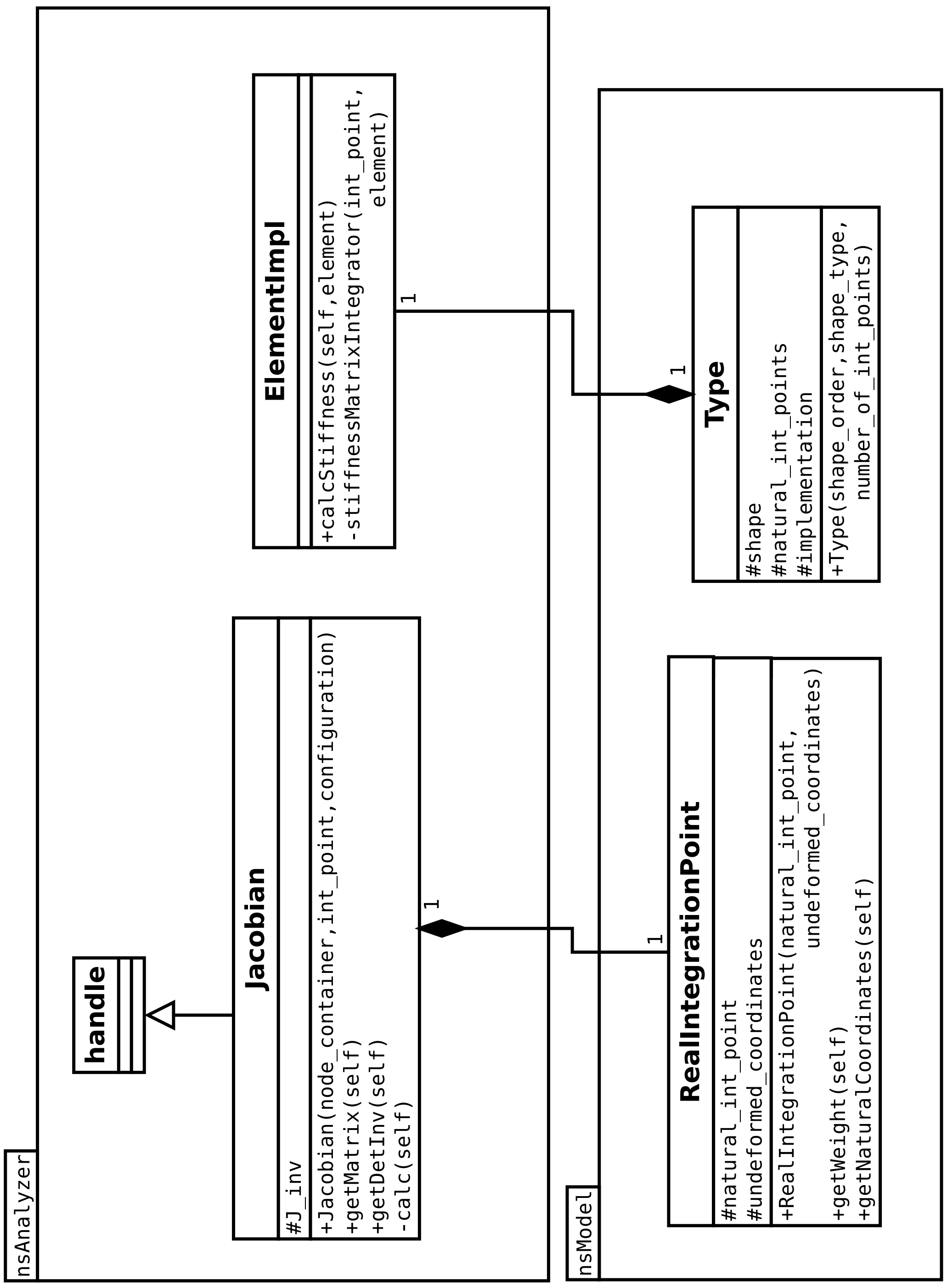
9.2 UML-Diagram nsNumeric


handle
nsModel
RealIntegrationPoint
#natural_int_point
#undeformed_coordinates
+RealIntegrationPoint(natural_int_point, material_coordinates)
+getWeight(self)
+getNaturalCoordinates(self)
1
1
nsNumeric
NaturalIntegrationPoint
#weight
#natural_coordinates
+NaturalIntegrationPoint(weight,natural_coordinates)
NumInt
+getNaturalIntegrationPoints(shape_type, number_of_int_points)
+integrate(func,int_point_array)
+methodIntegrate(func,int_point_array,parameters)
IPData
+getIpData(number_of_int_points)

