View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Open Archive Toulouse Archive Ouverte

An object-oriented programming of an explicit dynamics code: application to impact simulation

Olivier Pantalé*

LGP CMAO, ENIT, 47 Av d'Azereix, BP 1629, 65016 Tarbes Cedex, France

Abstract

During the last fifty years, the development of better numerical methods and more powerful computers has been a major enterprise for the scientific community. Recent advances in computational softwares have lead to the possibility of solving more physical and complex problems (coupled problems, nonlinearities, high strain and high strain rate problems, etc.). The development of object-oriented programming leads to better structured codes for the finite element method and facilitates the development, the maintainability and the expandability of such codes.

This paper presents an implementation in C++ of an explicit finite element program dedicated to the simulation of impacts. We first present a brief overview of the kinematics, the conservative and constitutive laws related to large deformation inelasticity. Then we present the design and the numerical implementation of some aspects developed with an emphasis on the object-oriented programming adopted. Finally, the efficiency and accuracy of the program are investigated through some benchmark tests.

Keywords: Nonlinear finite-element; Explicit integration; Large deformations; Plasticity; Impact; C++; Object-oriented programming

1. Introduction

After a long time of intensive developments, the finite element method has become a widely used tool for researchers and engineers. An accurate analysis of large deformation inelastic problems occurring in impact simulations is extremely important as a consequence of a high amount of plastic flow. This research field has been widely explored and a number of computational algorithms for the integration of constitutive relations have been developed for the analysis of large deformation problems [1,2].

In this paper an object-oriented (OO) implementation of an explicit finite element program called DynELA is presented. This FEM program is written in C++ [3]. The development of object-oriented programming (OOP) leads to better-structured codes for the finite element method and facilitates the development and maintainability [4,5]. A significant advantage of OOP concerns the modeling of complex physical systems such as deformation processing where the overall complex problem is partitioned in individual subproblems based on physical, mathematical or geometric reasoning. Therefore, the finite element concept leads naturally to an object representation.

2. Governing equations, discretization and numerical integration

The conservative laws and the constitutive equations for path-dependent material are formulated in an updated Lagrangian finite element method in large deformations. Both the geometrical and material nonlinearities are included in this setting. The finite element method (FEM) is used for the discretization of the conservative equations, and an explicit integration scheme is then adopted for time discretization of those equations. In the next paragraphs, we summarize some basic results concerning nonlinear mechanics relevant to our subsequent developments and refers for example to Hughes [6] or Simo and Hughes [7] for details concerning finite element method and the integration of constitutive laws.

2.1. Basic kinematics and constitutive equations

In a Lagrangian description, the mass, momentum and

* Tel.: +33-5-62-44-27-00; fax: +33-5-62-44-27-08.
E-mail address: oliver.pantale@enit.fr (O. Pantalé).

energy equations which govern the continuum are given by

$$\dot{\rho} + \rho \operatorname{div} \vec{v} = 0 \tag{1}$$

$$\rho\dot{\vec{v}} = \rho\vec{f} + \operatorname{div} \sigma \tag{2}$$

$$\rho\dot{e} = \sigma : \mathbf{D} - \operatorname{div} \vec{q} + \rho r \tag{3}$$

where ρ is the mass density, $(\dot{\cdot})$ the time derivative of ( ), $\vec{v}$ the material velocity, $\vec{f}$ the body force vector, σ the Cauchy stress tensor, $\mathbf{D}$ the spatial rate of deformation, e the specific internal energy, r the body heat generation rate and $\vec{q}$ is the heat flux vector. The symbol ':' denotes the contraction of a pair of repeated indices which appear in the same order, so $\mathbf{A} : \mathbf{B} = A_{ij}B_{ij}$. The matricial forms of Eqs. (1)–(3) are obtained, according to the finite element method, by subdividing the domain of interest Ω_x into a finite number of elements, leading to the following forms of the conservative equations:

$$\mathbf{M}^{\rho}\dot{\rho} + \mathbf{K}^{\rho}\rho = 0 \tag{4}$$

$$\mathbf{M}^{v}\dot{\vec{v}} + \mathbf{F}^{\text{int}} = \mathbf{F}^{\text{ext}} \tag{5}$$

$$\mathbf{M}^{e}\dot{e} + \mathbf{g} = \mathbf{r} \tag{6}$$

If we use the same form $\varphi^{(\,)}$ for the shape and test function (as usually done for an serendipity element), one may obtain the following expressions for the elementary matrices of Eqs. (4)–(6)

$$\mathbf{M}^{\rho} = \int_{\Omega_x} \varphi^{\rho^{\mathrm{T}}} \varphi^{\rho}\, \mathrm{d}\Omega_x, \qquad \mathbf{K}^{\rho} = \int_{\Omega_x} \varphi^{\rho^{\mathrm{T}}} \nabla v \varphi^{\rho}\, \mathrm{d}\Omega_x \tag{7}$$

$$\begin{aligned} \mathbf{M}^{v} &= \int_{\Omega_x} \rho\varphi^{v^{\mathrm{T}}} \varphi^{v}\, \mathrm{d}\Omega_x, \qquad \mathbf{F}^{\text{int}} = \int_{\Omega_x} \nabla\varphi^{v^{\mathrm{T}}} \sigma\, \mathrm{d}\Omega_x, \\ \mathbf{F}^{\text{ext}} &= \int_{\Omega_x} \rho\varphi^{v^{\mathrm{T}}} \vec{b}\, \mathrm{d}\Omega_x + \int_{\Gamma_x} \varphi^{v^{\mathrm{T}}} \vec{t}\, \mathrm{d}\Gamma_x \end{aligned} \tag{8}$$

