Development of an object-oriented finite element program: application to metal-forming and impact simulations

O. Pantalé[a],*, S. Caperaa[a], R. Rakotomalala[a]

[a] *L.G.P C.M.A.O - E.N.I.T, 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. In the same time, the finite element method has become a widely used tool for researchers and engineers. Recent advances in computational software have made possible to solve more physical and complex problems such as coupled problems, non-linearities, high strain and high-strain rate problems. In this field, an accurate analysis of large deformation inelastic problems occuring in metal-forming or impact simulations is extremely important as a consequence of high amount of plastic flow.

In this presentation, the object-oriented implementation, using the C++ language, of an explicit finite element code called DynELA is presented. The object-oriented programming (OOP) leads to better-structured codes for the finite element method and facilitates the development, the maintainability and the expandability of such codes. The most significant advantage of OOP is in the modeling of complex physical systems such as deformation processing where the overall complex problem is partitioned in individual sub-problems based on physical, mathematical or geometric reasoning.

We first focus on the advantages of OOP for the development of scientific programs. Specific aspects of OOP, such as the inheritance mechanism, the operators overload procedure or the use of template classes are detailed. Then we present the approach used for the development of our finite elment code through the presentation of the kinematics, conservative and constitutive laws and their respective implementation in C++. Finally, the efficiency and accuracy of our finite element program are investigated using a number of benchmark tests relative to metal forming and impact simulations.

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 metal-forming or 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.

* tel:+(33).5.62.44.27.00, fax:+(33).5.62.44.27.08, E-mail: Olivier.Pantale@enit.fr.

In this presentation an object-oriented (OO) implementation of an explicit finite element program called DynELA is presented. This FEM program is written in C++ [1]. The development of object-oriented programming (OOP) leads to better structured codes for the finite element method and facilitates the development and maintainability [2]. 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.

2 Governing equations and 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 non-linearities are included in this setting. In the next paragraph, we summarize some basic results concerning non-linear mechanics relevant to our subsequent developments.

2.1 Basic kinematics and constitutive equations

One of the most important aspect in the development of a finite element code for non-linear mechanics involves the proper determination of the kinematic description. In a Lagrangian description let's $\overrightarrow{X}$ be the reference coordinates of a material point in the reference configuration $\Omega_X \subset \mathbb{R}^3$ at time $t = 0$, and $\overrightarrow{x}$ be the current coordinates of the same material point in the current configuration $\Omega_x \subset \mathbb{R}^3$ at time t. The motion of the body is then defined by $\overrightarrow{x} = \phi(\overrightarrow{X}, t)$. Let $\mathbf{F} = \frac{\partial \overrightarrow{x}}{\partial \overrightarrow{X}}$ be the deformation gradient with respect to the reference configuration Ω_X and $\mathbf{C} = \mathbf{F}^T\mathbf{F}$ the left Cauchy-Green tensor. According to the polar decomposition theorem, $\mathbf{F} = \mathbf{RU} = \mathbf{VR}$, $\mathbf{U}$ and $\mathbf{V}$ are the right and left stretch tensors respectively and $\mathbf{R}$ is the rotation tensor. By computing the rate of change of the deformation gradient $\mathbf{F}$, one may introduce the spatial velocity gradient $\mathbf{L} = \dot{\mathbf{F}}\,\mathbf{F}^{-1}$ where $\dot{()}$ is the time derivative of $()$. The symmetric part of $\mathbf{L}$, denoted by $\mathbf{D}$, is the spatial rate of deformation and its skew-symmetric part $\mathbf{W}$ is the spin tensor. According to this kinematics, the mass, momentum and energy equations which govern the continuum are given bellow:

$$\dot{\rho} + \rho \mathrm{div}\, \overrightarrow{v} = 0 \tag{2.1}$$

$$\rho\, \dot{\overrightarrow{v}} = \rho\, \overrightarrow{f} + \mathrm{div}\sigma \tag{2.2}$$

$$\rho\, \dot{e} = \sigma : \mathbf{D} - \mathrm{div}\, \overrightarrow{q} + \rho r \tag{2.3}$$

