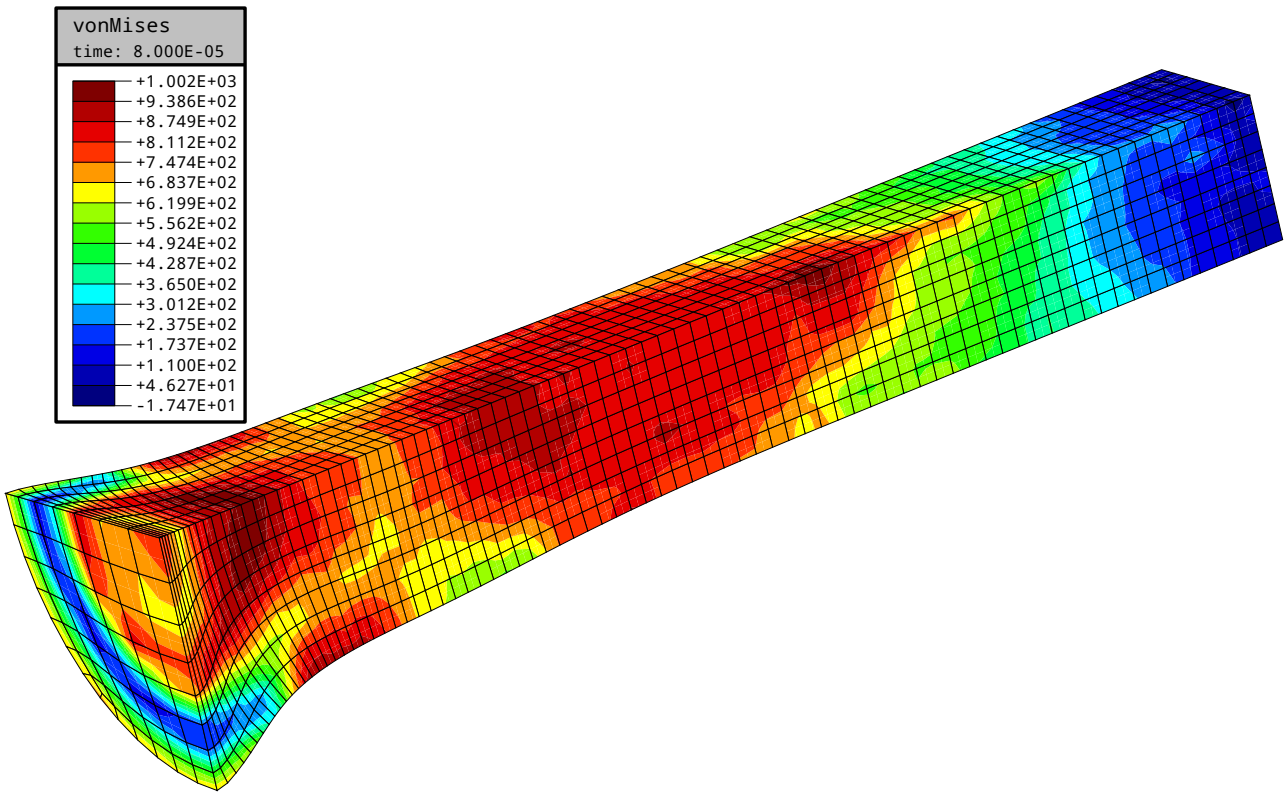
DynELA Finite Element Code

v.4.0

Olivier PANTALE



Laboratoire Génie de Production – Equipe M2SP
- 2020 -

Contents

Notations 1

1 Installation of the DynELA Finite Element Code 5
1.1 Prerequisites . . . 5
1.2 Download and compilation . . . 6
1.3 Testing and usage . . . 6

I DynELA FEM code theory 9

II DynELA Programming 11

1 DynELA programming language 13
1.1 Introduction and basic knowledge . . . 13
1.1.1 Calling the python interpreter . . . 13
1.1.2 Formalism of a DynELA python file . . . 14
1.2 The Kernel library . . . 15
1.3 The Maths library . . . 15
1.4 Model, Nodes and Elements . . . 15
1.4.1 Model . . . 16
1.4.2 Nodes . . . 16
1.4.3 Elements . . . 17
1.4.4 Coordinates transformations . . . 18
1.5 Materials . . . 19
1.5.1 Declaration of materials . . . 19
1.5.2 Johnson-Cook constitutive law . . . 20
1.6 Boundaries conditions . . . 21

1.6.1 Restrain boundary condition . . . 21
1.6.2 Amplitude . . . 21
1.6.3 Constant speed . . . 21
1.6.4 Initial speed . . . 21
1.7 Fields . . . 21
1.7.1 Nodal fields . . . 21
1.7.2 Element fields . . . 22
1.7.3 Global fields . . . 23
1.8 Data Output during computation . . . 23
1.8.1 VTK Data files . . . 23
1.8.2 History files . . . 24
1.9 Solvers . . . 25
1.9.1 Parallel solver . . . 26
1.9.2 Solving procedure . . . 26
1.10 Vectorial SVG meshes and contourplots . . . 26
1.10.1 Initialization of an SVG object . . . 27
1.10.2 Drawing options . . . 28
1.10.3 Creation of the SVG file . . . 29