$$\begin{aligned} \mathbf{M}^{e} &= \int_{\Omega_x} \varphi^{e^{\mathrm{T}}} \varphi^{e}\, \mathrm{d}\Omega_x, \qquad \mathbf{g} = \int_{\Omega_x} \nabla\varphi^{e^{\mathrm{T}}} \vec{q}\, \mathrm{d}\Omega_x, \\ \mathbf{r} &= \int_{\Omega_x} \varphi^{e^{\mathrm{T}}} (\sigma : \mathbf{D} + \rho r)\, \mathrm{d}\Omega_x - \int_{\Gamma_x} \varphi^{e^{\mathrm{T}}} \theta\, \mathrm{d}\Gamma_x \end{aligned} \tag{9}$$

where ∇ is the gradient operator, Γ_x is the surface of the domain Ω_x, $\mathbf{M}^{(\,)}$ are consistent mass matrices, $\mathbf{F}^{\text{ext}}$ is the external force vector and $\mathbf{F}^{\text{int}}$ is the internal force vector. As usually done, we associate the explicit integration scheme with the use of lumped mass matrices in calculations, therefore quantities $(\dot{\,})$ are directly obtained from Eqs. (4)–(6) without the need of any matrix inversion algorithm.

2.2. Constitutive law

This finite element code is dedicated to large strains simulations, therefore we must ensure the objectivity of all the terms appearing in the constitutive law. The symmetric part of the spatial velocity gradient $\mathbf{L}$, denoted by $\mathbf{D}$ is objective while its skew-symmetric part $\mathbf{W}$, called the spin tensor, is not objective. The incremental formulation of the constitutive law is given by $\dot{\sigma} = f(\mathbf{D}\cdots)$. Assuming that the Cauchy stress tensor σ is objective, its material time derivative $\dot{\sigma}$ is nonobjective, so one must introduce an objective rate notion which is a modified time derivative form of the Cauchy stress tensor σ as the Jaumann–Zaremba or the Green–Naghdi derivatives. One of the solutions to this problem consists of defining a new Cauchy stress rate in a rotating referential defined using a rotation tensor $\mathbf{w}$ with $\dot{w} = \omega\mathbf{w}$. Defining any quantity ( ) in this rotating referential as a corotational one denoted by $(\,)^{\mathrm{c}}$, one may obtain:

$$\sigma^{\mathrm{c}} = \mathbf{w}^{\mathrm{T}}\sigma\mathbf{w} \text{ and } \dot{\sigma}^{\mathrm{c}} = \mathbf{w}^{\mathrm{T}}\dot{\sigma}\mathbf{w} \tag{10}$$

In fact, the choice of $\omega = \mathbf{W}$ with the initial condition $\mathbf{w}(t_0) = \mathbf{I}$ corresponds to the Jaumann rate. The major consequence of corotational rates is that if we choose the local axis system as the corotational one, constitutive laws integration can be performed as in small deformation. According to the decomposition of the Cauchy stress tensor into a deviatoric term $\mathbf{s}$ and an hydrostatic term p, one may obtain

$$\dot{\mathbf{s}}^{\mathrm{c}} = \mathbf{C}^{\mathrm{c}} : \mathbf{D}^{\mathrm{c}} \text{ and } \dot{p} = K \operatorname{tr}[\mathbf{D}]^{\mathrm{c}} \tag{11}$$

where K is the bulk modulus of the material and $\mathbf{C}$ is the fourth-order constitutive tensor. In this application, we use a J_2 plasticity model with nonlinear isotropic/kinematic hardening law. The associated von Mises yield criterion allows the use of the radial-return mapping strategy briefly summarized hereafter.

2.2.1. Elastic prediction

Due to the objectivity, and therefore the use of a corotational system, all the terms of the constitutive equation are corotational ones, so we can drop the subscript c in the following equations for simplicity. The elastic stresses are calculated using the Hooke's law, according to Eq. (11), by the following equation

$$p_{n+1}^{\text{trial}} = p_n + K \operatorname{tr}[\Delta\mathbf{e}] \text{ and } s_{n+1}^{\text{trial}} = s_n + 2G\Delta\mathbf{e} \tag{12}$$

where $\Delta\mathbf{e}$ is the corotational strain increment tensor between increment n and increment $n+1$. Hence, the deviatoric part of the predicted elastic stress is given by

$$\phi_{n+1}^{\text{trial}} = s_{n+1}^{\text{trial}} - \alpha_n \tag{13}$$

where α_n is the back-stress tensor. The von Mises criterion f is then defined by:

$$f_{n+1}^{\text{trial}} = \sqrt{\frac{2}{3}\phi_{n+1}^{\text{trial}} : \phi_{n+1}^{\text{trial}}} - \sigma_{\mathrm{v}} \tag{14}$$

where σ_{v} is the yield stress in the von Mises sense. Hence, if $f_{n+1}^{\text{trial}} \leq 0$, the predicted solution is physically admissible, and the whole increment is assumed to be elastic.