where ρ is the mass density, $\overrightarrow{v}$ the material velocity, $\overrightarrow{f}$ the body force vector, σ the Cauchy stress tensor, e the specific internal energy, r the body heat generation rate and $\overrightarrow{q}$ 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 finite element method (FEM) is used for the discretization of the conservative equations. An explicit integration scheme is then adopted for time discretization of those equations. The matricial forms of equations 2.1, 2.2 and 2.3 are obtained, according to the finite element method, by subdividing the domain of interest Ω_x into a finite number of elements Ω_e^h. This leads to the following matricial forms of the conservative equations below:

$$\mathbf{M}^\rho\, \dot{\rho} + \mathbf{K}^\rho \rho = 0 \tag{2.4}$$

$$\mathbf{M}^v \dot{\vec{v}} + \mathbf{F}^{int} = \mathbf{F}^{ext} \tag{2.5}$$

$$\mathbf{M}^e \dot{e} + \mathbf{g} = \mathbf{r} \tag{2.6}$$

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 equations 2.4 to 2.6:

$$\begin{aligned} \mathbf{M}^\rho &= \int_{\Omega_x} \varphi^{\rho^T} \varphi^\rho d\Omega_x \\ \mathbf{K}^\rho &= \int_{\Omega_x} \varphi^{\rho^T} \nabla v \varphi^\rho d\Omega_x \end{aligned} \tag{2.7}$$

$$\begin{aligned} \mathbf{M}^v &= \int_{\Omega_x} \rho \varphi^{v^T} \varphi^v d\Omega_x \\ \mathbf{F}^{int} &= \int_{\Omega_x} \nabla \varphi^{v^T} \sigma d\Omega_x \\ \mathbf{F}^{ext} &= \int_{\Omega_x} \rho \varphi^{v^T} \vec{b}\, d\Omega_x + \int_{\Gamma_x} \varphi^{v^T} \vec{t}\, d\Gamma_x \end{aligned} \tag{2.8}$$

$$\begin{aligned} \mathbf{M}^e &= \int_{\Omega_x} \varphi^{e^T} \varphi^e d\Omega_x \\ \mathbf{g} &= \int_{\Omega_x} \nabla \varphi^{e^T} \vec{q}\, d\Omega_x \\ \mathbf{r} &= \int_{\Omega_x} \varphi^{e^T} (\sigma : \mathbf{D} + \rho r)\, d\Omega_x - \int_{\Gamma_x} \varphi^{e^T} \theta d\Gamma_x \end{aligned} \tag{2.9}$$

In previous equations, $\mathbf{M}^{()}$ are consistent mass matrices, $\mathbf{F}^{ext}$ is the external force vector and $\mathbf{F}^{int}$ 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 (2.4-2.6) without the need of any matrix inversion algorithm.

2.2 Constitutive law

Concerning the constitutive law, we use a J_2 plasticity model with nonlinear isotropic/kinematic hardening. The algorithm presented here applies to both the three-dimensional, axially symmetric and plane strain cases. The simplicity of the von Mises yield criterion allows the use of the radial-return mapping strategy briefly summarized hereafter.

2.2.1 Elastic prediction

According to the decomposition of the Cauchy stress tensor σ into a deviatoric part $\mathbf{s}$ and an hydrostatic term p , the elastic stresses are calculated using the Hooke's law, by the following equations:

$$p_{n+1}^{trial} = p_n + K tr[\Delta \mathbf{e}] \tag{2.10}$$

$$\mathbf{s}_{n+1}^{trial} = \mathbf{s}_n + 2G\Delta \mathbf{e} \tag{2.11}$$

where $\Delta \mathbf{e}$ is the strain increment tensor between increment n and increment $n+1$, K is the Bulk modulus of the material, $tr[\Delta \mathbf{e}]$ is the trace of the strain increment tensor and G is the shear modulus. Hence, the deviatoric part of the predicted elastic stress is given by:

$$\phi_{n+1}^{trial} = \mathbf{s}_{n+1}^{trial} - \alpha_n \tag{2.12}$$

where α_n is the back-stress tensor (in our case, the center of the von Mises sphere in the stresses space). The von Mises criterion f is defined by