2 Curves utility **31**
2.1 Introduction and presentation of the script . . . 31
2.1.1 Usage of the Python script . . . 31
2.1.2 Datafile format . . . 31
2.2 Configuration file to define the plots . . . 32
2.2.1 Global parameters for a graph . . . 33
2.2.2 Plotting curve parameters . . . 33
2.2.3 Legend definition . . . 34

3 Abaqus extractor utility **35**
3.1 Introduction and presentation of the script . . . 35
3.2 Syntax of the configuration file . . . 35

III DynELA Samples **37**

1 DynELA single element sample cases **39**
1.1 Uniaxial tensile tests . . . 40
1.1.1 Element plane tensile test . . . 40

1.1.2 Element 3D tensile test 42
1.1.3 Element radial tensile test 42
1.1.4 Element radial torus test 46
1.2 Uniaxial one element shear test 49
1.2.1 Elastic case 49
1.2.2 Johnson-Cook plastic behaviour 51

2 DynELA plasticity sample cases **53**
2.1 Necking of a circular bar 54
2.1.1 Axisymmetric Bar Necking 54

3 DynELA impact sample cases **57**
3.1 Taylor impact sample 58
3.1.1 Axisymmetric Taylor impact 58
3.1.2 3D Taylor impact 59

Bibliography **63**

List of Figures

II.1.1 Export of a CSV history file 24
II.1.2 Export of a mesh in SVG format 27
II.1.3 Export of a von Mises contourplot in SVG format 27

III.1.1 Numerical model for the one element tensile test 40
III.1.2 Comparison of numerical and analytical results for the one element tensile test . . . 41
III.1.3 Numerical model for the one element tensile test 42
III.1.4 Comparison of numerical and analytical results for the 3D one element tensile test 44
III.1.5 Comparison of numerical and analytical results for the one element radial tensile test 45
III.1.6 Numerical model for the one element torus test 46
III.1.7 Comparison of the right edge displacement for the one element torus test 46
III.1.8 Numerical model for the one element torus test under Abaqus 47
III.1.9 Comparison of numerical results for the one element torus tensile test 48
III.1.10 Numerical model for the one element shear test 49
III.1.11 Comparison of numerical and analytical results for the one element elastic shear test 50
III.1.12 Comparison of numerical and analytical results for the one element shear test . . . 51

III.2.1 Numerical model for the necking of a circular bar 54
III.2.2 Temperature T contourplot for the necking of a circular bar (DynELA left and Abaqus right) 55
III.2.3 von Mises stress $\overline{\sigma}$ contourplot for the necking of a circular bar (DynELA left and Abaqus right) 55
III.2.4 Comparison of numerical and analytical results necking of a circular bar test 56

III.3.1 Numerical model for the Axisymmetric Taylor impact test 58
III.3.2 Temperature contourplots for Axisymmetric Taylor impact test (DynELA left and Abaqus right) 59
III.3.3 Comparison of numerical and analytical results for the Axisymmetric Taylor impact test 60

III.3.4 Numerical model for the 3D Taylor impact test 61
III.3.5 Temperature contourplot for the 3D Taylor impact test 61
III.3.6 Comparison of numerical and analytical results for the 3D Taylor impact test 62

List of Tables

II.1.1 General properties of materials 19
II.1.2 Nodal fields 22
II.1.3 Element fields 23
II.1.4 Global fields 23

III.1.1 Material parameters of the Johnson-Cook behavior for the numerical tests 39

III.2.1 Material parameters of the Johnson-Cook behavior for the numerical tests 53
III.2.2 Comparison of numerical results for the necking of a circular bar test 54

III.3.1 Material parameters of the Johnson-Cook behavior for the numerical tests 57
III.3.2 Comparison of numerical results for the Axisymmetric Taylor impact test 58
III.3.3 Comparison of numerical results for the 3D Taylor impact test 60

Notations

From a general point of view, it is usual to observe that one of the main difficulties in the field of mechanics, as in other fields, is the non-homogeneity of notations between the various authors. It is then easy to make completely incomprehensible the slightest theory when one decides to change notation. As the notion of universal notation is not yet valid (even if certain conventions can be assimilated to universal concepts), then we present below the set of notations used throughout this document and in a broader way in all the other documents in that series.

Notations Conventions

a	Scalar
$\vec{a}$	Vector
A	2^{nd} order Tensor or matrix
$\mathcal{A}$	3^{rd} order Tensor
$\mathbb{A}$	4^{th} order Tensor

Linear Algebra and Mathematical Operators

$\vec{a} \cdot \vec{b}$	Dot product of the vectors $\vec{a}$ and $\vec{b}$
$\vec{a} \otimes \vec{b}$	Tensor (or Dyadic) product of the vectors $\vec{a}$ and $\vec{b}$
$\vec{a} \wedge \vec{b}$	Vectorial product of the vectors $\vec{a}$ and $\vec{b}$
A : B	Double contracted product of the two tensors A et B
$\dot{\square}$	Time derivative of quantity $\square$
$\ddot{\square}$	Second order time derivative of quantity $\square$
$\square_{,\square}$	Partial derivative of quantity $\square$ with respect to $\square$
$\square^T$	Transpose of a matrix or a vector $\square$
tr $\square$	Trace of a matrix or a tensor $\square$ (tr $\square = \sum \square_{ii}$)
dev $\square$	Deviatoric part of a tensor $\square$ (dev $\square = \square - \frac{1}{3}$ tr $\square$ 1)
δ_{ij}	Kronecker delta identity
1	Unity matrix or second order tensor
$\mathbb{I}$	Unity fourth order tensor