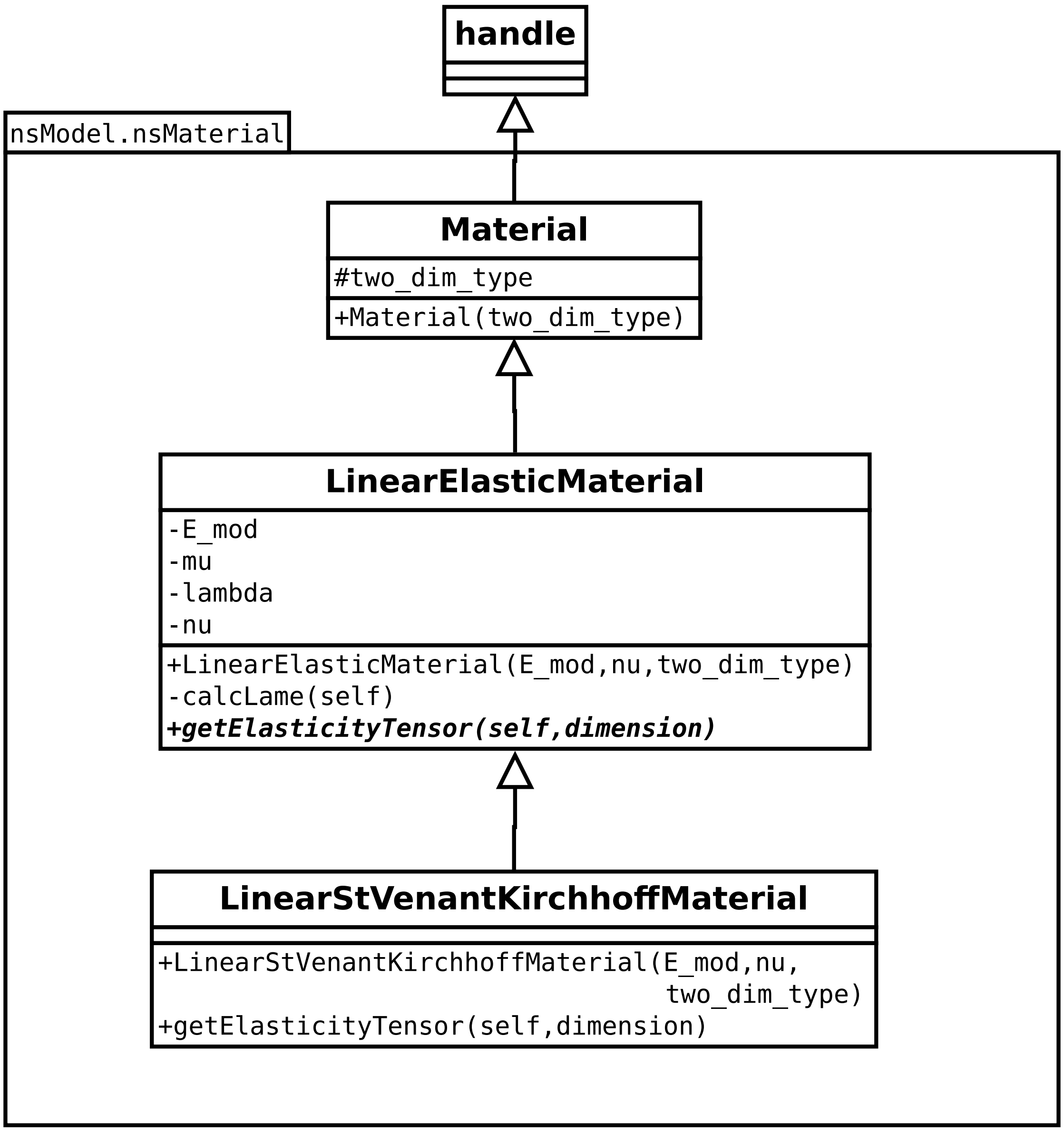
9.3 UML-Diagram nsModel.nsType


nsModel.nsShape
Shape
+Shape(order,dimension)
+getArray(self,coordinates)
+getDerivativeArray(self,coordinates)
1
nsNumeric
NaturalIntegrationPoint
#weight
#natural_coordinates