2.2.2. Plastic correction

If the predicted elastic stresses does not correspond to a physically admissible state, a plastic correction has to be

Box 1
Flowchart for explicit time integration

1. Initial conditions and initialization: $n = 0$; $\sigma_0 = \sigma(t_0)$; $x_0 = x(t_0)$; $v_0 = v(t_0)$
2. Update quantities: $n := n + 1$; $\sigma_n = \sigma_{n-1}$; $x_n = x_{n-1}$; $v_{n+1/2} = v_{n-1/2}$
3. Compute the time-step and update current time: $t_n = t_{n-1} + \Delta t$
4. Update nodal displacements: $x_n = x_{n-1} + \Delta t v_{n-1/2}$
5. Compute internal and external force vector $\mathbf{f}_n^{\text{int}}$, $\mathbf{f}_n^{\text{ext}}$
6. Integrate the conservative equations and compute accelerations: $\dot{v}_n = \mathbf{M}^{-1}(\mathbf{f}_n^{\text{ext}} - \mathbf{f}_n^{\text{int}})$
7. Update nodal velocities: $v_{n+1/2} = v_{n-1/2} + \Delta t \dot{v}_n$
8. Enforce essential boundary conditions: if node I on Γ_v
9. Output; if simulation not complete go to 2

performed. The previous trial stresses serve as the initial condition for the so-called return-mapping algorithm. This one is summarized by the following equation:

$$s_{n+1} = s_{n+1}^{\text{trial}} - 2G\gamma\mathbf{n} \tag{15}$$

where $\mathbf{n} = (\phi_{n+1}^{\text{trial}} / \|\phi_{n+1}^{\text{trial}}\|$ is the unit normal to the von Mises yield surface, and γ is the consistency parameter defined as the solution of the one scalar parameter (γ) nonlinear equation below:

$$f(\gamma) = \left|\phi_{n+1}^{\text{trial}}\right| - 2G\gamma - \sqrt{\frac{2}{3}}(\sigma_v(\gamma) - \|\alpha(\gamma)\|) = 0 \tag{16}$$

Eq. (16) is effectively solved by a local Newton iterative procedure [7]. Since $f(\gamma)$ is a convex function, convergence is guaranteed.

2.3. Time integration

All above equations are integrated by an explicit scheme associated with lumped mass matrices. The flowchart for the explicit time integration of the Lagrangian mesh is given in Box 1.

3. Object-oriented design

Object-oriented calculations have received extensive attention in computational mathematics and several engineering applications have already been published in computational journals. The benefits of OOP to implementation of FEM programs has been explored by Miller [8] and Mackie [9], and more recently applied to a Lagrangian analysis of thermo-plasticity coupled with ductile damage at finite strains by Zabaras and Srikanth [10]. The main used language in OOP is the C++, but some prospectives have been made to use other languages such as Java [11] with an extensive performance analysis. In this section only some aspects of the architecture are presented. Section 3.2 describes the basic classes and linear algebra. Some more specific aspects of the numerical implementation are presented in Section 3.3.

3.1. Overview of object-oriented programming

Traditionally, numerical softwares are based on use of a procedural programming language such as C or Fortran, in which the finite element algorithm is broken down into procedures that manipulate data. When developing a large application, the procedures are wrapped up in libraries which are used as modules and sometimes linked with external libraries such as the well-known Blas [12] one for linear algebra. Over the last few years, the use of object-oriented programming techniques has increased, leading to highly modularized code structure through user defined classes which can be seen as the association of data and methods. In OOP, an object is in fact an instance of a class. This approach is very attractive because of well-defined mechanisms for modular design and re-use of code. Briefly speaking, OOP encourages computer implementations of mathematical abstractions such as the work done concerning partial differential equations with Diffpack [13]. Efficient OOP results in the association of low level numeric computations encapsulated in high level abstraction such as inheritance, members and operators overload, abstraction and polymorphism or templates [14]. All those well-known characteristics for programmers are briefly presented here after, with their applications to numerical FEM program development.

Inheritance is a mechanism which allows the exploitation of commonality between objects. For example, assuming that we implements and `Element` base class containing methods such as integration of conservation laws over the element, one can derivate this class to create two-dimensional, three-dimensional or axisymmetric elements. Those inherited classes, for example the two-dimensional element class, may be derivated one more time to create rectangular or triangular planar elements defined with various number of nodes as shown in Fig. 1. Therefore, only the highly specialized code, as shape functions calculations, is implemented in those derived classes.

Members and operators overload allow an easy writing of mathematical functions such as matrices product using a generic syntax of the form $A = B * C$ where A, B and C are three matrices of compatible sizes. The overloaded operators $*$ and $=$ may use efficient matrix calculation and affectation algorithms associated with a set of basic checks like size compatibility of the operators. The same kind of operation is possible when the parameters are instances of different classes, such as the definition of the product of a matrix and a vector.

Abstraction is the ability of defining abstract objects using virtual member methods. Abstract classes allow the writing of generic algorithms and the easy extension of the existing code. The resulting class is said to have a polymorphic behavior. An example of an abstract class is the class Element defined in Fig. 1. In this case, we never create an instance of the class Element, but only instances of

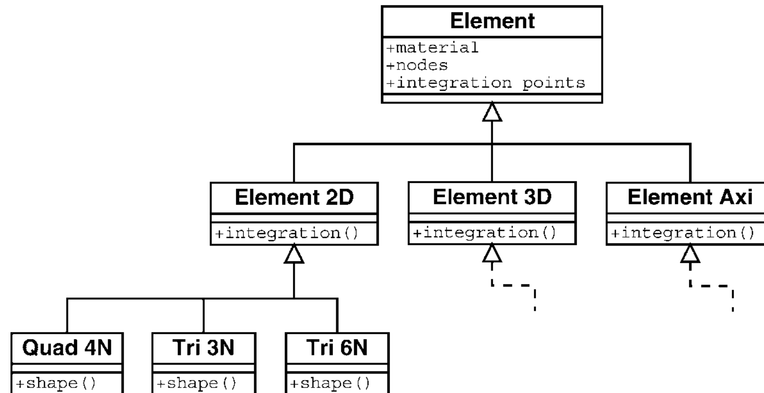

Fig. 1. UML diagram of the element class (simplified representation).

derived classes depending on the type of the element needed.

