ADVANCES IN ENGINEERING SOFTWARE 32, 8, 611-628(2001)

Object-Oriented Nonlinear Finite Element Programming: a Primer

Stéphane Commend, Thomas Zimmermann

Laboratory of Structural and Continuum Mechanics (LSC)
Swiss Federal Institute of Technology, 1015 Lausanne, Switzerland

Abstract

This article describes an introductory object-oriented finite element program for static and dynamic nonlinear applications. This work can be considered as an extension of the original FEM_Object environment dealing with linear elasticity [1] and nonlinearity [2]. Mainly the static aspects are discussed in this paper. Interested readers will find a detailed discussion of the object-oriented approach applied to finite element programming in [15-18] and also in [7-8] and references therein. Our ambition, in this paper, is limited to a presentation of an introductory object-oriented finite element package for nonlinear analysis. Our goal is to make a starting package available to newcomers to the object-oriented approach and to provide an answer to the large number of demands for such a program received in recent time.

In the first part of the paper, a brief recall of the basics of finite element modeling applied to continuum mechanics is given. Von Misès plasticity including isotropic and kinematic hardening, which is used as model problem, is described. This first part also presents an overview of the main features of the object-oriented approach. In the second part of this paper, classes and associated tasks forming the kernel of the code are described in detail. A hierarchy of classes is proposed and discussed; it provides an immediate overview of the program's capabilities. Finally interactions between classes are explained and numerical examples illustrate the approach.

1 Introduction

1.1 The static boundary value problem

The strong form of the problem can be stated as: find $\mathbf{u}$ such that:

$$L(\mathbf{u}) = \sigma_{ji,j} + f_i = 0 \quad \text{on } \Omega \tag{1}$$

with boundary conditions:

$$\sigma_{ji} n_j = \bar{t}_i \quad \text{on } \Gamma_1 \tag{2}$$

$$u_i = \overline{u_i} \quad \text{on } \Gamma_2 \text{ and } \Gamma = \Gamma_1 + \Gamma_2 \tag{3}$$

with the kinematic relation:

$$\varepsilon_{kl} = \frac{1}{2}\left(u_{k,l} + u_{l,k}\right) = u_{(k,l)} \tag{4}$$

and the incremental elastoplastic constitutive equation:

$$d\sigma_{ij} = D^{ep}_{ijkl} d\varepsilon_{kl} \tag{5}$$

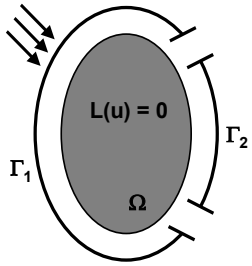

Figure 1: Problem statement

Although the program supports dynamic analysis, this will be not discussed in detail herein.

1.2 Von Misès plasticity

Plasticity requires the definition of a yield function, a flow rule and a hardening law; the consistency condition completes the formulation. Von Misès yield function is adopted here.

1.2.1 Von Misès criterion

This criterion assumes that plastic yielding will occur when the second invariant J_2 of the deviatoric stress tensor reaches a critical value, which is a material property. It is expressed by:

$$f(\boldsymbol{\sigma}) = \sqrt{J_2} - k = 0 \tag{6}$$

where:

$$J_2 = \frac{1}{2} s_{ij} s_{ij} \tag{7}$$

and $\mathbf{s}$ is the deviatoric stress tensor.

The criterion can be represented by a cylinder of radius R in the three-dimensional stress space, with:

$$R = \sqrt{2}k \tag{8}$$

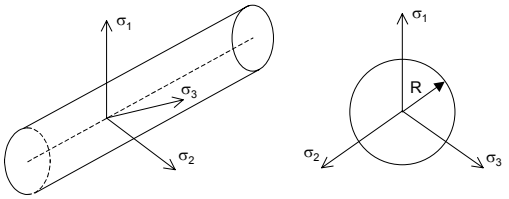

Figure 2: Von Misès yield surface in principal stress space and in the deviatoric plane

1.2.2 Flow rule

Let:

$$d\boldsymbol{\varepsilon}^{\mathbf{p}} = d\gamma \cdot \mathbf{r} \tag{9}$$

where $\mathbf{r}$ defines the plastic flow direction and $d\gamma$ the flow amplitude. Assuming associative flow, we determine the unit flow direction vector $\mathbf{r}(\boldsymbol{\sigma})$ as follows:

$$\mathbf{r}(\boldsymbol{\sigma}) = \mathbf{n}(\boldsymbol{\sigma}) = \frac{\dfrac{df}{d\boldsymbol{\sigma}}}{\left\| \dfrac{df}{d\boldsymbol{\sigma}} \right\|} = \frac{1}{\sqrt{2J_2}} s_{ij} \tag{10}$$

1.2.3 Hardening

Hardening defines an evolution law of the yield surface in the stress space:

$$f(\boldsymbol{\sigma}, \mathbf{q}) = 0 \tag{11}$$

where $\mathbf{q}$ is a set of hardening parameters which can be scalar or tensorial. A distinction is made between: isotropic hardening: the yield surface grows in size but its center in the deviatoric plane remains fixed, kinematic hardening: the radius of the yield surface remains constant, but its center is translated in the deviatoric plane, and mixed isotropic-kinematic hardening: combining the two previous ones.

Introducing two new variables, the back-stress $\boldsymbol{\alpha}$ (for kinematic hardening) and the yield radius R (for isotropic hardening), equation (11) can be written:

$$f(\boldsymbol{\sigma}, \boldsymbol{\alpha}, R) = \left\| \mathbf{s} - \boldsymbol{\alpha} \right\| - R = \left\| \boldsymbol{\xi} \right\| - R = 0 \tag{12}$$

where $\mathbf{s}$ is the deviatoric part of $\boldsymbol{\sigma}$, and (6) can be easily retrieved.

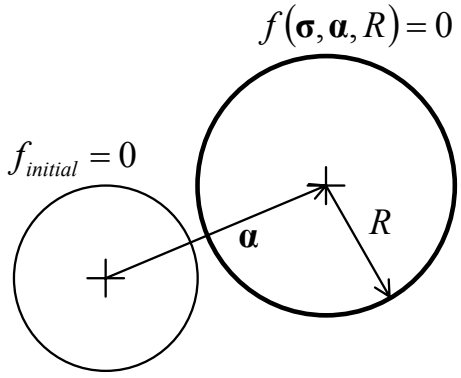

Figure 3: Evolution of the yield surface

If we consider a linear combination of kinematic and isotropic hardening, the evolution law (or hardening rule) for the set of hardening parameters $\mathbf{q} = (\boldsymbol{\alpha}, R)$ can be derived in the following way.

First, we need to relate these parameters to the experimental uniaxial stress-strain curve. For that, we use two variables, namely the equivalent (or effective) stress σ_{eq} and the equivalent (or effective) plastic strain ε_{eq}^{p} [3]:

$$\sigma_{eq} = \left(\frac{3}{2} s_{ij} s_{ij} \right)^{0.5} = \sqrt{3J_2} \tag{13}$$

$$\varepsilon_{eq}^{p} = \left(\frac{2}{3} \varepsilon_{ij}^{p} \varepsilon_{ij}^{p} \right)^{0.5} \tag{14}$$

We verify that for the uniaxial test:

$$\sigma_{eq} = \sqrt{3J_2} = \sqrt{3} \sqrt{\frac{1}{3}\sigma_{11}^2} = \sigma_{11} \tag{15}$$

and due to incompressibility in the plastic range (which leads to $\varepsilon_{22}^{p} = \varepsilon_{33}^{p} = -\frac{1}{2}\varepsilon_{11}^{p}$):

$$\varepsilon_{eq}^{p} = \varepsilon_{11}^{p} \tag{16}$$

In the case of linear hardening, the relation between these two variables can be expressed with the help of the plastic modulus H', as:

$$d\sigma_{eq} = H' d\varepsilon_{eq}^{p} \tag{17}$$

and we will assume in the sequel that $H' \geq 0$.

- Isotropic hardening

The amplitude of the plastic strain increment is by definition given by the plastic multiplier $d\gamma$. In the uniaxial case, we can therefore write:

$$\left\|d\boldsymbol{\varepsilon}^{\mathbf{p}}\right\| = \left[\left(d\varepsilon_{11}\right)^2 + \left(\frac{1}{2}d\varepsilon_{11}\right)^2 + \left(\frac{1}{2}d\varepsilon_{11}\right)^2 \right]^{0.5} = \sqrt{\frac{3}{2}}d\varepsilon_{11} = \sqrt{\frac{3}{2}}d\varepsilon_{eq}^{p} = d\gamma \tag{18}$$

For Von Misès criterion, equation (6) ($\sqrt{J_2} = k$) yields (with the use of equation (18)):

$$dk = d\sqrt{J_2} = \frac{1}{\sqrt{3}}d\sigma_{eq} = \frac{1}{\sqrt{3}}H'd\varepsilon_{eq}^{p} = \frac{\sqrt{2}}{3}H'd\gamma \tag{19}$$

and with equation (8), we finally get the expression for the evolution of the yield radius:

$$dR = \sqrt{2}dk = \frac{2}{3}H'd\gamma \tag{20}$$

- Kinematic hardening

If the radius of the yield surface remains constant, we can write it as:

$$d\sqrt{J_2^*} = 0 \;;\; J_2^* = \frac{1}{2}\left(s_{ij} - \alpha_{ij}\right)\left(s_{ij} - \alpha_{ij}\right) \tag{21), (22}$$

Equations (18), (21) and (22) yield:

$$\left\|d\boldsymbol{\alpha}\right\| = \left\|d\mathbf{s}\right\| = \sqrt{2}d\sqrt{J_2} = \sqrt{2}\frac{1}{\sqrt{3}}d\sigma_{eq} = \frac{\sqrt{2}}{\sqrt{3}}H'd\varepsilon_{eq}^{p} = \frac{2}{3}H'd\gamma \tag{23}$$

and finally, the evolution of the back-stress writes:

$$d\boldsymbol{\alpha} = \frac{2}{3}H'd\gamma\,\mathbf{r} \tag{24}$$

- Mixed hardening

A straightforward combination of the two types of hardening can be introduced wih a parameter β such that $\beta = 0$ for kinematic hardening, $\beta = 1$ for isotropic hardening and:

$$dR = \frac{2}{3}\beta H'd\gamma \tag{25}$$

$$d\boldsymbol{\alpha} = \frac{2}{3}\left(1-\beta\right)H'd\gamma\,\mathbf{r} \tag{26}$$