$$f_{n+1}^{trial} = \sqrt{\frac{2}{3}\phi_{n+1}^{trial} : \phi_{n+1}^{trial}} - \sigma_v \tag{2.13}$$

where σ_v is the yield stress in the von Mises sense. Hence, if $f_{n+1}^{trial} \leq 0$, 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 doesn't correspond to a physically admissible state, a plastic correction has to be performed. The previously trial stresses serves as the initial condition for the so-called return mapping algorithm. This one is summarized by the following equation:

$$\mathbf{s}_{n+1} = \mathbf{s}_{n+1}^{trial} - 2G\gamma\mathbf{n} \tag{2.14}$$

where $\mathbf{n} = \frac{\phi_{n+1}^{trial}}{\|\phi_{n+1}^{trial}\|}$ 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) non-linear equation below:

$$f(\gamma) = \left\|\phi_{n+1}^{trial}\right\| - 2G\gamma - \sqrt{\frac{2}{3}}\left(\sigma_v(\gamma) - \|\alpha(\gamma)\|\right) = 0 \tag{2.15}$$

Equation (2.15) is effectively solved by a local Newton iterative procedure [3]. Since $f(\gamma)$ is a convex function, convergence is guaranteed. Only a very few number of iterations are needed to obtain the final solution, so the algorithm isn't cost expensive.

2.3 ***Time integration***

As briefly presented earlier, the coupled equations will be integrated by an explicit scheme associated with lumped mass matrices. The integration algorithm is based on the central difference scheme given hereafter

$$\overrightarrow{v}_{t+\frac{\Delta t}{2}} = \overrightarrow{v}_{t-\frac{\Delta t}{2}} + \dot{\overrightarrow{v}}_t\,\Delta t \tag{2.16}$$

$$x_{t+1} = x_t + \Delta t\,\overrightarrow{v}_{t+\frac{\Delta t}{2}} \tag{2.17}$$

This integration scheme is conditionally stable, hence, the time increment value Δt is subjected to the Courant stability criterion. The flowchart for explicit time integration of the Lagrangian mesh is given in box 1.

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 $\mathbf{f}_n^{int}, \mathbf{f}_n^{ext}$
6. Integrate the conservative equations and compute accelerations: $\dot{v}_n = \mathbf{M}^{-1}(\mathbf{f}_n^{ext} - \mathbf{f}_n^{int})$
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 goto 2.

3 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[4] one for linear algebra. Object-oriented programming uses user defined classes which can be seen as the association of data and methods (remembering that what we call an object is in fact an instance of a class).

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, but studies on relative efficiency of C++ numerical computations [5] have shown that there's a performance increase with optimized codes. A survey of the main object-oriented features is presented here after:

- Inheritance is a mechanism which allows the exploitation of commonality between objects. For example, as illustrated on figure 1, we can define many classes derivated from the class Element which differ by the level of specialization that they present. Therefore, only the highly specialized code, as shape functions calculations for example, are implemented in those derived classes.

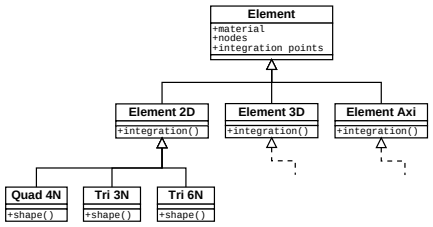

Figure 1: UML diagram of the element class (simplified representation)

- Member and operator overload allows an easy writing of mathematical functions such as matrix products using a generic syntax of the form $A = B * C$ where A, B and C are three matrices of compatible sizes. The same kind of operation also is possible when the parameters are instances of different classes.

- Template classes are generic ones, for example generic lists of any kind of object (nodes, elements, integration points...). Templates are the fundamental enabling technology that supports construction of maintainable highly abstract, high performance scientific codes in C++ [6].

For further details concerning OOP we refer to Stroustrup [1].

3.1 Basic classes used in our FEM application

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 and linear algebra classes. Other projects described in literature are usually based on free or commercial libraries of C++ as the work done by Zabaras [7] with Diffpack. This choice has been done because we need linear algebra classes optimized for an explicit FEM program and in

order to distribute the FEM program with the GNU general public license. In the linear algebra part, we use low level C and Fortran routines comming from the Lapack and Blas[4] libraries. Highly optimized C and Fortran routines collected in libraries are easily called within a C++ method.