+NaturalIntegrationPoint(weight, natural_coordinates)
n
1
nsModel.nsType
Type
#shape
#natural_int_points
#implementation
+Type(shape_order,shape_type, number_of_int_points)

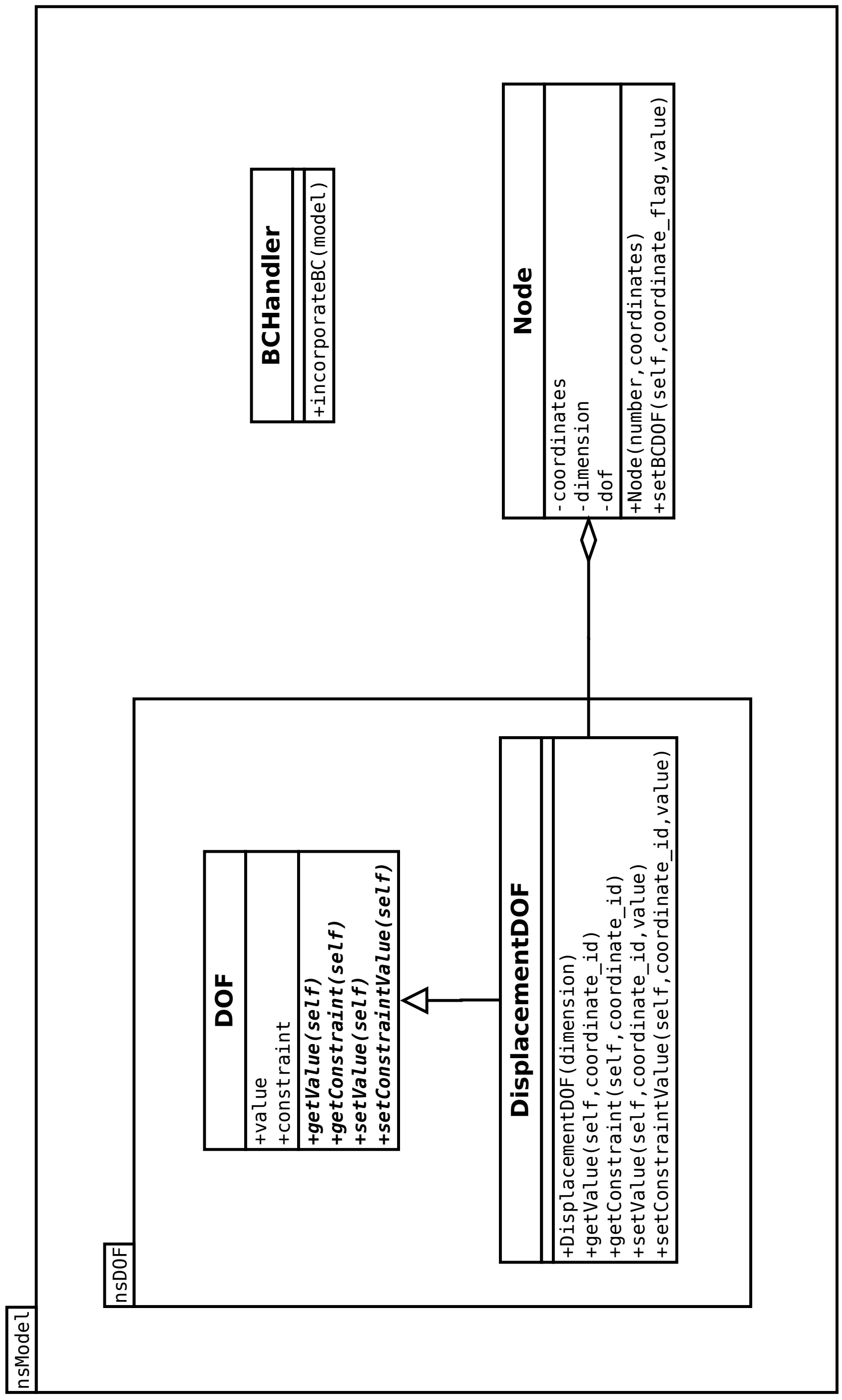
9.4 UML-Diagram nsAnalyzer.nsJacobian and nsAnalyzer.nsImplementation