Basic Continuum Mechanics

$\vec{x} = \begin{bmatrix} x & y & z \end{bmatrix}^T$	Coordinates in the physical domain
$\vec{u} = \begin{bmatrix} u & v & w \end{bmatrix}^T$	Displacement field

$\vec{\omega} = \begin{bmatrix} \omega_x & \omega_y & \omega_z \end{bmatrix}^T$	Rotation field
$\mathbf{\Omega}$	Arbitrary body in the current configuration
$\mathbf{\Gamma}$	Boundary of an arbitrary body $\mathbf{\Omega}$ in the current configuration
ρ	Material density
E	Young's modulus of a material
ν	Poisson's ratio of a material
K	Bulk modulus of a material
λ	Lamé's first parameter of a material
$\mu = G$	Lamé's second parameter / Coulomb's shear modulus
$\vec{F}$	External load vector
$\vec{f}$	External load vector
$\boldsymbol{\varepsilon}$	Green-Lagrange strain tensor
$\boldsymbol{\sigma}$	Cauchy stress tensor
s	Deviatoric part of the Cauchy stress tensor
$\boldsymbol{\alpha}$	Backstress tensor
$\boldsymbol{\phi}$	$\boldsymbol{\phi} = \mathsf{s} - \boldsymbol{\alpha}$

Constitutive laws

f	
n	Direction of the plastic flow
q	Heredity variables in an elastoplastic behavior
$\overline{\sigma}$	von Mises equivalent stress
$\overline{\varepsilon}^p$	Equivalent plastic strain
$\dot{\overline{\varepsilon}}^p$	Equivalent plastic strain rate
Λ	Norm of the plastic strain
σ^v	
σ_0^v	
σ_∞^v	

Large Deformations

$\vec{X} = \begin{bmatrix} X & Y & Z \end{bmatrix}^T$	Coordinates in the reference domain
E	Green-Lagrange deformation tensor
F	Deformation gradient tensor
U, V	Right and left pure deformation tensors
R	Rotation tensor
L	Deformation speed tensor
D	Symmetric part of the L tensor
W	Skew-symmetric part of the L tensor

Finite Element Data Structures

N	Shape functions matrix
$\vec{\xi} = \begin{bmatrix} \xi & \eta & \zeta \end{bmatrix}^T$	Coordinates in the parent domain
B	Derivatives of the shape functions
$\square^e$	Quantity $\square$ related to element e
J	Jacobian matrix

M	Mass matrix
K	Stiffness matrix
$\overline{\mathsf{F}}$	External surfacic load vector
F	External load vector
$\overline{\mathsf{f}}$	External volumic load vector
q	Nodal unknowns vector
n_g	Number of nodes of the current element
n_Q	Number of integration points of the current element

Chapter 1

Installation of the DynELA Finite Element Code

1.1 Prerequisites 5
1.2 Download and compilation 6
1.3 Testing and usage 6

The DynELA Finite Element Code is an Explicit FEM code written in C++ using a Python's interface for creating the Finite Element Models. This is a new version of the early proposed v.2 code written between 1996 and 2010. The previous version has been included into the CAE Linux distribution some year ago and the corresponding work has been published in some Scientific Journals [1–7] , some international conferences [8–13] and served to support for some Ph.D. theses [14, 15] . The aim of v.4.0 is to provide an enhanced version of the code with enhancements concerning the constitutive laws, a new programming interface based on Python 3 formalism, along with some enhanced documentation. The DynELA Finite Element Code is developed under Linux (an ubuntu 20.04 LTS is currently used for the development). All source code and material can be downloaded from the following gitlab website :

https://git.enit.fr/opantale/dynela-v.-4.0.git

The DynELA Finite Element Code is licensed under BSD-3-Clause license [(1)].

1.1 Prerequisites

Compilation of the DynELA Finite Element Code requires several libraries. Generation of Makefiles for the compilation of the DynELA Finite Element Code is based on the use of the CMake tool. CMake is a cross-platform, open-source build system generator. Under ubuntu it can be installed with the following command:

```
sudo apt install cmake
```

DynELA is written in C++ and Python 3 therefore it needs a C++ compiler and some Python 3 libraries. Under ubuntu those libraries can be installed with the following command:

[(1)] See sources of information on Internet if you don't know what it means.

```
sudo apt install build-essential swig zlib1g-dev liblapacke-dev python3-dev
```

It also needs some Python 3 modules to run properly and at least numpy and matplotlib:

```
sudo apt install python3-numpy python3-matplotlib texlive dvipng \
texlive-latex-extra texlive-fonts-recommended
```

1.2 Download and compilation

Downloading of the source code from the github repository, compilation and installation of the software into a sub-directory can be done using the following procedure:

```
git clone https://git.enit.fr/opantale/dynela-v.-4.0.git
cd DynELA
mkdir Build
cd Build
cmake ../Sources
make
```