3.2 Overview of the finite element classes

As it can be found in many other papers dealing with the implementation of FEM [8, 9, 7] some basic FEM classes have been introduced in this work. In this paragraph, an overview of the FEM classes is presented. 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 on figure 2.

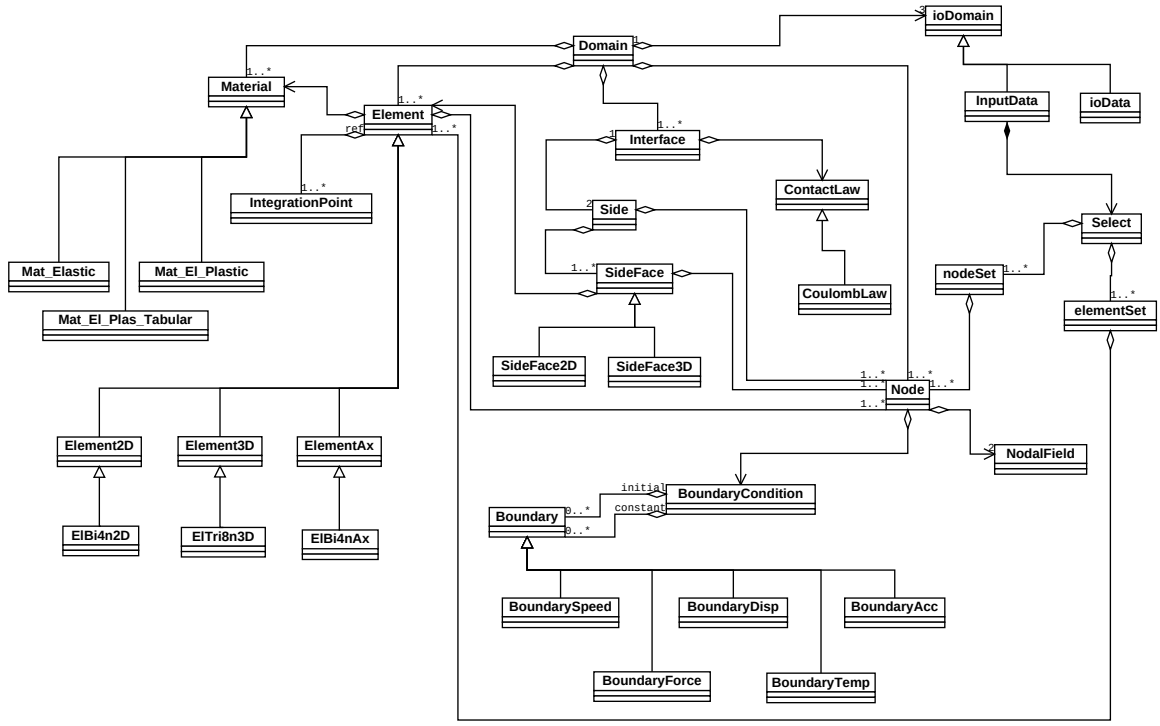

Figure 2: Simplified UML diagram of the Object Oriented Framework

- The class `Node` contains nodal data, such as node number, nodal coordinates... Two instances of the `NodalField` class containing all nodal quantities at each node are linked to each node of the structure. 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 through the `BoundaryCondition` class affect the behaviour of each node. Those boundary conditions appears through a dynamic list attached to each node, thus, one may attach or detach any type of condition during the main solve loop.

- The class `Element` is a virtual class that contains the definition of each element of the structure (we refer to figure 1 for a more detailed description of the `Element` class). 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. Of course, it is possible to mix together various types

of elements in the same computation. Each element of the structure contains a number of nodes, depending on its shape, may have an arbitrary number of integration points (see `IntegrationPoint` class) and refers an associate constitutive law through the `Material` class.

- The `Interface` class contains all definitions concerning the contact interfaces of the model including the contact law through the `ContactLaw` class and the contact definition through the `Side` class.
- The class `ioDomain` is used to serve as an interface between the `Domain` and input/output files. The class `ioDomain` serves as a base class for many other derived classes which implement specific interfaces for various file formats. The most important of them is the class `InputData` used to read the model from the specific preprocessor language.
- 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.