nsAnalyzer
handle
Jacobian
#J_inv
+Jacobian(node_container,int_point,configuration)
+getMatrix(self)
+getDetInv(self)
-calc(self)
ElementImpl
+calcStiffness(self,element)
-stiffnessMatrixIntegrator(int_point, element)
1
1
nsModel
Type
#shape
#natural_int_points
#implementation
+Type(shape_order,shape_type, number_of_int_points)
1
1
RealIntegrationPoint
#natural_int_point
#undeformed_coordinates
+RealIntegrationPoint(natural_int_point, undeformed_coordinates)
+getWeight(self)
+getNaturalCoordinates(self)

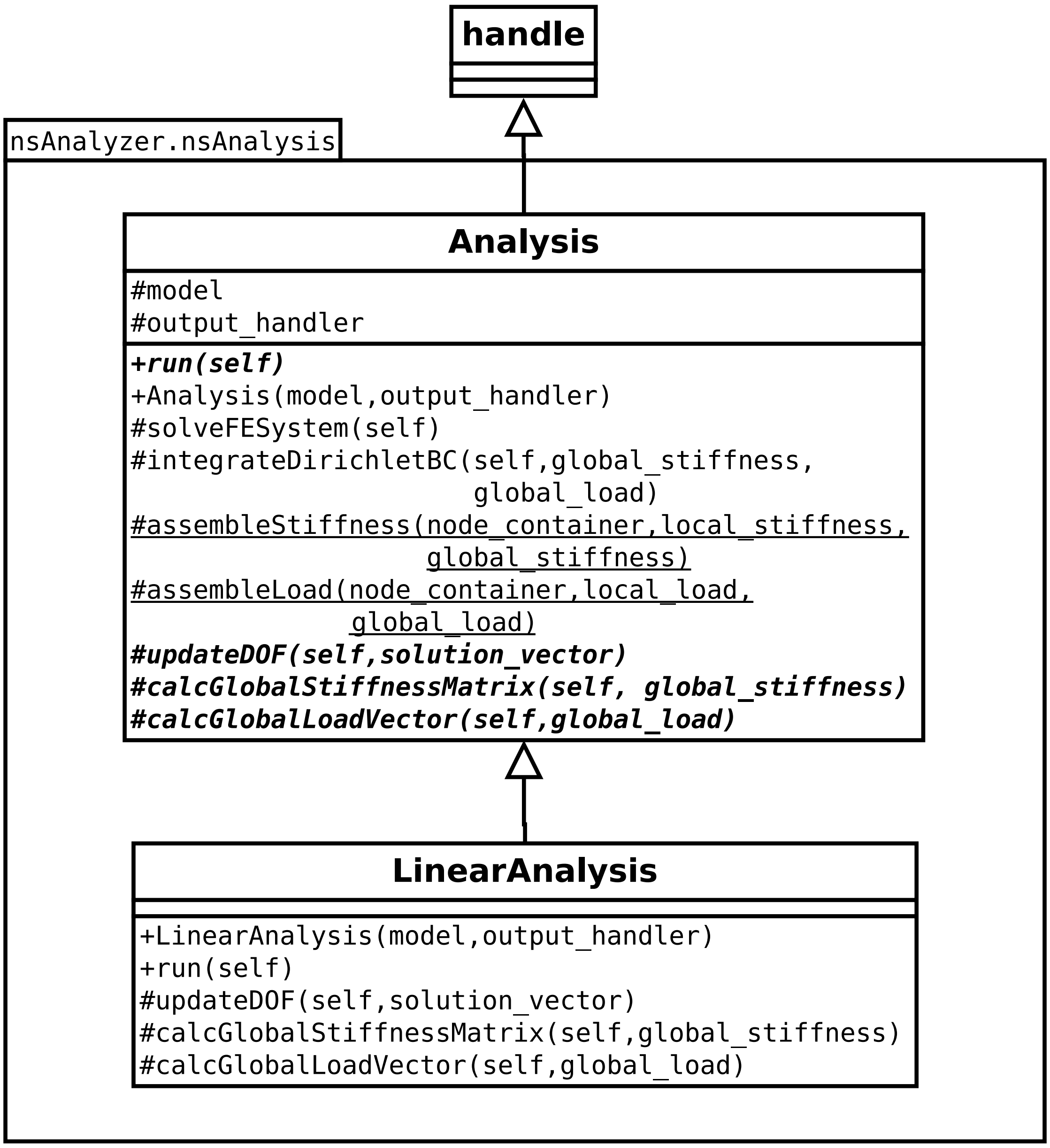
9.5 UML-Diagramm nsModel.nsMaterial


handle
nsModel.nsMaterial
Material
#two_dim_type
+Material(two_dim_type)
LinearElasticMaterial
-E_mod
-mu
-lambda
-nu
+LinearElasticMaterial(E_mod,nu,two_dim_type)
-calcLame(self)
+getElasticityTensor(self,dimension)
LinearStVenantKirchhoffMaterial
+LinearStVenantKirchhoffMaterial(E_mod,nu, two_dim_type)
+getElasticityTensor(self,dimension)

9.6 UML-Diagramm nsModel.nsDOF


nsModel
BCHandler
+incorporateBC(model)
Node
-coordinates
-dimension