After downloading and compilation, there is no need to install the executable or something similar to use the FEM code. You just have to modify the .bashrc file and add the following lines where *path_to_DynELA* points to the top directory of your DynELA Finite Element Code installation:

```
export DYNELA="path_to_DynELA"
export PATH=$PATH:$DYNELA/bin
export DYNELA_BIN=$DYNELA/Build/bin
export DYNELA_LIB=$DYNELA/Build/lib
export PYTHONPATH="$DYNELA_BIN:$PYTHONPATH"
export PYTHONPATH="$DYNELA_LIB:$PYTHONPATH"
export LD_LIBRARY_PATH=$DYNELA_LIB:$LD_LIBRARY_PATH
```

1.3 Testing and usage

Testing of the installation can be done by running one of the provided samples. All samples of the DynELA Finite Element Code are located into the sub-directories of the Samples folder. Running a simulation is done using the following command in one of the Samples sub-directories:

```
python sample.py
```

Running the tests in the Samples directories can also be done with regard to the Makefiles contained in the Samples directories. Benchmark tests can be run from any sub-directory of the Sample folder using the following command:

```
make
```

The DynELA Finite Element Code now has a class for direct export of contourplot results using SVG vectorial format for a 2D or 3D mesh and time-history curves through the Python command interface. See the documentation for all instructions concerning SVG and time-history outputs and the examples included in the Samples directories.

The DynELA FEM code can generate VTK files for the results. The Paraview postprocessor can be used to visualize those results. Paraview is available here:

https://www.paraview.org

Part I

DynELA FEM code theory

Part II

DynELA Programming

Chapter 1

DynELA programming language

1.1 Introduction and basic knowledge 13
1.2 The Kernel library 15
1.3 The Maths library 15
1.4 Model, Nodes and Elements 15
1.5 Materials 19
1.6 Boundaries conditions 21
1.7 Fields 21
1.8 Data Output during computation 23
1.9 Solvers 25
1.10 Vectorial SVG meshes and contourplots 26

This chapter deals about the DynELA Finite Element Code programming language. This language is based on Python 3 and all models must be described using this formalism. Therefore, this chapter will describe step by step how to build a Finite Element Model for the DynELA Finite Element Code, using the Python 3 language. To set-up a model in DynELA, a good knowledge of the Python's 3 language is not mandatory, but it will help to be skillful concerning this language. On the other hand advanced features can be setup with the help of Python 3 language.

1.1 Introduction and basic knowledge

The Python 3 language serves as a support for describing FEM in DynELA. Construction of the Finite Element Model is done by calling some C++ methods of the DynELA library through Python language. In fact, the C++ methods of the DynELA library are encapsulated by the help of SWIG. Some methods of the DynELA library have therefore been encapsulated using the SWIG utility and are directly accessible from the Python language.

1.1.1 Calling the python interpreter

After the installation and compilation phase of the code[(1)], the DynELA Finite Element Code can be used using the following command:

[(1)] See the installation instructions in chapter 1 of the preamble, page 5

```
1 python model.py
```

where model.py is the Python 3 source file defining the Finite Element Model. The model.py file contains the definition of the Finite Element Model using a Python 3 language and calls to specific DynELA methods written in C++. The formalism used to set up this Python source file are described is the subsequent chapters.

1.1.2 Formalism of a DynELA python file

To build a Finite Element Model, it is mandatory to import the dnlPython interpreter from the .py script. Conforming to this formalism, we give hereafter the minimal piece of Python code to set up a Finite Element Model in the DynELA Finite Element Code.

```
1 #!/usr/bin/env python3
2 import dnlPython as dnl # Imports the dnlPython library as dnl
3 model = dnl.DynELA()    # Creates the main Object
4 ...                     # Set of instructions to build the FE model
5 ...                     # conforming to the DynELA language and Python 3
6 model.solve()           # Runs the solver
7 ...                     # Set of instructions to postprocess the FE model
```

In the preceding piece of code, line 2 is used to load into the namespace dnl the dnlPython module containing the interface to all C++ methods of the DynELA Finite Element Code, based on the use of the SWIG Python interface. Of course, and conforming to the import command, any name can be used as a namespace to import the DynELA top library. Therefore, all public methods of the DynELA Finite Element Code written in C++ can be called from the Python script to build the Finite Element Model, launch the solver, produce output results,...

In the proposed piece of code, line 3 is used to create an object of type DynELA (the higher object type in the DynELA Finite Element Code library) and instantiate it as the model object[(2)], while line 6, the solver of the DynELA Finite Element Code library is called to solve the problem and produce the results.

It is also possible, but not recommended, to use the following syntax for lines 2 and 3 of the previous block:

```
1 #!/usr/bin/env python3
2 from dnlPython import * # Imports the dnlPython library in current namespace
3 model = DynELA()        # Creates the main Object
4 ...                     # Rest of the source code defining the FE model
```

and then access all top methods and object of the DynELA library directly without referring to the namespace, such as what is proposed in line 3 of the previous block.