1.2.4 Consistency

The actual amplitude of plastic flow results from the consistency condition which imposes:

$$\dot{f} = 0 \tag{27}$$

under plastic loading. This we write in vector notation as:

$$\dot{f} = \frac{\partial f}{\partial \boldsymbol{\sigma}}^{T} d\boldsymbol{\sigma} + \frac{\partial f}{\partial \mathbf{q}}^{T} d\mathbf{q} = \frac{\partial f}{\partial \boldsymbol{\sigma}}^{T} \mathbf{D}^{\mathbf{el}}\left(d\boldsymbol{\varepsilon} - d\boldsymbol{\varepsilon}^{\mathbf{p}}\right) + \frac{\partial f}{\partial \mathbf{q}}^{T} d\mathbf{q} = 0 \tag{28}$$

where the incremental elastoplastic constitutive equation (5) has been rewritten as:

$$d\boldsymbol{\sigma} = \mathbf{D}^{\mathbf{el}}\, d\boldsymbol{\varepsilon}^{\mathbf{e}} = \mathbf{D}^{\mathbf{el}}\left(d\boldsymbol{\varepsilon} - d\boldsymbol{\varepsilon}^{\mathbf{p}}\right) \tag{29}$$

We will see later (equations (45) and (68)) that equation (28) leads to the actual computation of the plastic multiplier $d\gamma$; but for the case of a Von Misès criterion a more efficient numerical implementation can be formulated.

1.3 The finite element method

We consider here a displacement formulation of the elastoplastic matrix problem, which is stated as:

$$\mathbf{N}(\mathbf{d}) = \mathbf{F}^{\mathbf{ext}} \tag{30}$$

where $\mathbf{N}(\mathbf{d})$ is a nonlinear matrix function of the displacement vector $\mathbf{d}$ and $\mathbf{F}^{\mathbf{ext}}$ is the vector of applied forces. The resulting linearized problem (see figure 4) then reads:

$$\mathbf{K}_{\mathbf{T}}\Delta\mathbf{d} = \mathbf{F}_{\mathbf{n+1}}^{\mathbf{ext}} - \mathbf{N}\left(\mathbf{d}_{\mathbf{n+1}}^{\mathbf{i}}\right) \tag{31}$$

$$\mathbf{d}_{\mathbf{n+1}}^{\mathbf{i+1}} = \mathbf{d}_{\mathbf{n+1}}^{\mathbf{i}} + \Delta\mathbf{d} \tag{32}$$

which must be solved for $\Delta\mathbf{d}$ iteratively at each loading step $(n+1)$. $\mathbf{K}_{\mathbf{T}}$ is the tangent stiffness, i the iteration counter and $(n+1)$ the current loading step.

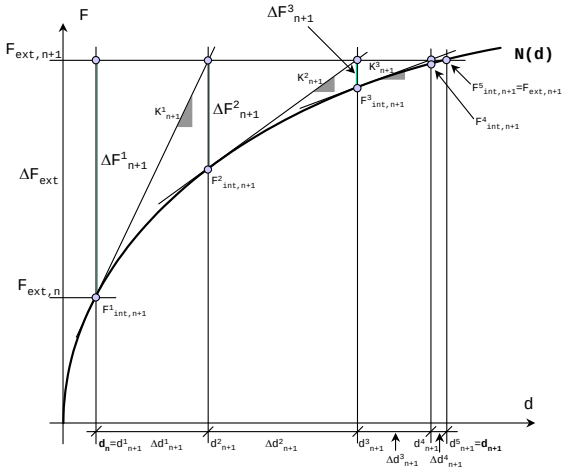

Figure 4: Linearized problem

1.4 Volumetric locking due to material incompressibility

Von Misès plasticity induces incompressible behavior in the plastic range, which in turn can induce locking phenomena.

Figure 5a illustrates a simple mesh discussed in [4] with three fixed nodes and one free node. Assume linear displacement constant pressure elements; incompressible (constant volume) deformation must take place. As a result of kinematic constraint node N is required to move horizontally in the lower triangular element, and required to move vertically as a result of the kinematic constraint in the upper triangular element; locking results, and obviously the reasoning can be extended to a n x n mesh (figure 5b). This kind of locking typically appears when using the full **B** matrix as described later. Different methods can be used in order to avoid this phenomenon [5]. Keeping in mind the incompressible behavior of Von Misès material, the $\overline{\mathbf{B}}$ approach can be applied here.

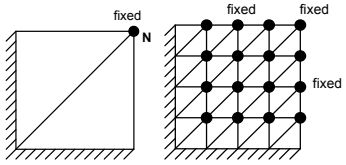

Figure 5a Figure 5b

1.5 Why object-oriented programming?

Object-oriented programming (see e.g. [6], [7], [8] and references therein) has proven in recent years to be one of the easiest, fastest and most efficient ways to program robust scientific software. The basic components of the finite element method, like the node, the element, the material, can easily be fitted into an objects world, with their own behavior, state and identity. We review here the key features of object-oriented programming:

a) Robustness and modularity: encapsulation of data

An object is a device capable of performing predefined actions, such as storing information (in its variables), executing tasks (through his methods), or accessing other objects (by sending messages). Variables describe the state of the object, while methods define its behavior. Objects hide their variables from other components of the application. For instance, class Element does not have direct access to its Young Modulus. The Young Modulus is stored in class Material. The object has to send a message, like *myMaterial→giveYoungModulus()* to access it.

b) Inheritance and polymorphism: the hierarchy of classes

Every object is an instance of a class. A class is an abstract data type which can be considered as the mold of the object. Classes are organised within a hierarchy (class-subclass), which allows a subclass (say, Truss2D) to inherit the methods and variables from its superclass (say, Element). Polymorphism expresses the fact that two different classes will react differently (in their own manner) to the same message. For instance, the message *myElement→ giveBMatrix()* will be interpreted differently by an object of the class Quad_U (defining quadrilateral elements) and an object of the class Truss2D (defining truss elements).

The hierarchy of classes of a simple nonlinear finite element code, providing an overview of the entire software, is given in section 3. The fact that the code can be described in such a compact way can be very valuable, when extensions are considered.

c) Non-anticipation and state encapsulation

Non-anticipation expresses the fact that the content of a method should not rely on any assumption on the state of the variables. Strict obedience to non-anticipation will contribute significantly to code robustness.

d) Efficiency

As far as numerical performance is concerned, languages such as C++ have shown performances similar to Fortran. With respect to code development speed using object-oriented techniques, the programmer can maximize reusability of the software and «program like he thinks», which leads to faster prototyping.

2 Main tasks of the finite element program

2.1 Element level (class `Element`)

2.1.1 Forming the elementary stiffness

The definition of the elemental stiffness matrix $\mathbf{K_e}$ writes:

$$\mathbf{K_e} = \int_e \mathbf{B}^T \mathbf{D}^{\mathbf{ep}} \mathbf{B}\, dV_e \tag{33}$$

The following method of class Element illustrates how the element is forming its stiffness matrix.

```
//---------------------------------------------------------
FloatMatrix*  Element :: computeTangentStiffnessMatrix ()
//---------------------------------------------------------

{
Material         *mat;
GaussPoint       *gp;
FloatMatrix      *b,*db,*d;
double           dV;
int              i;
if (stiffnessMatrix) {
        delete stiffnessMatrix;
}
stiffnessMatrix = new FloatMatrix();
mat    = this->giveMaterial();
for (i=0; i<numberOfGaussPoints; i++) {
        gp = gaussPointArray[i];
        b  = this->ComputeBmatrixAt(gp);
        d  = mat->ComputeConstitutiveMatrix(gp,this);
        dV = this->computeVolumeAround(gp);
        db = d->Times(b);
        stiffnessMatrix->plusProduct(b,db,dV);
        delete d; delete db; delete b;
}
return stiffnessMatrix->symmetrized();
}
```

This method shows that the construction of two matrices, namely $\mathbf{D^{ep}}$ and $\mathbf{B}$, contribute to build the stiffness matrix. Integration over the element is achieved with a loop over the Gauss points of the element. In sections 2.1.1.1 and 2.1.1.2, the construction of $\mathbf{B}$ (or $\overline{\mathbf{B}}$) matrix will be scrutinized. In

section 2.1.1.3, the construction of constitutive matrix $\mathbf{D}^{ep}$ will be discussed.

2.1.1.1 The $\mathbf{B}$ matrix (class `Quad_U`)

The $\mathbf{B}$ matrix defines the kinematic relation between the strain vector $\boldsymbol{\varepsilon}$ and the nodal displacements $\mathbf{d}$:

$$\boldsymbol{\varepsilon} = \mathbf{B}\mathbf{d} \tag{34}$$

$$\varepsilon_{ij} = \frac{1}{2}(u_{i,j} + u_{j,i}) \tag{35}$$

Considering a plane strain bilinear isoparametric quadrilateral element, this relation can be stated starting from the definition of $\boldsymbol{\varepsilon}$, and approximating the displacement field $\mathbf{u}$ with the help of the nodal displacements $\mathbf{d}$ and the interpolation functions N_a:

$$u(\xi,\eta) = \sum_{a=1}^{4} N_a(\xi,\eta) d_a \tag{36}$$

The corresponding definition of the $\mathbf{B}$ matrix can be written following [9], notice the 4x2 matrix size:

$$\mathbf{B} = \begin{bmatrix} \mathbf{B}_1 & \mathbf{B}_2 & \mathbf{B}_3 & \mathbf{B}_4 \end{bmatrix} \; with \; \mathbf{B}_i = \begin{bmatrix} \frac{\partial N_i}{\partial x_1} & 0 \\ 0 & \frac{\partial N_i}{\partial x_2} \\ \frac{\partial N_i}{\partial x_2} & \frac{\partial N_i}{\partial x_1} \\ 0 & 0 \end{bmatrix} \tag{37}$$

The partial derivatives of the interpolation functions are calculated using the inverse of the Jacobian matrix $\mathbf{J}^{-1}$:

$$\begin{bmatrix} \frac{\partial N_a}{\partial x} & \frac{\partial N_a}{\partial y} \end{bmatrix} = \begin{bmatrix} \frac{\partial N_a}{\partial \xi} & \frac{\partial N_a}{\partial \eta} \end{bmatrix} \frac{1}{j} \begin{bmatrix} \frac{\partial y}{\partial \eta} & -\frac{\partial x}{\partial \eta} \\ -\frac{\partial y}{\partial \xi} & \frac{\partial x}{\partial \xi} \end{bmatrix} \tag{38}$$

$$j = \frac{\partial x}{\partial \xi}\frac{\partial y}{\partial \eta} - \frac{\partial x}{\partial \eta}\frac{\partial y}{\partial \xi} \tag{39}$$

2.1.1.2 The $\bar{\mathbf{B}}$ matrix (class `Quad_U_BBar`)

The $\bar{\mathbf{B}}$ approach introduces a modification of the dilatational contribution to the standard $\mathbf{B}$ matrix, $\mathbf{B}_{dil}$, which is underintegrated or averaged over the element.