Template classes are generic ones, for example generic lists of any kind of object (nodes, elements, integration points, etc.). Templates are the fundamental enabling technology that supports construction of maintainable highly abstract, high performance scientific codes in C++ [15]. The use of OOP, and here the C++ language, has been criticized because its computational efficiency is commonly believed to be much lower than the one of comparable Fortran codes. Recent studies on relative efficiency of C++ numerical computations [15] have shown that there is a performance increase with optimized codes but libraries must be implemented carefully so that the CPU intensive numerics take place in functions that are easily optimized by C compilers. Creation of user defined class libraries with overload operators and encapsulation of low level operations on the basic data types allows for optimizations to be introduced incrementally through the development cycle. For example, in the linear algebra library, we use low level C and Fortran routines coming from the Lapack and Blas [12] libraries.

3.2. Basic classes and linear algebra

In a FEM application, the most logical point of departure will be the creation of a basic and mathematical class library. In this project, we have made the choice of developing our own basic classes such as the template class `List` (used to manage a list of any kind of object: `Node`, `Element`, etc.) and linear algebra ones such as `Vector`, `Matrix` and `Tensor` classes. Other projects described in literature are usually based on free or commercial libraries of C++ as the work done by Zabaras [16] with Diffpack. This choice has been done because we need linear algebra classes optimized for an explicit FEM program and in order to distribute this work according to the GNU general public license. Also, we did not wanted to waste a lot of time working with a free library becoming no longer free distributed from one day to another but rather a commercial package like the Diffpack library.

To illustrate one of the major advantage of the OOP, if we consider that the objects $\dot{\mathbf{s}}^c$ and $\mathbf{D}^c$ are instances of the `Tensor2` class, while the object $\mathbf{C}^c$ is an instance of the `Tensor4` class, this allows us to implement both terms of Eq. (11) in a simple, compact and elegant manner:

```
Tensor2 dS_c,D_c; // two instances of the Tensor2 class
Tensor4 C_c; // an instance of the Tensor4 class
double K,dP; // two scalars
··· // some various operations
dP = K * D_c.trace( ); // first equation
```
($\dot{p} = K \operatorname{tr}[\mathbf{D}^c]$)
```
dS_c = C_c * D_c; // second equation
```
($\dot{\mathbf{s}}^c = \mathbf{C}^c : \mathbf{D}^c$)

Box 2 presents the minimum parts of the two classes `Tensor2` and `Tensor4` needed to implement those C++ code lines.

Box 2
Headers of the Tensor2 and Tensor4 classes

```
class Tensor2{
...
Public:
Tensor2( ); // constructor
~Tensor2( ); // destructor
...
Tensor2 operator = (const Tensor2& t);
Friend Tensor2 operator * (const double& value, const Tensor2& tensor);
double trace( );
};
class Tensor4 {
...
Public:
Tensor4( ); // constructor
~Tensor4( ); // destructor
...
Tensor2 operator * (const Tensor2& t) const;
};
```

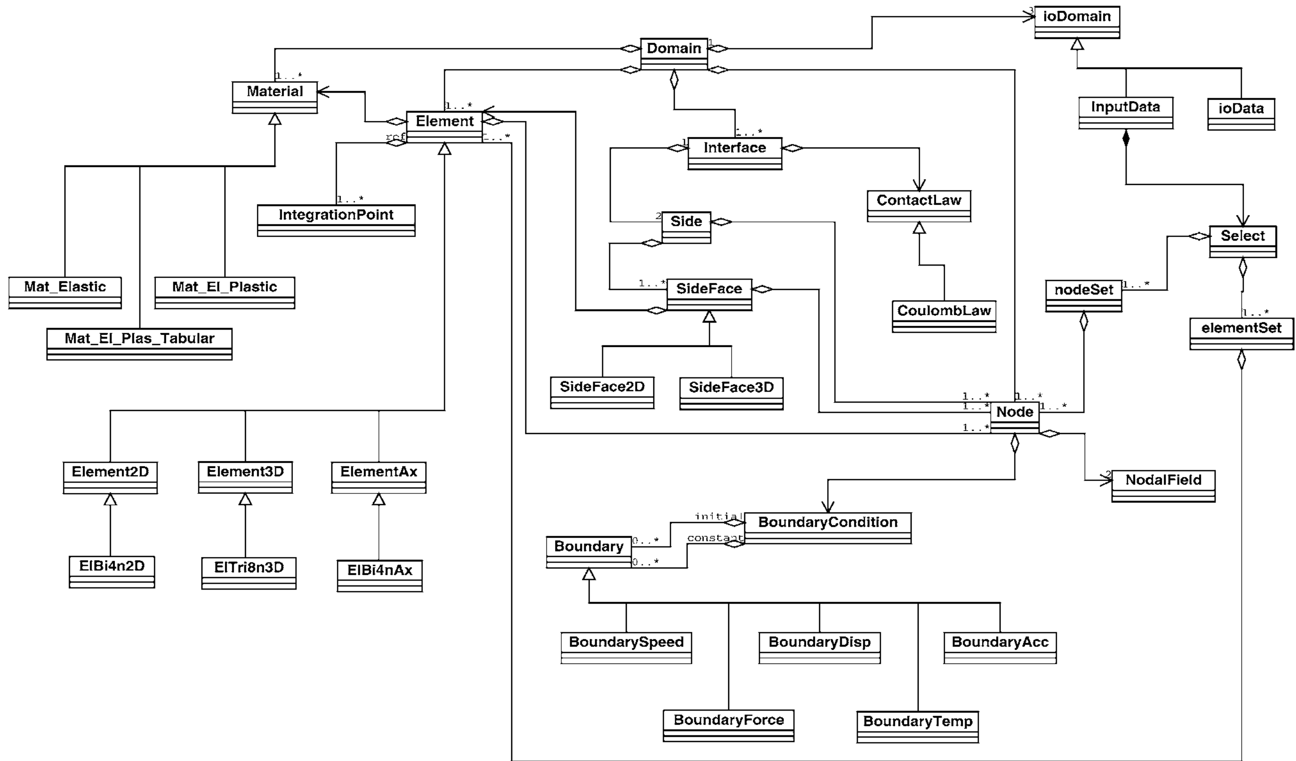

Fig. 2. Simplified UML diagram of the Object oriented framework.