As the interpreter of the DynELA Finite Element Code is based on Python 3 language, we can use all instructions valid in Python 3 along with the specific DynELA instructions. In the rest of this documentation, we assume that the notions of programming in Python are mastered, and we will focus only on the functions specific to the DynELA Finite Element Code.

[(2)] For the rest of this chapter, we assume that the name of the instantiated DynELA object is model.

1.2 The Kernel library

#include 'LogFile.h'
#include 'MacAddress.h'
#include 'Settings.h'
#include 'String.h'
#include 'System.h'
#include 'Timer.h'
#include 'Field.h'

1.3 The Maths library

#include 'DiscreteFunction.h'
#include 'DiscreteFunctionSet.h'
#include 'Function.h'
#include 'Matrices.h'
#include 'Matrix.h'
#include 'MatrixDiag.h'
#include 'PolynomialFunction.h'
#include 'RampFunction.h'
#include 'SinusFunction.h'
#include 'SymTensor2.h'
#include 'Tensor2.h'
#include 'Tensor3.h'
#include 'Tensor4.h'
#include 'Vec3D.h'
#include 'Vector.h'
#include 'ColorMap.h'

1.4 Model, Nodes and Elements

All Finite Element Models involves nodes and elements. The very first part of the model is therefore to create the nodes and the elements of the structure to set up a Finite Element Model. The DynELA Finite Element Code library doesn't include any meshing procedure yet, therefore, it is mandatory to create all elements and all nodes by hand or using Python loops in case it can be used.

Another way is to use an external meshing program and convert the output of this program to produce the ad hoc lines of Python to describe the elements and the nodes of the model. This has been used many times by the author, and the Abaqus Finite Element code is an efficient way to create the mesh using the .inp text file generated by the CAE Abaqus program and then convert the nodes and elements description parts of the Abaqus inp file into DynELA formalism.

1.4.1 Model

Definition of a model in the DynELA Finite Element Code is done by creating an instance of the DynELA object into memory. This is done by calling the dnlPython.DynELA(string) method that returns an object of type DynELA as presented hereafter.

```
1 import dnlPython as dnl        # Imports the dnlPython library as dnl
2 model = dnl.DynELA('Taylor')   # Creates the main Object model named Taylor
```

In line 2 of the preceding piece of code, a reference name[3] Taylor is associated to the model object during creation. Once the model is created, one can then define all nodes, elements, materials, constitutive laws, boundary conditions,...

1.4.2 Nodes

1.4.2.1 Definition of the nodes

In the DynELA Finite Element Code, creation of nodes is done by calling the DynELA::createNode() method. Therefore, a node is created by calling the createNode() method and giving the new node number and the x, y and z coordinates of the new node as presented just below.

```
1 model.createNode(1, 0.0, 0.0, 0.0)  # Creates node 1, coordinates [0.0, 0.0, 0.0]
2 model.createNode(2, 1.0, 2.0, -1.0) # Creates node 2, coordinates [1.0, 2.0, -1.0]
```

An alternative method can be used if the coordinates of the node are already stored into a Vec3D object as presented hereafter.

```
1 vect = dnl.Vec3D(1.0, 2.0, -1.0) # Creates a Vec3D object [1.0, 2.0, -1.0]
2 model.createNode(1, vect)        # Creates node 1 with coordinates vect
```

A check of the total number of nodes of the structure can be done using the DynELA.getNodesNumber() method that returns the total number of nodes created.

1.4.2.2 Definition of the Nodes sets

Manipulation of nodes, application of boundaries conditions, etc,...is done through the definition of nodes sets. Such nodes sets are used to group nodes under a NodeSet object for further use. A NodeSet object contains a reference name and a list of nodes. Creation of a NodeSet is done using the DynELA.NodeSet() method that returns an new NodeSet instance. The NodeSet can be named during the creation by specifying its name as a string.

```
1 nset = dnl.NodeSet('NS_All')
```

[3]The reference name is a string used to identify the object, this is completely optional but useful for debugging purposes for example as one can know the associated name to an object.

When the NodeSet has been created, one can now define the list of nodes constituting the NodeSet with the generic DynELA.add() method with the following formalism:

DynELA.add(nodeset, start, end, increment)

Hereafter is some self explaining examples to illustrate this process.

```
nset = dnl.NodeSet('NS_All')
model.add(nset, 2)       # Add node number 2 to node set
model.add(nset, 1, 4)    # Add nodes number 1-4 to node set
model.add(nset, 1, 4, 2) # Add nodes number 1 and 3 to node set
```

1.4.3 Elements

1.4.3.1 Definition of the elements

Creation of elements is done by calling the DynELA.createElement() method. An element is created by calling the createNode() method and giving the new element number and the list of nodes defining the element shape separated by comas and ordered thanks to the element definition as presented just hereafter.

DynELA.createElement(elementNumber, node1, node2,...)

Before creating the very first element of the structure, it is necessary to define the element shape using the DynELA.setDefaultElement() method. An example of element creation combining the two preceding methods is presented hereafter.

```
model.setDefaultElement(dnl.Element.ElQua4N2D) # Defines the default element
model.createElement(1, 1, 2, 3, 4)             # Creates element 1 with nodes 1,2,3,4
```

The following elements are available in the DynELA Finite Element Code.