The strain relation then reads (see [4] for details):

$$\boldsymbol{\varepsilon} = \bar{\mathbf{B}}\mathbf{d} \tag{40}$$

$$\bar{\mathbf{B}} = \mathbf{B}_{dev} + \bar{\mathbf{B}}_{dil} \quad \text{and} \quad \mathbf{B}_{dev} = \mathbf{B} - \mathbf{B}_{dil} \tag{41}$$

For plane strain $\bar{\mathbf{B}}$ results as:

$$\begin{bmatrix} \frac{\partial N_i}{\partial x_1} + B_4 & B_6 \\ B_4 & \frac{\partial N_i}{\partial x_2} + B_6 \\ \frac{\partial N_i}{\partial x_2} & \frac{\partial N_i}{\partial x_1} \\ B_4 & B_6 \end{bmatrix} \; with \; B_4 = \frac{\frac{\partial \bar{N}_i}{\partial x_1} - \frac{\partial N_i}{\partial x_1}}{3} \quad B_6 = \frac{\frac{\partial \bar{N}_i}{\partial x_2} - \frac{\partial N_i}{\partial x_2}}{3} \tag{42}$$

This formulation has been shown to be appropriate for overcoming locking due to incompressibility in very general situations.

2.1.1.3 Constitutive matrices (class `Material` and its subclasses)

Different algorithms use different constitutive matrices. The elastic stiffness matrix based on the elastic constitutive matrix (section 2.1.1.3.1) is used for a constant stiffness algorithm, while the Newton-Raphson algorithm requires a tangent stiffness matrix (section 2.1.1.3.2) or even a consistent tangent stiffness matrix (section 2.1.1.3.3) in order to improve convergence.

2.1.1.3.1 The elastic constitutive matrix $\mathbf{D}^{el}$

Class ElasticMaterial forms its constitutive matrix in the following method:

```
//-----------------------------------------------------------
FloatMatrix* ElasticMaterial :: ComputeConstitutiveMatrix
                              (GaussPoint* ip, Element* elem)
//-----------------------------------------------------------

{
return elem->giveConstitutiveMatrix()->GiveCopy();
}
```

This method actually calls a method of class element (here, class Quad_U):

```
//-------------------------------------------------
FloatMatrix*  Quad_U :: computeConstitutiveMatrix ()
//-------------------------------------------------

{
Material *mat = this -> giveMaterial() ;
double   e,nu,ee;
e     = mat -> give('E') ;
nu    = mat -> give('n') ;
ee    = e / ((1.+nu) * (1.-nu-nu)) ;
constitutiveMatrix = new FloatMatrix(4,4) ;
constitutiveMatrix->at(1,1) = (1.-nu) * ee ;
constitutiveMatrix->at(1,2) =     nu  * ee ;
constitutiveMatrix->at(2,1) =     nu  * ee ;
constitutiveMatrix->at(2,2) = (1.-nu) * ee ;
constitutiveMatrix->at(3,3) = e / (2.+nu+nu) ;
constitutiveMatrix->at(1,4) =     nu  * ee ;
constitutiveMatrix->at(2,4) =     nu  * ee ;
constitutiveMatrix->at(4,1) =     nu  * ee ;
constitutiveMatrix->at(4,2) =     nu  * ee ;
constitutiveMatrix->at(4,4) = (1.-nu) * ee ;
return constitutiveMatrix ;
}
```

2.1.1.3.2 The elasto-plastic tangent constitutive matrix $\mathbf{D}^{ep}$

Definition:

$$\mathbf{D}^{\mathbf{ep}} = \frac{d\boldsymbol{\sigma}}{d\boldsymbol{\varepsilon}} \tag{43}$$

Consistency imposes $\dot{f} = 0$, with f the yield function, which yields in vector notation:

$$\frac{df}{d\boldsymbol{\sigma}}^T d\boldsymbol{\sigma} = \frac{df}{d\boldsymbol{\sigma}}^T \mathbf{D}^{\mathbf{el}} d\boldsymbol{\varepsilon} - \frac{df}{d\boldsymbol{\sigma}}^T \mathbf{D}^{\mathbf{el}} d\boldsymbol{\varepsilon}^{\mathbf{p}} = 0 \tag{44}$$

Introducing the flow rule (equation (9)) into (44), one gets:

$$d\gamma = \frac{\frac{df}{d\boldsymbol{\sigma}}^T \mathbf{D}^{\mathbf{el}} d\boldsymbol{\varepsilon}}{\frac{df}{d\boldsymbol{\sigma}}^T \mathbf{D}^{\mathbf{el}} \mathbf{r}} \tag{45}$$

Introducing (45) into (9) and then into the constitutive equation (29) we get:

$$d\boldsymbol{\sigma} = \mathbf{D}^{\mathbf{el}} d\boldsymbol{\varepsilon} - \mathbf{D}^{\mathbf{el}} \frac{\frac{df}{d\boldsymbol{\sigma}}^T \mathbf{D}^{\mathbf{el}} d\boldsymbol{\varepsilon}}{\frac{df}{d\boldsymbol{\sigma}}^T \mathbf{D}^{\mathbf{el}} \mathbf{r}} \mathbf{r} \tag{46}$$

And finally:

$$d\boldsymbol{\sigma} = \mathbf{D}^{\mathbf{ep}} d\boldsymbol{\varepsilon} = \left[\mathbf{D}^{\mathbf{el}} - \frac{\left(\mathbf{D}^{\mathbf{el}} \mathbf{r}\right)\left(\mathbf{D}^{\mathbf{el}} \frac{df}{d\boldsymbol{\sigma}}\right)^T}{\frac{df}{d\boldsymbol{\sigma}}^T \mathbf{D}^{\mathbf{el}} \mathbf{r}} \right] d\boldsymbol{\varepsilon} \tag{47}$$

2.1.1.3.3 Improving convergence: the consistent elasto-plastic tangent constitutive matrix $\mathbf{D}^{\mathbf{ep*}}$

Using the following algorithmic approximation of the plastic strain increment (with q the plastic potential):

$$d\boldsymbol{\varepsilon}^{\mathbf{p}} = d\gamma \cdot \mathbf{r} + \gamma \cdot d\mathbf{r} = d\gamma \frac{dq}{d\boldsymbol{\sigma}} + \gamma \frac{d^2 q}{d\boldsymbol{\sigma}^2} d\boldsymbol{\sigma} \tag{48}$$

One gets:

$$d\boldsymbol{\sigma} = \mathbf{D}^{\mathbf{el}} \left(d\boldsymbol{\varepsilon} - d\gamma \frac{dq}{d\boldsymbol{\sigma}} - \gamma \frac{d^2 q}{d\boldsymbol{\sigma}^2} d\boldsymbol{\sigma} \right) \tag{49}$$

Remark: here, $\gamma = d\gamma$ as the return takes place in one step. Rearranging terms of (49):

$$\left[\mathbf{I} + d\gamma \mathbf{D}^{\mathbf{el}} \frac{d^2 q}{d\boldsymbol{\sigma}^2} \right] d\boldsymbol{\sigma} = \mathbf{D}^{\mathbf{el}} \left(d\boldsymbol{\varepsilon} - d\gamma \frac{dq}{d\boldsymbol{\sigma}} \right) \tag{50}$$

And multiplying (50) by the compliance matrix $\mathbf{C}^{\mathbf{el}} = \mathbf{D}^{\mathbf{el}-1}$ on the left:

$$\left[\mathbf{C}^{\mathbf{el}} + d\gamma \frac{d^2 q}{d\boldsymbol{\sigma}^2} \right] d\boldsymbol{\sigma} = d\boldsymbol{\varepsilon} - d\gamma \frac{dq}{d\boldsymbol{\sigma}}$$

$$\uparrow \quad \mathbf{D}^{\mathbf{el}*-1} \tag{51}$$

Finally, multiplying (51) by matrix $\mathbf{D}^{\mathbf{el}*}$ on the left, one gets:

$$d\boldsymbol{\sigma} = \mathbf{D}^{\mathbf{el}*} \left(d\boldsymbol{\varepsilon} - d\gamma \frac{dq}{d\boldsymbol{\sigma}} \right) \tag{52}$$

And using the procedure described in the previous section for the tangent matrix $\mathbf{D}^{\mathbf{ep}}$, we get the consistent tangent operator $\mathbf{D}^{\mathbf{ep*}}$:

$$d\boldsymbol{\sigma} = \left[\mathbf{D}^{\mathbf{el}*} - \frac{\left(\mathbf{D}^{\mathbf{el}*} \frac{dq}{d\boldsymbol{\sigma}}\right)\left(\mathbf{D}^{\mathbf{el}*} \frac{df}{d\boldsymbol{\sigma}}\right)^T}{\frac{df}{d\boldsymbol{\sigma}}^T \mathbf{D}^{\mathbf{el}*} \frac{dq}{d\boldsymbol{\sigma}}} \right] d\boldsymbol{\varepsilon} \tag{53}$$

2.1.2 Forming the right-hand side

The computation of the right-hand side (or internal forces) proceeds as follows:

$$\mathbf{F}^{\mathbf{int}} = \mathbf{N}\left(\mathbf{d}^{\mathbf{i}}_{\mathbf{n+1}}\right) = \overset{\#\,el.}{\underset{e=1}{\mathbf{A}}} \int_e \mathbf{B}^T \boldsymbol{\sigma}^{\mathbf{i}}_{\mathbf{n+1}} \, dV_e \tag{54}$$

where $\mathbf{N}$ was defined earlier and $\overset{\#\,el.}{\underset{e=1}{\mathbf{A}}}$ denotes an assembly of elemental contributions. The following method of class Element is computing internal forces:

```
//-----------------------------------------------------
FloatArray*  Element :: ComputeInternalForces
                                    (FloatArray* dElem)
//-----------------------------------------------------

{
Material         *mat;
GaussPoint         *gp;
FloatMatrix      *b;
FloatArray       *f;
double           dV;
int i;
mat   = this->giveMaterial();
f     = new FloatArray();
for (i=0; i<numberOfGaussPoints; i++) {
        gp    = gaussPointArray[i];
        b     = this->ComputeBmatrixAt(gp);
        mat -> ComputeStress(dElem,this,gp);
        dV    = this->computeVolumeAround(gp);
        f->plusProduct(b,gp->giveStressVector(),dV);
        delete b;
}
delete dElem;
return f;
}
```

This method has the same structure as the one described in section 2.1.1, forming the stiffness matrix.