- To implement those operations, we first need of course the default constructors and destructors of both classes `Tensor2` and `Tensor4`. Those two methods take no arguments here.
- For the first equation, we need the implementation of the method `trace()` used to compute the trace of a `Tensor2`, and an overload of the operator between a scalar value, and a `Tensor2` object. This one is to be declared as a friend method because we need to access some private members of the `Tensor2` class in this method.
- An overloading of the operator between the two classes `Tensor2` and `Tensor4` and of the operator $=$ between two `Tensor2` classes have been implemented for the second equation.

3.3. Finite element classes

As it can be found in many other papers dealing with the implementation of FEM [8,9,16] some basic classes have been introduced in this work. In this paragraph, an overview of the FEM classes is presented. Then, we focus on the implementation of the nonlinear material behavior used in this FEM code to illustrate the use of OOP in FEM.

3.3.1. Overview of the FEM classes

The FEM represented by the class `Domain` is mainly composed by the modules represented by the abstract classes `Node`, `Element`, `Material`, `Interface` and `ioDomain` as shown in Fig. 2.

The class `Node` contains nodal data, such as node number or nodal coordinates. Two instances of the `NodalField` class containing all nodal quantities at each node are linked to each node of the structure. The first one is relative to time t, the second one to $t + \Delta t$. At the end of the increment, we just have to swap the references to those objects to transfer all quantities from one step to another (see step 2 of the explicit time integration flowchart in Box 1). Boundary conditions (BC) through the `BoundaryCondition` class may affect the behavior of each node in particular subtreatments such as contact conditions, external forces or thermal flux treatment. A list of BC is attached to each node, this gives the ability to change the BC during the main solve loop. For example a call to the `Node::updatePosition()` method changes the coordinates according to the current BC.

The class `Element` is a virtual class that contains the definition of each element of the structure (see Fig. 1). This class serves as a base class for a number of other classes depending on the type of analysis and the nature of elements needed. The difference between all derived element classes concerns for example the shape functions. Of course, it is possible to mix together various types of elements in the same computation. The only restriction here concerns the first level of inheritance, you cannot have an axisymmetric element and a plane strain one in the same model. Each element of the structure contains a given number of nodes,

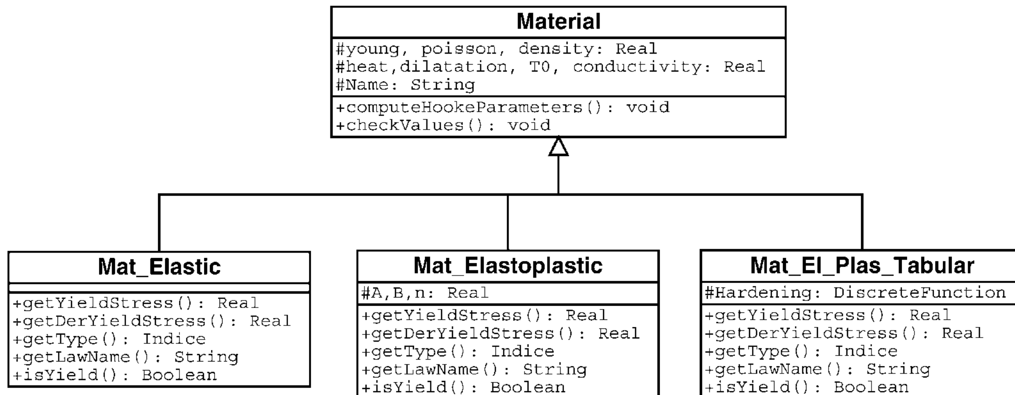

Fig. 3. UML diagram of the material class (simplified representation).

an arbitrary number of integration points (see IntegrationPoint class) and refers an associate constitutive law through the material definition.

The Interface class contains all definitions concerning the contact interfaces the contact law through the ContactLaw class and the contact definition through the Side class. We do not present more this one here. The class ioDomain is used to serve as an interface between the Domain and input/output files. This class serves as a base class for many other derived classes which implements specific interfaces for various file formats. The most important one is the class InputData used to read the model.

The class Material is used for the definition of the materials used in various models. This class is a generalization for all possible kinds of material definition. Some details concerning the implementation of this class are given here after.

3.3.2. Implementation of the nonlinear material behavior

The isotropic inelastic material behavior is defined via the evolution of the equivalent plastic strain σ_v and the evolution of a number of state variables. A simplified UML diagram concerning the Material class is presented in Fig. 3. From this later, we can see that the class Material is virtual and serves as a base class for other material classes such as Mat_Elastic, Mat_Elastoplastic or Mat_El_Plas_Tabular. The first one is used for the definition of an elastic material, the second one for an elastoplastic material of the form $\sigma_v = A + B\bar{\varepsilon}^{p^n}$ where A, B and n are material constants, and the last one allows us to define an arbitrary form for the strain hardening function using a tabular function $\sigma_v = f(\bar{\varepsilon}^p)$.