ElQua4n2D : 4 nodes bi-linear 2D quadrilateral element.

ElQua4NAx : 4 nodes bi-linear axisymmetric quadrilateral element.

ElTri3N2D : 3 nodes 2D triangular element.

ElHex8N3D : 8 nodes 3D hexahedral element.

ElTet4N3D : 4 nodes 3D tetrahedral element.

ElTet10N3D : 10 nodes 3D tetrahedral element.

The total number of elements of the structure can be checked using the DynELA.getElementsNumber() method that returns the total number of elements created.

1.4.3.2 Definition of the Element sets

Declaration of materials, boundaries conditions, etc...is done through the definition of elements sets. Such elements sets are used to group elements under an ElementSet object for further use. An ElementSet object contains a reference to a name and a list of elements. Creation of an ElementSet is done using the DynELA.ElementSet() method that returns a new ElementSet instance. The ElementSet can be named during the creation by specifying its name as a string.

```
eset = dnl.ElementSet('ES_All')
```

When the ElementSet has been created, one can now define the list of elements constituting the ElementSet with the generic DynELA.add() method according to the following formalism:

DynELA.add(elementset, start, end, increment)

Hereafter is some self explaining examples to illustrate this process.

```
eset = dnl.ElementSet('ES_All')
model.add(eset, 2)       # Add element number 2 to element set
model.add(eset, 1, 4)    # Add elements number 1-4 to element set
model.add(eset, 1, 4, 2) # Add elements number 1 and 3 to element set
```

1.4.4 Coordinates transformations

When the mesh has been created, it is always possible to modify the geometry of the structure by applying some geometrical operations such as translations, rotations and change of scale. Those operations apply on a NodeSet.

1.4.4.1 Translations

One can define a translation of the whole model or a part of the model by defining a translation vector (an instance of the DynELA Finite Element Code Vec3D) and apply this translation to the whole structure (without specifying the NodeSet) or a NodeSet using the DynELA.translate() method with the following syntax.

```
vector = dnl.Vec3D(1, 0, 0)   # Defines the translation vector
model.translate(vector)       # Translates the whole model along [1, 0, 0]
model.translate(vector, nset) # Translates the NodeSet nset along [1, 0, 0]
```

1.4.4.2 Rotations

One can define a rotation of the whole model or a part of the model by defining a rotation vector (global axes $\vec{x}$, $\vec{y}$, $\vec{z}$ or an instance of the DynELA Finite Element Code Vec3D) and an angle α then apply this rotation to the whole structure (without specifying the NodeSet) or a NodeSet using the DynELA.rotate() method with the following syntax.

```
model.rotate('X', angle)        # Rotation of the whole structure around X
model.rotate('X', angle, nset)  # Rotation of NodeSet nset around X
axis = dnl.Vec3D(1.0, 1.0, 1.0) # Defines the axis of rotation
model.rotate(axis, angle)       # Rotation of the whole structure around axis
model.rotate(axis, angle, nset) # Rotation of NodeSet nset around axis
```

1.4.4.3 Scaling

One can define a scaling of the whole model or a part of the model by defining a scale factor or a scale vector (an instance of the DynELA Finite Element Code Vec3D) and apply this scaling operation to the whole structure (without specifying the NodeSet) or a NodeSet using the DynELA.scale() method with the following syntax.

```
model.scale(value)              # Scales the whole structure by factor value
model.scale(value, nset         # Scales the NodeSet nset by factor value
vec = dnl.Vec3D(2.0, 1.0, 1.0) # Defines the scale vector
model.scale(vec)                # Scales the whole structure by a factor of 2.0 on x
model.scale(vec, nset)          # Scales the NodeSet nset by a factor of 2.0 on x
```

1.5 Materials

1.5.1 Declaration of materials

1.5.1.1 Material declaration

Creation of a Material is done using the DynELA.Material() method. It is possible to give a name to a material during the creation process by specifying it through a string during the declaration. This can be used further.

```
# Creates the material
steel = dnl.Material('Steel')
```

1.5.1.2 General properties of materials

General properties of materials in DynELA Finite Element Code concerns the general constants such as Young's modulus, Poisson's ratio, density,...The complete list of parameters is reported in Table II.1.1. After creating an instance of the object dnl.Material, on can apply the prescribed values to all

Name	Symbol	Unit	Description
youngModulus	E	MPa	Young modulus
poissonRatio	ν		Poisson ratio
density	ρ	kg/m^3	Density
heatCapacity	C_p	$J/°C$	Heat capacity
taylorQuinney	η		Taylor-Quinney coefficient
initialTemperature	T_0	$°C$	Initial temperature

***Table II.1.1** – General properties of materials*

those parameters using the following syntax.

```
# Creates the material
steel = dnl.Material('Steel')
# Apply all parameters
steel.youngModulus = 206e9
steel.poissonRatio = 0.3
steel.density = 7830
steel.heatCapacity = 46
steel.taylorQuinney = 0.9
steel.initialTemperature = 25
```

1.5.1.3 Material affectation to a set of elements

And, the material can be affected to the elements of the model by the DynELA.add() method as proposed hereafter.

```
# Creates the material
steel = dnl.Material('Steel')
# Apply all parameters
...
# Affect the material to the element set eset
model.add(steel, eset)
```