The loop on Gauss points covers the whole element volume, and the method ComputeStress(dElem, this, gp) of class Material stores the stress state computed at the Gauss point.

2.1.2.1 Elastic case

The following method of class ElasticMaterial computes the stress state at the given Gauss point for the given element:

```
//---------------------------------------------------------
void  ElasticMaterial :: ComputeStress
        (FloatArray* dxacc, Element* elem, GaussPoint* gp)
//---------------------------------------------------------

{
FloatArray *sigma, *deltaEpsilon;
deltaEpsilon = elem->computeStrainIncrement(gp,dxacc);
sigma   = elem->giveConstitutiveMatrix()
                                ->Times(deltaEpsilon);
sigma->add(gp->givePreviousStressVector());
delete deltaEpsilon;
gp->letStressVectorBe(sigma);
}
```

2.1.2.2 Elasto-plastic case

2.1.2.2.1 General form of the stress computation algorithm

The following general stress-return algorithm can be formulated.

Problem: given $\boldsymbol{\sigma}_{\mathbf{n}}$ and $d\boldsymbol{\varepsilon}_{\mathbf{n+1}} = \mathbf{B}\, d\mathbf{d}^{\mathbf{acc}}_{\mathbf{n+1}}$, find $\boldsymbol{\sigma}_{\mathbf{n+1}}$ (for iteration i and step n+1)

First, compute a trial stress state:

$$\boldsymbol{\sigma}^{\mathbf{tr}}_{\mathbf{n+1}} = \boldsymbol{\sigma}_{\mathbf{n}} + d\boldsymbol{\sigma}^{\mathbf{tr}} = \boldsymbol{\sigma}_{\mathbf{n}} + \mathbf{D}^{\mathbf{el}} d\boldsymbol{\varepsilon}_{\mathbf{n+1}} \tag{55}$$

$$\mathbf{s}^{\mathbf{tr}}_{\mathbf{n+1}} = \boldsymbol{\sigma}^{\mathbf{tr}}_{\mathbf{n+1}} - p^{tr}_{n+1}\boldsymbol{\delta} \tag{56}$$

Check yielding for the trial stress state:

$$\text{if} \quad f\left(\boldsymbol{\sigma}^{\mathbf{tr}}_{\mathbf{n+1}}\right) \le 0 \qquad \boldsymbol{\sigma}_{\mathbf{n+1}} = \boldsymbol{\sigma}^{\mathbf{tr}}_{\mathbf{n+1}} \tag{57}$$

Else, impose global consistency via $f\left(\boldsymbol{\sigma}_{\mathbf{n+1}}\right) = 0$ and define $d\boldsymbol{\sigma}$ such that:

$$\boldsymbol{\sigma}_{\mathbf{n+1}} = \boldsymbol{\sigma}^{\mathbf{tr}}_{\mathbf{n+1}} + d\boldsymbol{\sigma}^{\mathbf{p}} \tag{58}$$

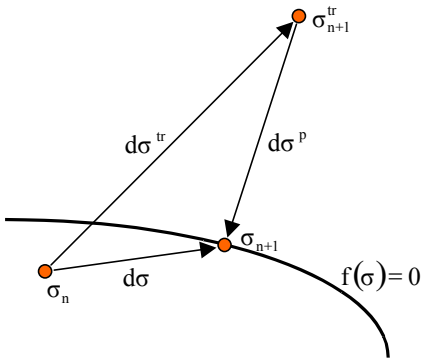

Figure 6: Stress increments

with $d\boldsymbol{\sigma}^{\mathbf{p}}$ derived as follows:

$$d\boldsymbol{\sigma} = \mathbf{D}^{\mathbf{el}}\left(d\boldsymbol{\varepsilon} - d\boldsymbol{\varepsilon}^{\mathbf{p}}\right) = d\boldsymbol{\sigma}^{\mathbf{tr}} + d\boldsymbol{\sigma}^{\mathbf{p}} \tag{59}$$

From consistency:

$$f\left(\boldsymbol{\sigma}_{\mathbf{n+1}}\right) = f\left(\boldsymbol{\sigma}^{\mathbf{tr}}_{\mathbf{n+1}} + d\boldsymbol{\sigma}^{\mathbf{p}}\right) \cong f\left(\boldsymbol{\sigma}^{\mathbf{tr}}_{\mathbf{n+1}}\right) + \left.\frac{df}{d\boldsymbol{\sigma}}\right|_{\boldsymbol{\sigma}^{\mathbf{tr}}_{\mathbf{n+1}}} d\boldsymbol{\sigma}^{\mathbf{p}} = 0 \tag{60}$$

Using the definition of $d\boldsymbol{\sigma}^{\mathbf{p}}$ given by (59) and the flow rule:

$$d\boldsymbol{\sigma}^{\mathbf{p}} = -\mathbf{D}^{\mathbf{el}} d\gamma\, \mathbf{r} \tag{61}$$

Introducing (61) into (60), we get (in vector notations):

$$d\gamma = \frac{f\left(\boldsymbol{\sigma}^{\mathbf{tr}}_{\mathbf{n+1}}\right)}{\frac{df}{d\boldsymbol{\sigma}}^{T} \mathbf{D}^{\mathbf{el}}\, \mathbf{r}} \tag{62}$$

A geometric interpretation of the stress return for Von Misès plasticity is described next.

2.1.2.2.2 Radial return for elastic-perfectly plastic associated Von Misès plasticity (following [10])

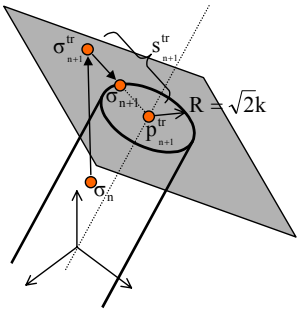

Figure 7: Radial return for Von Misès plasticity: three dimensional stress space view

In the case of associated Von Misès plasticity, the return happens always in the deviatoric plane (corresponding to incompressible flow). Geometrical considerations in this plane lead to:

$$\boldsymbol{\sigma}_{\mathbf{n+1}} = p^{tr}_{n+1}\boldsymbol{\delta} + \frac{R}{\left\|\mathbf{s}^{\mathbf{tr}}_{\mathbf{n+1}}\right\|}\mathbf{s}^{\mathbf{tr}}_{\mathbf{n+1}} \tag{63}$$

$$\mathbf{s} = \boldsymbol{\sigma} - p\boldsymbol{\delta} = \boldsymbol{\sigma} - \frac{\sigma_{kk}}{3}\boldsymbol{\delta} \tag{64}$$

Where $\mathbf{s}$ is the deviatoric stress tensor, p is the mean or hydrostatic stress tensor and $\boldsymbol{\delta}$ is the unit tensor. Finally, the norm of the deviatoric trial stress is given by:

$$\left\|\mathbf{s}_{n+1}^{tr}\right\| = \sqrt{\mathbf{s}_{n+1}^{tr\ T}\mathbf{A}\mathbf{s}_{n+1}^{tr}} \ ; \ \mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{65}$$

where the matrix $\mathbf{A}$ is introduced to maintain compatibility between the vectorial and tensorial notations.

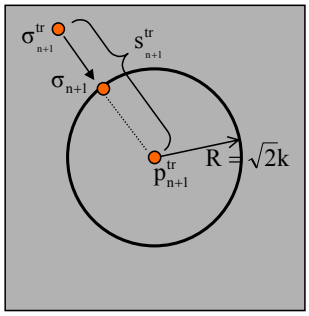

Figure 8: Radial return for Von Misès plasticity: deviatoric plane view

The following method of class VonMisesMaterial computes the stress state, enforcing consistency if needed and stores it at the Gauss point level:

```
//-----------------------------------------------------------
void  VonMisesMaterial :: ComputeStress
        (FloatArray* dxacc, Element* elem, GaussPoint* gp)
//-----------------------------------------------------------

{
FloatArray *sigmaTrial, *deltaEpsilon;
FloatMatrix *Del;
Del = elem->giveConstitutiveMatrix()->GiveCopy();
deltaEpsilon = elem->computeStrainIncrement(gp,dxacc);
sigmaTrial = Del->Times(deltaEpsilon);
sigmaTrial->add(gp->givePreviousStressVector());
double fSigTr = this->computeYieldFunctionFor(sigmaTrial);
if (fSigTr > 0) {
   double deltaGamma;
   FloatArray *dFDSigma = this->computeDFDSigma
                                          (sigmaTrial);
   deltaGamma = fSigTr / (dFDSigma->transposedTimes
                              (Del->Times(dFDSigma)));
  sigmaTrial->minus((Del->Times(dFDSigma))
                                 ->times(deltaGamma));
   gp->isPlastic();
   gp->setDeltaGamma(deltaGamma);
   delete dFDSigma;
}
gp->letStressVectorBe(sigmaTrial);
delete Del; delete deltaEpsilon;
}
```