Various constitutive models are represented as virtual functions in classes derivated from the Material base class. Some attributes and methods are implemented in the base class Material, while other attributes or methods are implemented in the derived classes. First ones concern methods and attributes that are common to each kind of material. For example the Young's modulus E, the density ρ or the Poisson ratio ν are common attributes shared by each kind of constitutive law. The A, B and n material constants are attributes dedicated to the Mat_Elastoplastic class. The definition of the nonlinear hardening law through a DiscreteFunction class is dedicated to the Mat_El_Plas_Tabular class. To define a new material law, one has to derivate a new class from the Material class. Box 3 presents a summary of the basic functionalities

Box 3
Headers of the material and Mat_El_Plast_Tabular classes

```
class Material {
  friend class List⟨Material * ⟩;
protected:
  Tensor4 C;
  double young, poisson, density;
  double heat, dilatation, T0 conductivity;
  String Name;
public:
  Material( );
  Material (const Material& material);
  virtual ~ Material( );
  ...
  virtual String getLawName( ) = 0;
  virtual double getYieldStress( ) = 0;
  virtual double getDerYieldStress( ) = 0;
  void computeHookeParameters( );
  void checkValues( );
  Friend ostream & operator ≪ (ostream & stream, Material& material);
  ...
};
class Mat_El_Plas_Tabular: public Material {
protected:
  DiscreteFunction * function; // used to define the tabular function
public:
  Mat_El_Plas_Tabular ( );
  Mat_El_Plas_Tabular (const Mat_El_Plas_Tabular& material);
  ~ Mat_El_Plas_Tabular ( );
  ...
  String getLaw Name( );
  double getYieldStress( );
  double getDerYieldStress( );
  void setFunction(DiscreteFunction * func) {function = func;}
  DiscreteFunction * getFunction( ) {return (function);}
  friend ostream & operator ≪ (ostream & os, Mat_El_Plas_Tabular& material);
  ...
};
```

of class Material and Mat_El_Plas_Tabular. The main effort to implement a new constitutive model is to define the getYieldStress( ) and getDerYieldStress( ) methods which must return, respectively, the value of the hardening parameter $\sigma_v = f(\bar{\varepsilon}^p, \ldots)$ and the slope of the hardening law $h = \partial\sigma_v(\bar{\varepsilon}^p, \ldots)/\partial\bar{\varepsilon}^p$.

3.4. Pre-processing language

In the FEM code DynELA, we developed a specific high level language using the Lex and Yacc [17] utilities. This language has a grammar presenting analogies with C++. The most important points are summarized here after:

- fully free-form language supporting classic features such as comments, files inclusion through #include commands
- supports for various computations between reals or vectors, arithmetic, trigonometric, increments or variables comparisons
- includes tests (if, then and else) and loops (for and while)
- i/o functionalities such as cout, fopen, fclose or < <
- many other useful features (we refer to the DynELA user manual [18]).

As an example we present here after a semi-analytic declaration of the nonlinear hardening law used in the necking of a circular bar example (see Eq. (17) and related parameters in Table 1). This nonlinear hardening law is well described using the Mat_El_Plas_tabular class associated with a discrete function. The definition of this hardening law using the DynELA specific language is given by:

```
// local variables declaration
sv_0 = 0.45e9;
sv_inf = 0.715e9;
delta = 16.93;
h = 0.12924;
// hardening law declaration using a discrete function
FOR (eps = 0;eps < 1;eps + = (eps/5) + 1/1000){
  DISCRETE FUNCTION: hard_funct { // name of the function
    eps,sv_0 + (sv_inf-sv_0) * (1 − EXP(−delta * (eps)) + h * (eps);  // add a new point
  };
};
// material declaration
MATERIAL: steel {
  YOUNG: 2.069E + 11; // Young modulus
  NU: 2.9E-01; // Poisson ratio
  DENSITY: 7.8E + 03; // Density
  ELASTOPLASTIC TABULAR { DISCRETE FUNCTION hard_funct; };
};
```

Table 1
Material properties for the circular bar

Young's modulus	E	206.9 GPa
Poisson ratio	ν	0.29
Initial flow stress	σ_v^0	450.0 MPa
Residual flow stress	σ_v^∞	715.0 MPa
Linear hardening	h	0.12924
Saturation exponent	δ	16.93

In this example we first begin the block with the definition of the material constants of the hardening law equation. By default, if no type specification is done, the pre-processor assumes that the variable is a scalar. Vectors, strings or other types are also available. Then, in the example we use a classic FOR loop in the range [0:1] to calculate and create each point of the hardening law via the definition of a discrete function named here hard_funct. This FOR loop have an increasing increment size because more points are needed for such function near the origin. Then, in the last part of the program, we define a new material, called steel here, and associates the previously defined discrete function hard_funct to it. This method allows us to modify in a simply way the definition of the hardening law by changing the variable values at top of the program. This can also be done externally from other program, and leads to parametrized numerical models used in identification of constitutive law parameters.

4. Numerical validations

The objective of this section is to assess the numerical implementations made in DynELA concerning the J_2 flow theory presented in Section 3. For this validation we consider two representative examples related to well documented numerical experiments available in literature, the necking of a circular bar subjected to traction forces and the simulation of a direct Taylor test impact. All computations were performed with an AMD K6-3 400 MHz under Linux.