1.5.2 Johnson-Cook constitutive law

The Johnson-Cook constitutive law is an hardening law defining the yield stress $\sigma^y(\overline{\varepsilon}^p, \dot{\overline{\varepsilon}}^p, T)$ by the following equation:

$$\sigma^y = \left(A + B\overline{\varepsilon}^{p^n}\right)\left[1 + C\ln\left(\frac{\dot{\overline{\varepsilon}}^p}{\dot{\overline{\varepsilon}}_0}\right)\right]\left[1 - \left(\frac{T - T_0}{T_m - T_0}\right)^m\right] \tag{1.1}$$

where $\dot{\overline{\varepsilon}}_0$ is the reference strain rate, T_0 and T_m are the reference temperature and the melting temperature of the material respectively and A, B, C, n and m are the five constitutive flow law parameters. Therefore, this kind of hardening law can be defined by using the following piece of code:

```
hardLaw = dnl.JohnsonCookLaw()                          # Hardening law
hardLaw.setParameters(A, B, C, n, m, depsp0, Tm, T0) # Parameters of the law
```

Once the hardening law has been created, one have to link this hardening law to a material already defined using the following piece of code:

```
# Creates the material
steel = dnl.Material('Steel')
# Creates the hardening law
hardLaw = dnl.JohnsonCookLaw()
# Attach hardening law to material
steel.setHardeningLaw(hardLaw)
```

1.6 Boundaries conditions

1.6.1 Restrain boundary condition

```
# Declaration of a boundary condition for top part
topBC = dnl.BoundaryRestrain('BC_top')
topBC.setValue(0, 1, 1)
model.attachConstantBC(topBC, topNS)
```

1.6.2 Amplitude

```
# Declaration of a ramp function to apply the load
ramp = dnl.RampFunction('constantFunction')
ramp.set(dnl.RampFunction.Constant, 0, stopTime)
```

1.6.3 Constant speed

```
# Declaration of a boundary condition for top part
topSpeed = dnl.BoundarySpeed()
topSpeed.setValue(displacement, 0, 0)
topSpeed.setFunction(ramp)
model.attachConstantBC(topSpeed, topNS)
```

1.6.4 Initial speed

```
# Declaration of a ramp function to apply the load
ramp = dnl.RampFunction('constantFunction')
ramp.set(dnl.RampFunction.Constant, 0, stopTime)
```

1.7 Fields

1.7.1 Nodal fields

Nodal fields are defined at nodes and cover types defined in table II.1.2. Concerning those fields, some of them are directely defined at nodes, some other are extrapolated from integration points and transfered to nodes as reported in column loc of table II.1.2. Concerning types, scalars, vec3D and tensors are available. Depending in the type of data, different methods can be used to acces those data:

scalar : Direct access to the value as it is unique.

vec3D : Access to all 3 components of a vec3D using nameX, nameY, nameZ or the norm of the vec3D using name.

tensor : Access to all 9 components of a tensor using nameXX, nameXY,..., nameZZ or the norm of the tensor using name.

Name		Type	nb	Loc	Description
density	ρ	scalar	1	IntPt	
displacementIncrement	$\Delta\vec{u}$	vec3D	$3+1$	node	
displacement	$\vec{u}$	vec3D	$3+1$	node	
energyIncrement		scalar	1	IntPt	
energy		scalar	1	IntPt	
gammaCumulate		scalar	1	IntPt	
gamma	Γ	scalar	1	IntPt	
internalEnergy		scalar	1	IntPt	
mass	m	scalar	1	node	
nodeCoordinate	$\vec{x}$	vec3D	$3+1$	node	
normal	$\vec{n}$	vec3D	$3+1$	node	
PlasticStrainInc		tensor	$9+1$	IntPt	
plasticStrainRate		scalar	1	IntPt	
plasticStrain		scalar	1	IntPt	
PlasticStrain		tensor	$9+1$	IntPt	
pressure	p	scalar	1	IntPt	
speedIncrement	$\Delta\vec{v}$	vec3D	$3+1$	node	
speed	$\vec{v}$	vec3D		node	
StrainInc		tensor	$9+1$	IntPt	
Strain		tensor	$9+1$	IntPt	
Stress	$\boldsymbol{\sigma}$	tensor	$9+1$	IntPt	
temperature	T	scalar	1	IntPt	
vonMises	$\overline{\sigma}$	scalar	1	IntPt	
yieldStress	σ^y	scalar	1	IntPt	

***Table II.1.2** – Nodal fields*

1.7.2 Element fields

Element fields are defined at integration points and cover types defined in table II.1.3. Concerning types, scalars, vec3D and tensors are available. Depending in the type of data, different methods can be used to acces those data:

scalar : Direct access to the value as it is unique.

vec3D : Access to all 3 components of a vec3D using nameX, nameY, nameZ or the norm of the vec3D using name.

tensor : Access to all 9 components of a tensor using nameXX, nameXY,..., nameZZ or the norm of the tensor using name.