2.1.3 Introducing hardening

2.1.3.1 Consistency condition

We introduce a tensorial function $\mathbf{h}(\boldsymbol{\sigma},\mathbf{q})$ which defines the direction of the hardening parameters increments, by analogy with $\mathbf{r}(\boldsymbol{\sigma},\mathbf{q})$ giving the direction of the plastic strain increment:

$$d\mathbf{q} = d\gamma \cdot \mathbf{h}(\boldsymbol{\sigma},\mathbf{q}) \tag{66}$$

$$d\boldsymbol{\varepsilon}^{\mathbf{p}} = d\gamma \cdot \mathbf{r}(\boldsymbol{\sigma},\mathbf{q}) \tag{67}$$

Introducing (66)-(67) into the consistency condition (28), and using the constitutive equation (29), we finally get:

$$d\gamma = \frac{\frac{\partial f}{\partial \boldsymbol{\sigma}}^T \mathbf{D}^{\mathbf{el}} d\boldsymbol{\varepsilon}}{\frac{\partial f}{\partial \boldsymbol{\sigma}}^T \mathbf{D}^{\mathbf{el}}\mathbf{r} - \frac{\partial f}{\partial \mathbf{q}}^T \mathbf{h}} \tag{68}$$

2.1.3.2 A linear combination of isotropic and kinematic strain hardening

Following section 1.2.3, for the case of mixed hardening we write:

$$d\mathbf{q} = \begin{bmatrix} dR \\ d\boldsymbol{\alpha} \end{bmatrix} = d\gamma \cdot \mathbf{h}(\boldsymbol{\sigma},\mathbf{q}) = d\gamma \begin{bmatrix} \frac{2}{3}\beta H' \\ \frac{2}{3}(1-\beta)H'\mathbf{r} \end{bmatrix} \tag{69}$$

with $H' \geq 0$, and $H' = 0$ implying perfect plasticity. Parameter β determines the proportion of isotropic and kinematic hardening or softening.

2.1.3.3 The elasto-plastic tangent constitutive matrix for Von Misès plasticity with hardening

The partial derivatives of $f(\boldsymbol{\sigma},\boldsymbol{\alpha},R)$ write:

$$\frac{\partial f}{\partial \boldsymbol{\sigma}} = \frac{\boldsymbol{\xi}}{\|\boldsymbol{\xi}\|} = \mathbf{n}; \frac{\partial f}{\partial \boldsymbol{\alpha}} = -\frac{\boldsymbol{\xi}}{\|\boldsymbol{\xi}\|} = -\mathbf{n}; \frac{\partial f}{\partial R} = -1 \tag{70), (71), (72}$$

Rewriting the consistency condition:

$$\frac{\partial f}{\partial \boldsymbol{\sigma}}^T \mathbf{D}^{\mathbf{el}}(d\boldsymbol{\varepsilon} - d\gamma \cdot \mathbf{r}) + \frac{\partial f}{\partial \mathbf{q}}^T d\gamma \cdot \mathbf{h} = 0 \tag{73}$$

and using (68) and (69), the elasto-plastic tangent operator can be written:

$$\mathbf{D}^{\mathbf{ep}} = \mathbf{D}^{\mathbf{el}} - \frac{(\mathbf{D}^{\mathbf{el}}\mathbf{r})(\mathbf{D}^{\mathbf{el}}\mathbf{n})^T}{\mathbf{n}^T\mathbf{D}^{\mathbf{el}}\mathbf{r} + \frac{2}{3}H'} \tag{74}$$

2.1.3.4 Radial return for Von Misès plasticity with hardening

Similarly to the perfectly plastic case (see section 2.1.2.2.1), the following stress-return algorithm can be written.

Given $\boldsymbol{\sigma}_{\mathbf{n}}$, $\boldsymbol{\alpha}_{\mathbf{n}}$, R_n and $d\boldsymbol{\varepsilon}_{\mathbf{n+1}}$, find $\boldsymbol{\sigma}_{\mathbf{n+1}}$, $\boldsymbol{\alpha}_{\mathbf{n+1}}$ and R_{n+1}

First, compute the trial stress state as:

$$\boldsymbol{\sigma}_{\mathbf{n+1}}^{\mathbf{tr}} = \boldsymbol{\sigma}_{\mathbf{n}} + d\boldsymbol{\sigma}^{\mathbf{tr}} = \boldsymbol{\sigma}_{\mathbf{n}} + \mathbf{D}^{\mathbf{el}} d\boldsymbol{\varepsilon}_{\mathbf{n+1}} \tag{75}$$

$$\boldsymbol{\xi}_{\mathbf{n+1}}^{\mathbf{tr}} = \boldsymbol{\sigma}_{\mathbf{n+1}}^{\mathbf{tr}} - \boldsymbol{\alpha}_{\mathbf{n}} \tag{76}$$

Check yielding for the trial stress state:

$$\text{if } f\left(\boldsymbol{\sigma}_{\mathbf{n+1}}^{\mathbf{tr}}\right)\leq 0 \quad \boldsymbol{\sigma}_{\mathbf{n+1}}=\boldsymbol{\sigma}_{\mathbf{n+1}}^{\mathbf{tr}},\boldsymbol{\alpha}_{\mathbf{n+1}}=\boldsymbol{\alpha}_{\mathbf{n}},R_{n+1}=R_n \qquad (77)$$

Else, impose global consistency via:

$$f\left(\boldsymbol{\sigma}_{\mathbf{n+1}},\boldsymbol{\alpha}_{\mathbf{n+1}},R_{n+1}\right)=0 \qquad (78)$$

with:

$$\boldsymbol{\sigma}_{\mathbf{n+1}}=\boldsymbol{\sigma}_{\mathbf{n+1}}^{\mathbf{tr}}+d\boldsymbol{\sigma}^{\mathbf{p}} \qquad (79)$$

$$\boldsymbol{\alpha}_{\mathbf{n+1}}=\boldsymbol{\alpha}_{\mathbf{n+1}}^{\mathbf{tr}}+d\boldsymbol{\alpha} \qquad (80)$$

$$R_{n+1}=R_n+dR \qquad (81)$$

Introducing (69)-(72) into (78)-(81), we get for $d\gamma$:

$$d\gamma=\frac{f\left(\boldsymbol{\sigma}_{\mathbf{n+1}}^{\mathbf{tr}},\boldsymbol{\alpha}_{\mathbf{n+1}},R_{n+1}\right)}{\mathbf{n}^T\mathbf{D}^{\mathbf{el}}\mathbf{r}+\frac{2}{3}H'} \qquad (82)$$

2.2 Global level

2.2.1 Assembly and solution procedure (class `NLSolver` and its subclasses)

The assembly of the different elements and nodes of the problem is managed by an instance of class Domain which embodies the problem to be solved. The domain solves the problem with the help of the nonlinear solver.

2.2.1.1 Problem algorithm

The global algorithm for solving a nonlinear problem is illustrated in figure 9.

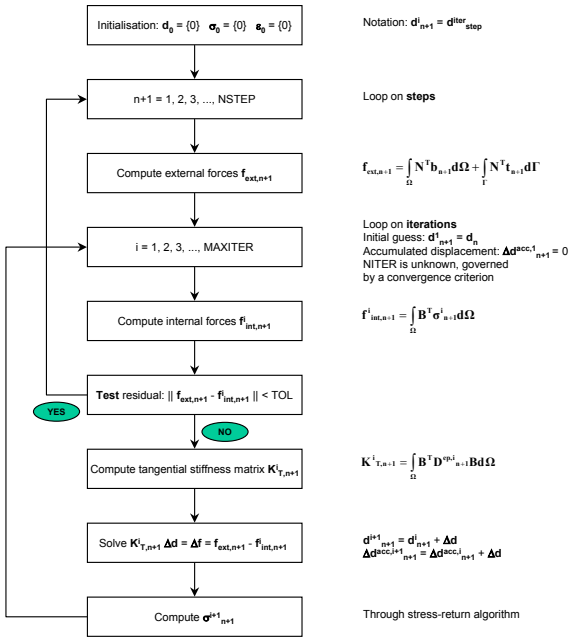

Figure 9: Global nonlinear algorithm

The Solve() method in class NLSolver is taken from [11]. The basic principle of this method is that the solver is an algebraic device whose only task is to solve an algebraic equation of type g(**x**) = 0. For that, the nonlinear solver needs an initial guess, a left-hand side (the Jacobian) and a right-hand side (the residual). These ingredients are held by the domain and returned to the solver whenever he needs it. The method Solve() takes the following form:

```
//------------------------------
FloatArray* NLSolver :: Solve ()
//------------------------------

{
Skyline      *jacobian;
FloatArray   *x,*y,*dx, *dxacc;
double       initialNorm, norm, tolerance, maxIterations;
const double PRECISION=1.e-9;
int          i,hasConverged=0;
tolerance = this->give('t');
maxIterations = this->give('n');
x = this->domain->GiveInitialGuess();
dxacc = new FloatArray(this -> domain
                          -> giveNumberOfFreeDofs());
for (i=1; i<=maxIterations; i++) {
        this->currentIteration = i;
        y = this->domain->ComputeRHSAt(dxacc)->minus();
        jacobian = this->domain->ComputeJacobian();
        this->linearSystem->setLHSTo(jacobian);
        this->linearSystem->setRHSTo(y);
        norm = y->giveNorm();
        if (i == 1) initialNorm = norm;
        if (initialNorm = 0.) initialNorm = 1.;
        if (norm/initialNorm<tolerance||norm<PRECISION) {
                hasConverged = 1;
                linearSystem->updateYourself();
                break;
        }
        if (norm/initialNorm > 10) {
                hasConverged = 0;
                linearSystem->updateYourself();
                break;
        }
        this->linearSystem->solveYourself();
        dx = this->linearSystem->giveSolutionArray();
        x->add(dx);
        dxacc->add(dx);
        linearSystem->updateYourself();
}
this->numberOfIterations = i;
this->convergenceStatus = hasConverged;
delete dxacc; delete jacobian; delete y;
return x;
}
```

2.2.2 How does the code work?

Figure 10 illustrates the interactions between the main objects that compose the application.