4.1. Necking of a circular bar

This experimentally well documented example [7,19] is concerned with necking of a circular bar with a radius of 6.413 mm and a length 52.34 mm, subjected to uniaxial tension resulting from an axial elongation of 14 mm. This example serves here as a testbed for the plastic algorithm developed in DynELA. The material considered here is a special steel (A533, Grade B, Class 1), with a general

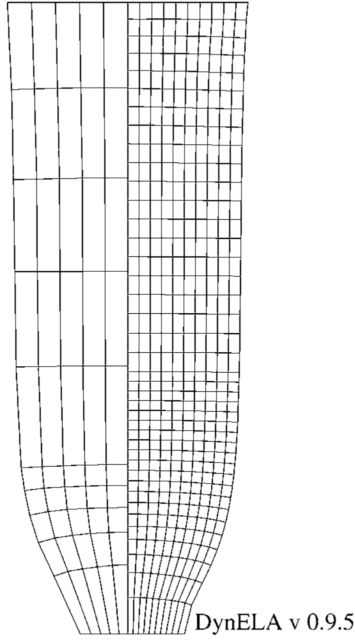

Fig. 4. Necking of a circular bar: final meshes obtained for 50 (left) and 400 (right) elements.

nonlinear hardening law of the form:

$$\sigma_v = \sigma_v^0 + (\sigma_v^\infty - \sigma_v^0)(1 - \exp(-\delta\bar{\varepsilon}^p) + h\bar{\varepsilon}^p) \quad (17)$$

Material properties given by Norris et al. [19] are reported in Table 1. This calculation problem is nonlinear, both by the constitutive equation and by the large deformation and rotation that occur at necking.

Two different meshes consisting of 50 and 400 elements are considered to assess the influence of the discretization. Only half of the axisymmetric geometry of the bar has been meshed in the model. This example is a quasi-static one, but because we used an explicit algorithm, we introduced a prescribed velocity of 7 m/s at the top of the workpiece to control the displacement. This rate corresponds to the one used in the numerical model presented by Norris et al.

Fig. 4 reports final meshes obtained for the full elongation. In this figure, the deformed solution obtained with the coarse and the finer meshes are in good agreement. Fig. 5 shows the ratio of the current to initial radius at the necking section vs. the axial displacement. It is a comparison between numerical (this work and Simo and Hughes [7]) and experimental results [19]. The results are in good agreement with experimental and previously reported computations.

Table 2
Material properties of the OHFC copper rod for the Taylor test

Young's modulus	E	117.0 GPa
Poisson ratio	ν	0.35
Density	ρ	8930 kg/m^3
Initial flow stress	σ_v^0	400.0 MPa
Linear hardening	h	100.0 MPa

4.2. Impact of a copper rod

The impact of a copper rod on a rigid wall problem, known as the Taylor impact problem, is a standard benchmark for dynamics computer codes. This problem simulates a high velocity impact of a copper rod on a rigid wall, it is used by many authors such as Liu et al. [20]. The initial dimensions of the rod are $r_0 = 3.2$ mm and $l_0 = 32.4$ mm. The impact is assumed to be frictionless and the impact velocity is set to 227 m/s. The final

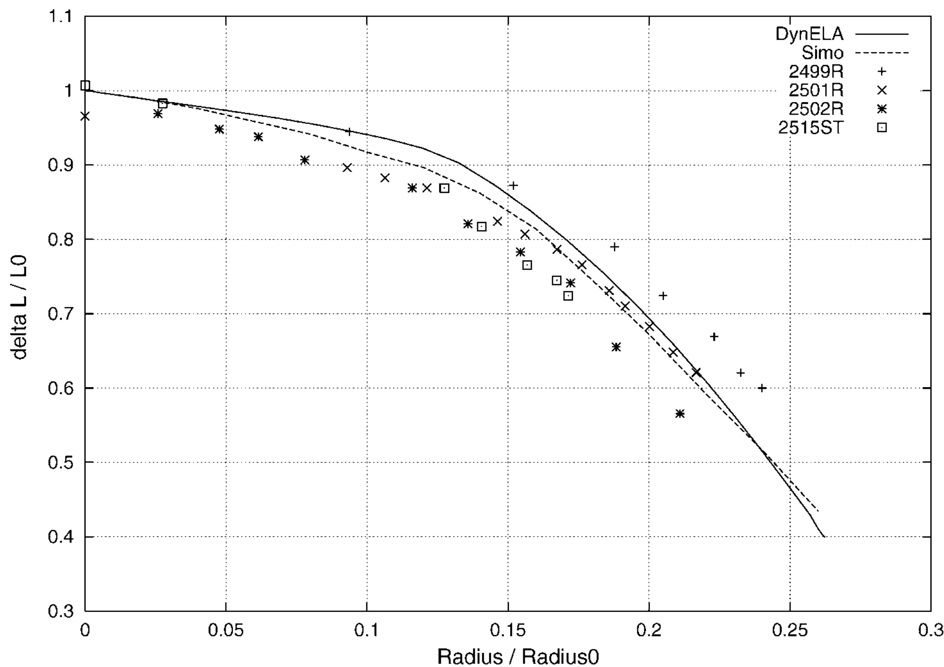

Fig. 5. Necking of a circular bar: ratio of the current to initial radius at the necking section versus axial displacement.

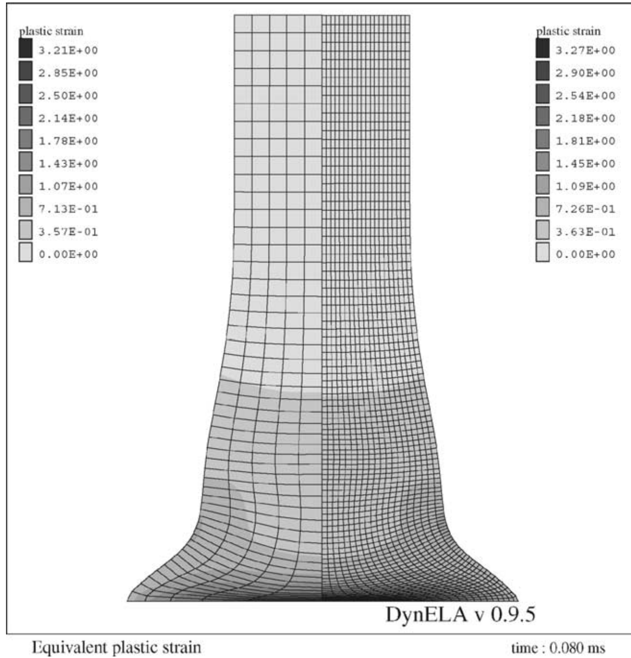

Fig. 6. Impact of a copper rod: equivalent plastic strains for the two meshes used.

configuration is obtained after 80 μs. The constitutive law is elastoplastic with a linear isotropic hardening, material properties given in Ref. [20] corresponding to an OFHC copper reported in Table 2. Here again, only half of the axisymmetric geometry of the rod has been meshed in the model. Two different meshes were used, the first one with 250 elements (50×5), and the second one with 2000 elements (20×100).