-dof
+Node(number,coordinates)
+setBCDOF(self,coordinate_flag,value)
nsDOF
DOF
+value
+constraint
+getValue(self)
+getConstraint(self)
+setValue(self)
+setConstraintValue(self)
DisplacementDOF
+DisplacementDOF(dimension)
+getValue(self,coordinate_id)
+getConstraint(self,coordinate_id)
+setValue(self,coordinate_id,value)
+setConstraintValue(self,coordinate_id,value)

9.7 UML-Diagramm nsAnalyzer.nsAnalysis


handle
nsAnalyzer.nsAnalysis
Analysis
#model
#output_handler
+run(self)
+Analysis(model,output_handler)
#solveFESystem(self)
#integrateDirichletBC(self,global_stiffness, global_load)
#assembleStiffness(node_container,local_stiffness, global_stiffness)
#assembleLoad(node_container,local_load, global_load)
#updateDOF(self,solution_vector)
#calcGlobalStiffnessMatrix(self, global_stiffness)
#calcGlobalLoadVector(self,global_load)
LinearAnalysis
+LinearAnalysis(model,output_handler)
+run(self)
#updateDOF(self,solution_vector)
#calcGlobalStiffnessMatrix(self,global_stiffness)
#calcGlobalLoadVector(self,global_load)