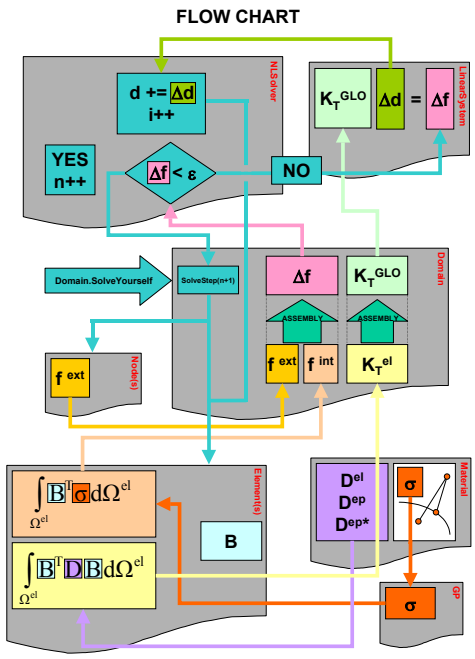

Figure 10: Applicaton flow chart

3 The class hierarchy

3.1 Existing code description

The class hierarchy of a finite element code that handles J_2-Plasticity is reviewed in this section. A brief description of each class is given. Particular attention is put on the most important classes (Element, GaussPoint, Material, Domain, NLSolver and their subclasses). The following notation rules are used in the class hierarchy (see Figure 11):

- classes are ordered alphabetically
- subclasses are indicated under their superclass with a right indent
- the classes that had to be added to the initial linear FEM_Object package [1] in order to handle nonlinearity are written in **bold** font

```
Dictionary
Dof
Domain
FEMComponent
        Element
                PlaneStrain
                        Quad_U
                                Quad_U_BBar
                                Triangle_U_Degen
                Truss2D
        Load
                BodyLoad
                        DeadWeight
                BoundaryCondition
                InitialCondition
                NodalLoad
        LoadTimeFunction
                ConstantFunction
                PeakFunction
                PiecewiseLinFunction
        Material
                ElasticMaterial
                VonMisesMaterial
                        VonMisesMaterial_H
        NLSolver
                ConstantStiffness
                ModNewtonRaphson
                NewtonRaphson
        Node
        TimeIntegrationScheme
                Newmark
                Static
        TimeStep
FileReader
FloatArray
        Column
GaussPoint
IntArray
LHS
        Skyline
LinearSystem
List
MathUtil
Matrix
        FloatMatrix
                DiagonalMatrix
        PolynomialMatrix
Pair
Polynomial
        PolynomialXY
```

Figure 11: The class hierarchy

- Class Dictionary

A dictionary is a collection with entries which have both a name and a value (class Pair).

Dictionaries are typically used by degrees of freedom (class Dof) for storing their unknowns (like ‘d’) and by materials for storing their properties (like ‘E’ or ‘ν’). The main task of a dictionary is to store pairs and return the pair’s value corresponding to the pair’s key.

- Class Dof

A degree of freedom is an attribute of a node. Its role is to relieve the node from the following tasks:
- managing the kinematic unknowns (like ‘d’)
- equation numbering
- checking the existence of an initial or boundary condition if any, and storing its number

- Class Domain

The domain can be considered as the “main” object that contains all the problems’ components; there is a single instance of Domain for each problem, and its principal tasks are:

- receiving the messages from the user and initiating the corresponding operations
- storing the components of the mesh: the list of nodes, elements, materials, loads, …
- storing the non-linear solver type (class NLSolver) and the time-integration scheme
- providing objects with access to the data file

Class Domain Inherits from : -		
Tasks	**Attributes**	**Methods**
1) creation	-	*Domain()* instanciateYourself()
2) management of the problem's components	elementList nodeList materialList loadList loadTimeFunctionList nlSolver timeIntegrationScheme numberOfElements numberOfNodes numberOfFreeDofs	giveElement(i) giveNode(i) giveMaterial(i) giveLoad(i) giveLoadTimeFunction(i) giveNLSolver() giveTimeIntegrationScheme() giveNumberOfElements() giveNumberOfNodes() giveNumberOfFreeDofs()
3) problem solving	unknownArray	giveUnknownArray() solveYourself() formTheSystemAt(aStep) solveYourselfAt(aStep) terminate(aStep)
4) interactions with the nonlinear solver	-	giveInitialGuess() givePastUnknownArray() computeRHSAt(aΔd) computeInternalForces(aΔd) computeLoadVectorAt(aStep) computeJacobian() computeTangentStiffnessMatrix()
5) input / output	dataFileName inputStream outputStream	giveDataFileName() giveInputStream() giveOutputStream() readNumberOf(aComponent)

Table 1: Class **Domain** description

▪ Class FEMComponent

This class, which is the superclass of classes Element, Load, LoadTimeFunction, Material, NLSolver, Node, TimeIntegrationScheme and TimeStep, regroups the attributes and methods which are common to all its subclasses (mainly access to the domain or to the input file).

▪ Class Element

Class Element regroups the attributes and the methods which are common to every element (which are instances of classes Quad_U, Quad_U_BBar or Truss2D). Its main tasks are:
- calculating its mass matrix (for dynamics), its stiffness matrix and its load vector
- giving its contributions to the left-hand side and the right-hand side of the system (through the assembly operation performed by class Domain and described in section 2.2.1)
- reading, storing and returning its data

Class Element Inherits from : FEMComponent		
Inherited Tasks	**Inherited Attributes**	**Inherited Methods**
creation, access to data	number domain	giveNumber() …
Tasks	**Attributes**	**Methods**
1) creation	-	*Element(aDomain, aNumber)* typed() ofType(anElementType) instanciateYourself()
2) attributes identification		
a) nodes	nodeArray numberOfNodes	giveNode(i)
b) material	material	giveMaterial()
c) loads	bodyLoadArray	giveBodyLoadArray()
d) Gauss points	gaussPointArray numberOfGaussPoints	-
3) computation & assembly		
a) stiffness matrix	stiffnessMatrix constitutiveMatrix	giveStiffnessMatrix() giveConstitutiveMatrix() computeTangentStiffnessMatrix() computeLHSAt(aStep) computeStaticLHSAt(aStep) computeNewmarkLHSAt(aStep)
b) mass matrix	massMatrix	giveMassMatrix()
c) load vector	-	computeLoadVectorAt(aStep) computeBcLoadVectorAt(aStep) computeVectorOfPrescribed(aStep) computeRHSAt(aStep) computeStaticRHSAt(aStep) computeNewmarkRHSAt(aStep)
d) internal forces	-	computeInternalForces(aΔd) computeStrainIncrement(aGP, aΔd)
e) assembly	locationArray	assembleYourselfAt(aStep) assembleLHSAt(aStep) assembleRHSAt(aStep) giveLocationArray()
4) output	-	printOutputAt(aStep, aFile)
5) internal handling	-	computeNumberOfDofs() updateYourself() giveClassName() printYourself()

Table 2: Class **Element** description

▪ Class PlaneStrain

This abstract class is the superclass of Quad_U. Its purpose is to give a generic superclass for other plane strain elements which would be later added in this environment (like the triangle for instance).

Class PlaneStrain Inherits from : Element, FEMComponent		
Inherited Tasks	**Inherited Attributes**	**Inherited Methods**
creation, access to data, computation & assembly, ...	number domain ...	giveNumber() ...
Tasks	**Attributes**	**Methods**
1) creation	-	*PlaneStrain(aDomain, aNumber)*

Table 3: Class **PlaneStrain** description

▪ Class **Quad_U**

This class implements a quadrilateral element. It inherits methods from its superclasses (PlaneStrain and Element) and adds its own behavior:

- calculating matrices **N** (shape functions), **B** (strains) and **D** (elastic constitutive matrix)
- numerical integration: calculating the position and the weight of the Gauss points, calculating the jacobian matrix

Class Quad_U Inherits from : PlaneStrain, Element, FEMComponent		
Inherited Tasks	**Inherited Attributes**	**Inherited Methods**
creation, access to data, computation & assembly, ...	number domain ...	giveNumber() ...
Tasks	**Attributes**	**Methods**
1) creation	-	*Quad_U(aDomain, aNumber)*
2) computation	-	computeNMatrixAt(aGP) computeBMatrixAt(aGP) computeCompactBMatrixAt(aXsiEtaPoint) computeConstitutiveMatrix()
3) numerical integration	jacobianMatrix	giveJacobianMatrix() computeGaussPoints() computeVolumeAround(aGP)

Table 4: Class **Quad_U** description

▪ Class **Quad_U_BBar**

The only specific task to this subclass of Quad_U is to compute its **B** matrix in a different way in order to overcome locking due to incompressiblility: a method returning the so-called **B-Bar** matrix is implemented in this class.

Class Quad_U_BBar Inherits from : Quad_U, PlaneStrain, Element, FEMComponent		
Inherited Tasks	**Inherited Attributes**	**Inherited Methods**
creation, access to data, computation & assembly, numerical integration ...	number domain ...	giveNumber() ...
Tasks	**Attributes**	**Methods**
1) creation	-	*Quad_U_BBar(aDomain, aNumber)*
2) B-bar matrix handling	-	computeBMatrixAt(aGP) giveBBarMatrix() computeMeanBBar() computeBBarAtCenter()

Table 5: Class **Quad_U_BBar** description

▪ Class **Triangle_U_Degen**

This subclass of Quad_U implements a linear triangular element obtained by degeneration of the bilinear quad (coalescing the nodes 3 and 4 of the quad, see [9] for details). This element will fail for incompressible tests.

▪ Class Truss2D

This class implements a two-node planar truss element. It defines its own methods for the calculation of matrices **N**, **B** and **D** and manages also its Gauss points. Additionally, it has the following tasks:

- characterizing its geometry (length and pitch)
- rotating its contributions to the system from its local coordinate frame to the global coordinate frame

▪ Class Load

This superclass implements the various actions applied on elements, nodes and degrees of freedom. Its subclasses (BodyLoad, DeadWeight, BoundaryCondition, InitialCondition and NodalLoad) are described next.

▪ Class BodyLoad

Body load is self explanatory.

▪ Class DeadWeight

This load, which is a subclass of class BodyLoad, implements a gravity-like body force. It is usually associated with every element of the mesh.