3.3 User interface

The very first developments we made concerning this project were in C and concerned only the pre and post-processing of FEM computations concerning numerical cutting [10]. This work was based on the RADIOSS finite volume program. Therefore, this works inherits some methods developped for those applications, and the pre and post-processor of DynELA may be seen as a new version of those two applications. DynELA uses a dedicated graphic post-processor (see figure 3 for a screen-copy). Many features developed earlier were included in this post-processor such as higly detailed postcsript output, OpenGL rendering, picking interface and curves treatment.

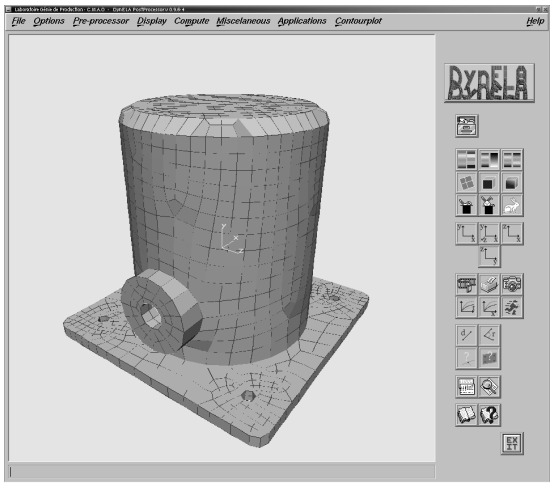

Figure 3: Graphic user interface of the DynELA FEM code

DynELA uses a specific language for the pre-processing of files presenting analogies with C++. The most important features are summarized here after:

- fully free format 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 usefull features (we refer to the DynELA user manual [11]).

4 Numerical application

As an illustration, we present in this paragraph a numerical example concerned with a dynamic compression of a vertical thin walled cylinder under uniaxial compression. Numerical results obtained with our FEM application are compared with the one obtained with Abaqus/Explicit. We used an isotropic linear-elastoplastic constitutive law of the form $\sigma_v = A + B\overline{\varepsilon^p}^n$. Initial dimensions of the specimen and material properties are given as follows:

inner radius=	$10.0\,mm$	E=	$206\,GPa$	A=	$1250\,MPa$
height=	$28.0\,mm$	ν=	$0,35$	B=	$685\,MPa$
thickness=	$1.0\,mm$	ρ=	$7836.1\,kg/m^3$	n=	$1,0$

The corresponding mesh reported on figure 4 is composed of 450 axisymmetric elements (6x75).

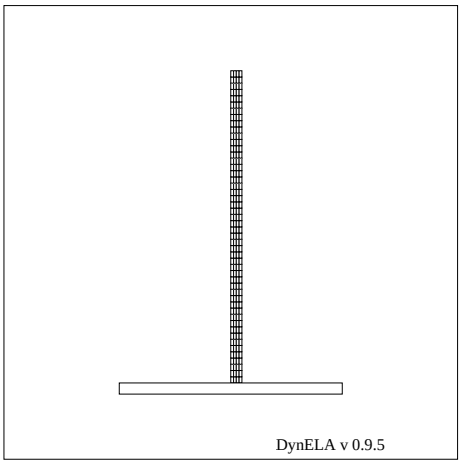

Figure 4: Initial mesh used for the thin walled cylinder under uniaxial compression

The associated boundary conditions are given by:

- All nodes located at the top of the cylinder have a prescribed vertical displacement with a constant speed $v = 100m/s$ and a nul horizontal displacement.
- All nodes located at the bottom of the cylinder are subjected to a frictionless contact with a rigid horizontal surface.

During the simulation, the inner surface of the cylinder may be in contact with the horizontal surface, so we take it into account and declared this surface as a contact surface in DynELA. We assume here also a frictionless contact. The top of the cylinder is subjected to a total vertical displacement of $d_v = 15mm$, therefore, total height reduction is about $53,6\%$.