Fig. 6 shows the equivalent plastic strain contour plot for both meshes. Comparison between left- and right-hand sides of this figure shows a good level of agreement both for the final geometry and for the equivalent plastic strain contour plot with previously reported results. Table 3 reports a comparison for the final length l_f, the footprint radius r_f and the maximum equivalent plastic strain $\bar{\varepsilon}^p_{max}$ obtained with our finite element code and other numerical results such as the one obtained by Liu et al. [20] or the same simulation problem with the Abaqus Explicit [21] program. The differences between the solutions are reasonable.

5. Conclusions

An object-oriented simulator was developed for the analysis of large inelastic deformations and impact processes. Several benchmark test problems were examined to demonstrate the accuracy of the developed algorithms. The benefits of using an OOP approach in comparison with traditional programming language approaches were presented in this paper. The use of OOP provides us with the ability of better representing, through the definition of classes and inheritance, the physical, mathematical and geometric structures of the kinematics and constitutive aspects of a FEM analysis. The main purpose of this FEM development is to serve as a testbed for new and more efficient algorithms related to various parts of a FEM program, such as new contact algorithms (here, the contact is included but has not been presented) or more efficient constitutive integration schemes. One of the main advantages of the present FEM code is that the class hierarchies adopted allow the implementation of additional constitutive models such as new constitutive laws, new elements or contact laws by derivating this new feature from existing one using the inheritance feature.

One of the future use of this simulator is related to inverse problem when one wants to make a parameter identification of the material coefficients. This FEM code is continuously developed and new features are implemented such as a new constitutive algorithm including damage effects or the use of various multi-grid resolution algorithms.

Table 3
Comparison of numerical results for the Taylor test impact

FEM code	r_f	l_f	$\bar{\varepsilon}^p_{max}$
DynELA 2000	7.17	21.42	3.26
DynELA 250	7.12	21.43	3.21
Abaqus	7.08	21.48	3.23
Liu	7.15	21.42	–

References

[1] Ortiz M, Simo JC. An analysis of a new class of integration algorithms for elastoplastic constitutive relations. Int J Numer Meth Engng 1986; 23:353–66.

[2] Zabaras N, Arif AFM. A family of integration algorithms for constitutive equations in finite deformation elasto-viscoplasticity. Int J Numer Meth Engng 1992;33:59–84.

[3] Stroustrup B. The C++ programming language, 2nd ed. Reading, MA: Addison-Wesley; 1991.

[4] Langer SH, comparison A. A comparison of the floating-point performance of current computers. Comput Phys 1998;12(4):338–45.

[5] Cross JT, Masters I, Lewis RW. Why you should consider object-oriented programming techniques for finite element methods. Int J Numer Meth Heat Fluid Flow 1999;9:333–47.

[6] Hughes TJR. The Finite element method; linear static and dynamic finite element analysis. New York: Prentice-Hall; 1987.

[7] Simo JC, Hughes TJR. Computational inelasticity. Berlin: Springer; 1998.

[8] Miller GR. An object oriented approach to structural analysis and design. Comput Struct 1991;40(1):75–82.

[9] Mackie RI. Object oriented programming of the finite element method. Int J Numer Meth Engng 1992;35:425–36.

[10] Zabaras N, Srikanth A. Using objects to model finite deformation plasticity. Engng Comput (Spec Issue Object Oriented Comput Mech Tech) 1999;15:37–60.

[11] Nikishkov GP. Performance of a finite element code written in Java. Adv Comput Engng Sci 2000;1:264–9.

[12] Lawson C, Hanson R, Kincaid D, Krogh F. Basic linear algebra subprograms for fortran usage. ACM Trans Math Software 1979;5: 308–29.

[13] Daehlen M, Tveito A. Numerical methods and software tools in industrial mathematics. Basel: Berkhauser; 1997.
[14] Haney S, Crotinger J. How templates enables high-performance scientific computing in C++. Comput Sci Engng 1999;66–72.
[15] Haney SW. Is C++ fast enough for scientific computing? Comput Phys 1994;8(6):690.
[16] Zabaras N, Bao Y, Srikanth A, Frazier WG, continuum A. Lagrangian sensitivity analysis for metal forming processes with applications to die design problems. Int J Numer Meth Engng 2000;48:679–720.
[17] Mason J, Levine D. Lex and Yacc, 2nd ed. No. 1-56592-000-7; 1992.
[18] Pantalé O. User manual of the finite element code DynELA v. 0. 9. 5. Av d'Azereix 65016 Tarbes, France: Laboratoire LGP ENI Tarbes; 2001.
[19] Norris DM, Morran JRB, Scudder JK, Quinones DF. A computer simulation of the tension test. J Mech Phys Solids 1978;26:1–19.
[20] Liu WK, Chang H, Chen JS, Belytschko T. Arbitrary Lagrangian–Eulerian Petrov–Galerkin finite elements for nonlinear continua. Comput Meth Appl Mech Engng 1988;68:259–310.
[21] Abaqus, reference manual, Hibbitt, Karlsson and Sorensen Inc, 100 Medway Street, Providence, RI 02906-4402, USA; 1989.