▪ Class BoundaryCondition

A boundary condition is a constraint imposed on degrees of freedom. It defines the prescribed values of the unknown and is the attribute of one or more degrees of freedom (class Dof).

▪ Class InitialCondition

An initial condition defines the initial value of an unknown at the start of the analysis. This concept is used for initial-boundary-value-problems.

▪ Class NodalLoad

A nodal load is a concentrated load which acts directly on the node. It is the attribute of one or more nodes. Its main task is to return the value of its components at a given time step.

▪ Class LoadTimeFunction

This superclass implements the functions that describe the evolution in time of a load. It is the attribute of one or more loads. Its task is to return its value at a given time step.

Its subclasses (ConstantFunction, PeakFunction, and PiecewiseLinFunction) are described next.

▪ Class ConstantFunction

This class implements load functions which are constant in time.

▪ Class PeakFunction

This class implements a load function whose value is zero everywhere, except in one point.

▪ Class PiecewiseLinFunction

This class implements a piecewise linear function.

▪ Class Material

This superclass was created in order to regroup common tasks for its subclasses.

Usually, a material is an attribute of many elements of the mesh. The constitutive information is stored in this class.

▪ Class **ElasticMaterial**

This class implements an elastic material. Its main task is to return its properties, e.g. Young modulus, Poisson ratio, …

Class **Material** Inherits from : FEMComponent		
Inherited Tasks	**Inherited Attributes**	**Inherited Methods**
creation, access to data	number domain	giveNumber() …
Tasks	**Attributes**	**Methods**
1) creation	-	*Material(aDomain, aNumber)* typed() ofType(aMaterialType)
2) attributes identification	propertyDictionnary	give(aProperty)
3) internal handling	-	giveClassName() giveKeyword() printYourself()

Table 6: Class **Material** description

▪ Class **VonMisesMaterial**

This class implements a plastic material of type Von Misès. This means that, apart from returning its properties like the ElasticMaterial class (including its Von Misès parameter k), it also performs two important tasks which have been described in section 2.1:

- it computes the stress state of its elements (or, more precisely, of its elements' Gauss points) through the stress return algorithm (see section 2.1.2.2)
- it computes the constitutive matrix which can either be elastic (elastic material or constant stiffness algorithm), tangent or tangent-consistent for an improved convergence (see section 2.1.1.3)

Class VonMisesMaterial Inherits from : Material, FEMComponent		
Inherited Tasks	**Inherited Attributes**	**Inherited Methods**
creation, access to data, attributes identification, ...	number domain propertyDictionary	giveNumber() ...
Tasks	**Attributes**	**Methods**
1) creation	-	VonMises*Material(aDomain, aNumber)* instanciateYourself()
2) computation	-	computeStress(aGP, anElem, aΔd) computeConstitutiveMatrix(aGP, anElem) computeDFDSigma(aStressState) computeYieldFunctionFor(aStressState) computeStressLevelFor(aStressState)
3) internal handling	-	giveClassName() printYourself()

Table 7: Class **VonMisesMaterial** description

▪ Class **VonMisesMaterial_H**

This subclass of VonMisesMaterial implements a plastic material of type Von Misès with a linear combination of kinematic and isotropic hardening (see sections 1.2.3 and 2.1.3 for more details).

Class VonMisesMaterial_H Inherits from : VonMisesMaterial, Material, FEMComponent		
Inherited Tasks	**Inherited Attributes**	**Inherited Methods**
creation, access to data, attributes identification, ...	number domain propertyDictionary	giveNumber() ...
Tasks	**Attributes**	**Methods**
1) creation	-	VonMises*Material_H(aDomain, aNumber)* instanciateYourself()
2) computation	-	computeStress(aGP, anElem, aΔd) computeConstitutiveMatrix(aGP, anElem) computeKsi(aStressState, anα) computeDFDSigma(aStressState, anα) computeYieldFunctionFor(aStressSt., anα) computeStressLevelFor(aStressState)
3) internal handling	-	giveClassName() printYourself()

Table 8: Class **VonMisesMaterial_H** description

▪ Class **NLSolver**

This class, which is the superclass of classes ConstantStiffness, ModNewtonRaphson and NewtonRaphson, implements a nonlinear solver (see section 2.2.1). Its main task is to solve the nonlinear problem at each iteration and each step. The convergence (or divergence) is also checked in this class. The type of left-hand side depends on the type of algorithm which is defined by the three following classes.

Class NLSolver Inherits from : FEMComponent		
Inherited Tasks	**Inherited Attributes**	**Inherited Methods**
creation, access to data	number domain	giveNumber() …
Tasks	**Attributes**	**Methods**
1) creation	-	*NLSolver(aDomain, aNumber)* typed() ofType(aNLSolverType)
2) attributes identification	propertyDictionnary	give(aProperty)
3) computation	linearSystem maxIterations numberOfIterations currentIteration tolerance convergenceStatus consistentDep	solve() giveLinearSystem() giveNumberOfIterations() giveCurrentIteration() giveConvergenceStatus() giveConsistentDep()
4) internal handling	-	updateYourself() giveClassName() giveKeyword() printYourself()

Table 9: Class **NLSolver** description

▪ Class **ConstantStiffness**

In this subclass of class NLSolver, the initial stiffness is kept through all the iterative process in order to form the left-hand side.

▪ Class **ModNewtonRaphson**

In this subclass of class NLSolver, the stiffness is updated each n_s steps and n_i iterations in order to form the left-hand side.

▪ Class **NewtonRaphson**

In this subclass of class NLSolver, the stiffness is updated at each iteration in order to form the left-hand side.

Class NewtonRaphson Inherits from : NLSolver, FEMComponent		
Inherited Tasks	**Inherited Attributes**	**Inherited Methods**
creation, access to data, computation, …	number domain linearSystem ...	giveNumber() …
Tasks	**Attributes**	**Methods**
1) creation	-	*NewtonRaphson(aDomain, aNumber)* instanciateYourself()
2) internal handling	-	giveClassName() printYourself()

Table 10: Class **NewtonRaphson** description

▪ Class Node

A node is the attribute of one or more elements. It has the following four tasks:

- returning its coordinates
- managing (creating and storing) its degrees of freedom (class Dof)
- computing and assembling its nodal load vector (class NodalLoad)
- updating its attributes at the end of each step

Class Node **Inherits from : FEMComponent**		
Inherited Tasks	**Inherited Attributes**	**Inherited Methods**
creation, access to data	number domain	giveNumber() …
Tasks	**Attributes**	**Methods**
1) creation	-	*Node(aDomain, aNumber)* instanciateYourself()
2) positioning in space	coordinates	getCoordinates() giveCoordinate(i)
3) management of the degrees of freedom	dofArray numberOfDofs	giveDof(i) giveNumberOfDofs()
4) management of the nodal load vector:		
a) computation	loadArray	computeLoadVectorAt(aStep) giveLoadArray()
b) assembly	locationArray	assembleYourLoadsAt(aStep) giveLocationArray()
5) output	-	printOutputAt(aStep, aFile) printBinaryResults(aStep, aFile)
6) internal handling	-	updateYourself() giveClassName() printYourself()

Table 11: Class **Node** description

▪ Class TimeIntegrationScheme

This class (and its subclasses Newmark and Static) define the time history of the problem. Its tasks are:

- managing the time history of the problem (i.e. the time steps (class TimeStep))
- returning its coefficients (for instance β or γ)

▪ Class Newmark

This subclass of TimeIntegrationScheme implements a predictor-corrector method for dynamic analysis.

▪ Class Static

This subclass of TimeIntegrationScheme implements a scheme supporting the static analysis of a structure subjected to various loading cases.

▪ Class TimeStep

This class implements a time step in the time history of the problem.

A time step is an attribute of a time integration scheme.

The tasks of a time step is to return its current time t and time increment Δt.

▪ Class FileReader

A file reader is an attribute of the domain. It provides non-sequential access to the data file.

▪ Class FloatArray

This class implements an array of double-precision decimal numbers. Its tasks are:

- storing and returning a coefficient, including index-overflow checking
- performing standard operations (addition, scalar multiplication, rotation, etc…)
- expanding its size
- assembling an elemental or nodal contribution, if the array is used as the right-hand side of the linear system

Stresses and strains are instances of the FloatArray class. This means that additional operations (for instance computing invariants) have been added.

▪ Class Column

A column is an attribute of a skyline matrix. It stores the coefficients of a column. Its tasks are some among the ones defined in class FloatArray, although they are implemented differently.

▪ Class GaussPoint

A Gauss point is an attribute of an element. Its task is to regroup the data which are specific to the Gauss point: the coordinates and the weight of the point in numerical integration, the strains, the stresses. Nonlinear analysis induces some special tasks for the Gauss point. It has to store the stress and strain state at the current iteration, but also remember the last converged stress state. It also manages the amplitude of the stress return ($\Delta\gamma$), the state of the point (elastic or plastic), and the stress level.

Class GaussPoint **Inherits from : -**		
Tasks	**Attributes**	**Methods**
1) creation	number element coordinates weight	*GaussPoint(aNumber, anElement, x, y, w)* giveNumber() giveCoordinates() giveCoordinate(i) giveWeight()
2) stresses / strains handling	stressVector previousStressVector strainVector	giveStressVector() givePreviousStressVector() giveStrainVector() letStressVectorBe(aStressState) letPreviousStressVectorBe(aStressState) letStrainVectorBe(aStrainState)
3) stress return computation	deltaGamma plasticCode stressLevel	setDeltaGamma(aDeltaGamma) giveDeltaGamma() isPlastic() givePlasticCode() computeStressLevel() giveStressLevel()
4) output	-	printOutput(aFile) printBinaryResults(aFile)
5) internal handling	-	updateYourself()

Table 12: Class **GaussPoint** description

▪ Class IntArray

This class implements an array of integers.