Figure 5 reports final equivalent plastic strain contourplots obtained with DynELA FEM code and Abaqus/Explicit. On this figure, the deformed geometry obtained with the DynELA code is in good agreement with the one obtained with Abaqus/Explicit. Numerical results are also in good agreement.

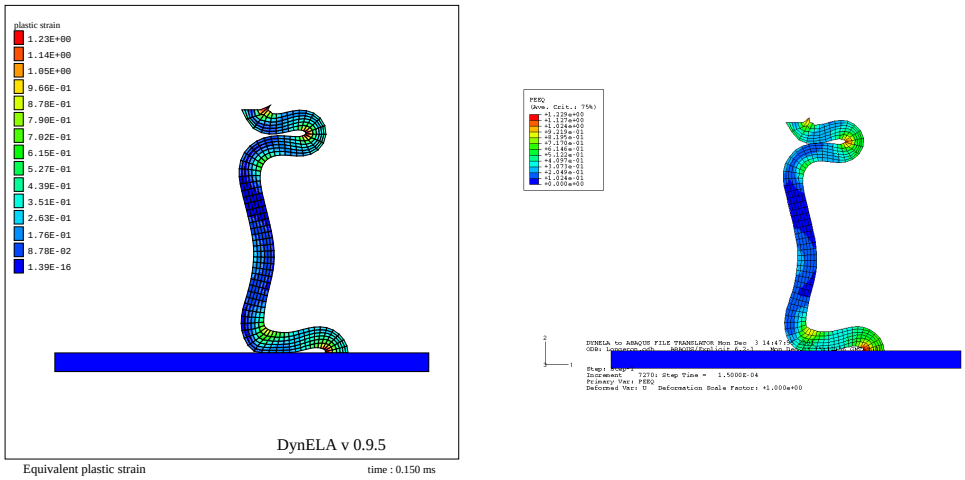

Figure 5: Equivalent plastic strain: DynELA (left) and Abaqus/Explicit (right)

5 Conclusion

An object-oriented simulator was developed for the analysis of large inelastic deformations and impact processes. Only one example has been presented, but many other have currently being tested to ensure the accuracy of the developed algorithms. Some of the benefits of using an OOP approach in comparison with traditional programming language were proposed in this presentation 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.

Current developments of this FEM code concerns the ability to use a multigrid resolution algorithm. To do so, we are currently adding new features in the linear algebra library to include sparse matrix, various preconditioners and iterative solving methods such as the Conjugate Gradient, the BiConjugate Gradient and other iterative methods.

References

[1] B. Stroustrup. *The C++ programming language*. Addison Wesley, second edition, 1991.

[2] J. T. Cross, I. Masters, and R. W. Lewis. Why you should consider object-oriented programming techniques for finite element methods. *International Journal of Numerical Methods for Heat and Fluid Flow*, 9:333–347, 1999.

[3] J. C. Simo and T. J. R. Hughes. *Computational inelasticity.* Springer, 1998.

[4] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. math. Software*, 5:308–329, 1979.

[5] S. W. Haney. Is c++ fast enough for scientific computing ? *Computers in Physics*, 8(6):690, Nov/Dec 1994.

[6] S. Haney and J. Crotinger. How templates enables high-performance scientific computing in c++. *Computing in Science and Engineering*, pages 66–72, jul/aug 1999.

[7] N Zabaras, Y. Bao, A. Srikanth, and W. G. Frazier. A continuum lagrangian sensitivity analysis for metal forming processes with applications to die design problems. *International Journal for Numerical Methods in Engineering*, 48:679–720, 2000.

[8] G. R. Miller. An object oriented approach to structural analysis and design. *Computers and Structures*, 40(1):75–82, 1991.

[9] R. I. Mackie. Object oriented programming of the finite element method. *International Journal for Numerical Methods in Engineering*, 35:425–436, 1992.

[10] O. Pantalé, R. Rakotomalala, and M. Touratier. An ale three-dimensional model of orthogonal and oblique metal cutting processes. *International Journal of Forming Processes*, 1(3):371–388, 09 1998.

[11] O. Pantalé. *User manual of the finite element code DynELA v. 0. 9. 5.* Laboratoire LGP ENI Tarbes, Av d'Azereix 65016 Tarbes, France, 2001.