▪ Class **LHS**

This generic superclass was created in order to account for different types of left-hand sides (i.e. skyline, GMRES, BFGS, …)

▪ Class Skyline

A skyline is a symmetric matrix stored in a variable-band form. A skyline is used as the left-hand side of a linear system. Its tasks are:

- setting up its own profile
- assembling to itself an elemental contribution (for instance a stiffness matrix)
- performing solving operations

▪ Class LinearSystem

The linear system is an attribute of class NLSolver. Its tasks:

- initializing its left-hand side, right-hand side and solution array, and returning them upon request
- solving itself

▪ Class List

A list is an array which coefficients are of type Element, Node, Load, … Its tasks are:

-storing, deleting or returning an element of the list
-expanding its own size, in order to accommodate more objects

Typically, the domain stores the nodes, the elements, the loads of the problem in lists.

▪ Class **MathUtil**

This class was created in order to store mathematical utilities.

▪ Class Matrix

This class is the superclass of different types of matrices (classes FloatMatrix, DiagonalMatrix and PolynomialMatrix). It implements basic operations such as index-overflow checking.

▪ Class FloatMatrix

This class implements a rectangular matrix which coefficients are double-precision decimal numbers. The tasks assigned to such matrices are:

- storing and returning a coefficient
- performing standard operations: addition, inversion, lumping

▪ Class DiagonalMatrix

This class implements a matrix with non-zero coefficients on the diagonal.

▪ Class PolynomialMatrix

This class implements a matrix which coefficients are polynomials. Typically, jacobian matrices of plane strain elements are polynomial matrices.

▪ Class Pair

A pair is a key/value association. Pairs are used as entries of class Dictionary.

▪ Class Polynomial

Polynomial are used as coefficients of polynomial matrices, for instance P(X,Y). The task of a polynomial is to return its value at a given point.

3.2 Object-oriented extendability

The addition of a new component in the code (say a new element, for instance a three-node linear triangle) is made naturally in the class hierarchy:

```
...
Element
        PlaneStrain
                Quad_U
                        Quad_U_BBar
                        Triangle_U_Degen
                Triangle_U
...
```

The new class inherits the behavior of its superclasses and only a minimal number of methods have to be rewritten. The same concept applies to other components, like algorithms, formulations or materials. For instance, Drucker-Prager or Mohr-Coulomb materials could be inserted in the hierarchy as subclasses of class Material and inherit most methods from their superclass:

```
...
Material
        ElasticMaterial
        VonMisesMaterial
                VonMisesMaterial_H
        MohrCoulombMaterial
        DruckerPragerMaterial
...
```

4 Examples

4.1 The footing problem

Problem and geometry

The problem of the bearing capacity of a superficial footing is described next. The geometry and characteristics of the problem are given in figure 12.

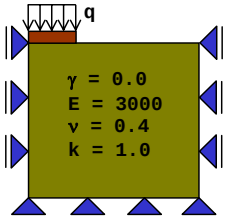

Figure 12: Geometry of the footing problem

The load q on the footing is increased until failure occurs.

Results

The solution converges at q = 6 kN/m and fails to converge at 7 kN/m, which, compared to the solution given by Terzaghi [12] q_u = 5 kN/m, is satisfactory for a crude mesh. Figure 13 illustrates the time history of the vertical displacement of a node at the interface between the footing and the soil. A clear divergence appears at time t = 7, illustrated by the failure mechanism (figure 14).

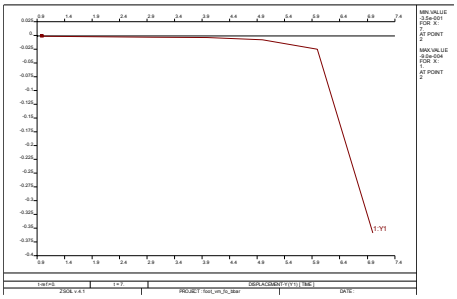

Figure 13: Time history

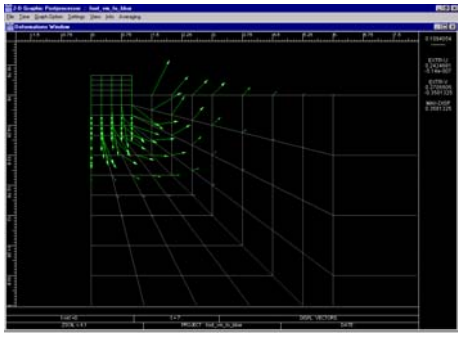

Figure 14: Failure mechanism

4.2 The thick cylinder test

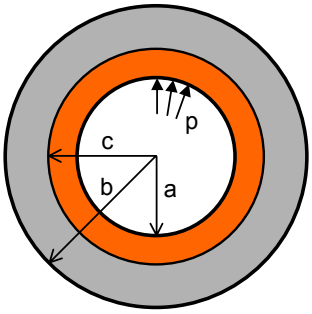

Figure 15: Geometry for the thick cylinder test

This experiment concerns a thick cylinder test, which has an analytical solution [13]. The internal and external radii of the cylinder are a = 1.0 and b = 2.0 m. Young's modulus E = 21000 kN/m^2 and Poisson's ratio ν = 0.49999. Von-Misès criterion is used with a yield stress σ_y = 24 kN/m^2 which corresponds to k = σ_y / √3 = 13.8564 kN/m^2. The internal pressure p varies between 8 kN/m^2 and 20 kN/m^2, this value corresponding to the total plastification of the cylinder and its failure. A 640 elements mesh (figure 16) has been used for a plane strain analysis.

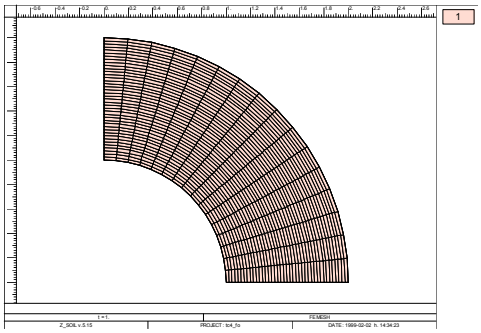

Figure 16: Finite element mesh

If we compare the evolution of the internal displacement in three cases: a) perfectly plastic material, b) H' = E / 3 = 7'000 kN/m^2, c) H' = 2E / 3 = 14'000 kN/m^2, the results obtained with the code are in good agreement with the theoretical solutions given in [12] for a). As expected (figure 17), the yield radius develops more slowly as H' increases.

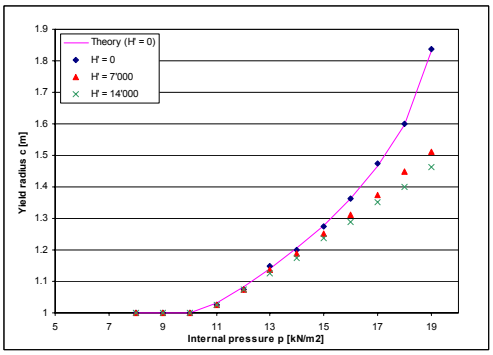

Figure 17: Evolution of the yield radius

5 Conclusions

An object-oriented finite element program for nonlinear structural and continuum analysis has been described in this paper using Von Misès plasticity as an illustration of constitutive theory. The usefulness of this object-oriented approach to solving nonlinear finite element problems has been demonstrated.

Students and engineers in practice will find here an optimal starting package for finite element programming in C++ which can be downloaded at:

http://www.zace.com/femobj_nl/femobj_nl.htm

Extensions to other plastic models or different finite element formulations can be introduced in the code with little effort because of the strong modularity supplied by the object-oriented approach. The interested reader will find in [14] a more detailed description of the code presented here.

Acknowledgements

We acknowledge the financial support of the Swiss National Science Foundation under grant n^o 2100-49404.96 and of the

"Fonds du 15ème congrès des Grands Barrages", for the first author.

References

[1] Y. Dubois-Pélerin, Th. Zimmermann - "Object-oriented finite element programming: Theory and C++ implementation for FEM_Object C++ 001", Elmepress International, 1993
[2] Ph. Menétrey, Th. Zimmermann - "Object-oriented nonlinear finite element analysis: application to J2 plasticity", Computers & Structures (49), pp. 767-777, 1993
[3] W. Chen - "Plasticity in reinforced concrete", McGraw-Hill, 1982
[4] T.J.R. Hughes - "Generalization of selective integration procedures to anisotropic and nonlinear media", International Journal for Numerical Methods in Engineering (15), pp. 1413-1418, 1980
[5] Th. Zimmermann, A. Truty, S. Commend - "A comparison of finite element enhancements for incompressible and dilatant behavior of geomaterials", CIMASI Conference Proceedings, 1998
[6] Th. Zimmermann, Y. Dubois-Pélerin, P. Bomme - "Object-oriented finite element programming: I Governing principles", Computer Methods in Applied Mechanics and Engineering (98), 1992
[7] Y. Dubois-Pélerin, Th. Zimmermann, P. Bomme - "Object-oriented finite element programming: II A prototype program in Smalltalk", Computer Methods in Applied Mechanics and Engineering (98), 1992
[8] Y. Dubois-Pélerin, Th. Zimmermann - "Object-oriented finite element programming: III An efficient implementation in C++", Computer Methods in Applied Mechanics and Engineering (108), 1993
[9] T.J.R. Hughes - "The finite element method", Prentice-Hall, 1987
[10] T.J.R. Hughes, T. Belytschko - "Nonlinear Finite Element Analysis", Course Notes, Paris, 1997
[11] Y. Dubois-Pélerin, P. Pegon - "Object-oriented programming in nonlinear finite element analysis", Computers & Structures (67), pp. 225-241, 1998
[12] K. Terzaghi - "Mécanique théorique des sols", Dunod, Paris, 1951
[13] R. Hill - "The mathematical theory of plasticity", Oxford University Press, 1950
[14] S. Commend - "An object-oriented approach to nonlinear finite element programming", LSC Internal Report 98/7, 1998
[15] D.R. Rehak, J.W. Baugh Jr. - "Alternative programming techniques for finite element program development", Proceedings IABSE Colloquium on Expert Systems in Civil Engineering, Bergamo, Italy, 1989
[16] B.W.R. Forde, R.B. Foschi, S.F. Steimer - "Object-oriented finite element analysis", Computers & Structures (34), pp. 355-374, 1990
[17] G.R. Miller - "An object-oriented approach to structural analysis and design"", Computers & Structures (40), pp. 75-82, 1991
[18] R.I. Mackie - "Object-oriented programming of the finite element method", International Journal for Numerical Methods in Engineering (35), pp. 425-436, 1992.