Name		Type	nb	Description
density	ρ	scalar	1	
gammaCumulate		scalar	1	
gamma	Γ	scalar	1	
internalEnergy		scalar	1	
plasticStrainRate		scalar	1	
plasticStrain		scalar	1	
PlasticStrain		tensor	$9+1$	
PlaticStrainInc		tensor	$9+1$	
pressure	p	scalar	1	
StrainInc		tensor	$9+1$	
Strain		tensor	$9+1$	
Stress	$\boldsymbol{\sigma}$	tensor	$9+1$	
temperature	T	scalar	1	
vonMises	$\overline{\sigma}$	scalar	1	
yieldStress	σ^y	scalar	1	

***Table II.1.3** – Element fields*

1.7.3 Global fields

Name		Type	nb	Description
kineticEnergy	E_c	scalar	1	
realTimeStep	Δt_r	scalar	1	
timeStep	Δt	scalar	1	

***Table II.1.4** – Global fields*

1.8 Data Output during computation

1.8.1 VTK Data files

The DynELA Finite Element Code is able to export results in VTK (The Visualization Toolkit[4]) ASCII files. Those files can be used to visualize the results of computations using the Paraview post-processing software[5].

Those VTK files are automatically created during the solving phase. Controlling the instant of file save is done using the following syntax:

```
1 model.setSaveTimes(startTime, stopTime, incrementTime)
```

[4] See : https://vtk.org
[5] See : https://www.paraview.org

where startTime, stopTime, incrementTime are 3 times defining respectively the times of the first file, the last one and the period of time between subsequent frames.

At any time, it is possible to export a VTK file using the method writeVTKFile( ) of the DynELA class.

1.8.2 History files

During the solving procedure, one can export time history datafiles that can be used to produce time-history plots. This kind of files are text files in CSV format directly exported from the DynELA solver. In order to control the production of those files, one have to create an instance of the HistoryFile class and add this instance to the current model. Those history-files can then be used to produce high-quality plots through the associated Curves Utility of the DynELA Finite Element Code presented in chapter 2, or use the produced data-file in any external plotting application.

As an illustration, the following commented piece of code has been used to produce the CSV history file reported in Figure II.1.1.

```
1 hist = dnl.HistoryFile()              # Constructor of the hist object
2 hist.setFileName('HistoryPlot.plot')  # Sets the filename of the history file
3 hist.add(dnl.Field.timeStep)                  # First column will be timeStep value
4 hist.add(histES, 0, dnl.Field.vonMises)       # Second one will be von Mises stress
5 hist.add(histRad, dnl.Field.nodeCoordinateX) # Third one will be X node coordinate
6 hist.setSaveTime(1e-6)                        # One data every 1E-6 s
7 model.add(hist)                               # Add the time history to the solver
```

```
1 #DynELA FEM Code v. 4.0 history file
2 #plotted :timeStep vonMises nodeCoordinateX
3 0.0000000E+00 6.2463483E-08 0.0000000E+00 3.2000000E+00
4 1.0623148E-06 6.2462797E-08 1.3439840E+03 3.5412151E+00
5 2.0617159E-06 6.2462515E-08 1.3643205E+03 3.7516529E+00
6 3.0611173E-06 6.2462676E-08 1.3748692E+03 3.9635932E+00
7 4.0605220E-06 6.2462901E-08 1.3662499E+03 4.1161636E+00
8 5.0599347E-06 6.2465035E-08 1.3539346E+03 4.2812566E+00
9 .....
```

Figure II.1.1 – *Export of a CSV history file*

In file reported in Figure II.1.1, the first two lines are comments. Line 1 shows the name and version of the DynELA Finite Element Code while line 2 shows the name of the fields from columns 2 to the end of the table (in this example, current time is in column 1, time-step Δt is in column 2, von Mises stress $\overline{\sigma}$ at the center of the element referred by element-set *histES* is in column 3 and nodal coordinate x of the node defined in *histRad* node-set is in the last column of the table). One data is more or less produced at each requested time ($t_i = 10^{-6}s$), but this depends on the time-step increment Δt of the Explicit solver.

1.8.2.1 Definition of data to save

add (*ElementSet, int, field*) : Add the output data *field* defined at integration point *int* for all elements defined in the element-set *ElemenSet* to the current time-history plot.

add (*NodeSet, Field*) : Add the output data *Field* for all nodes defined in the node-set *NodeSet* to the current time-history plot.

add (*Field*) : Add the global *Field* value to the current time-history plot.

setFileName (*string*) : Defines the name of the output CSV filename.

setSaveTime (*float*) : Defines the frequency of data store during the computation.

setSaveTime (*float*[start], *float*[stop], *float*[freq]) : Defines the start, stop and frequency of data store during the computation starting at *start*, ending at *stop* and with the frequency defined by *freq*.

When data storage has been defined, it is mandatory to add the HistoryFile object created to the model to produce the output files during computation. this is done using the following syntax:

```
1 model.add(hist)
```

where *model* is the model to solve and *hist* is the requested history-file object.

1.8.2.2 Check of data information

Data information defined in the previous subsection can be checked using one of the following commands:

getSaveTime ( ) : Returns the frequency of data store.

getStartTime ( ) : Returns the start-time of data store.

getStopTime ( ) : Returns the end-time of data store.

getFileName ( ) : Returns the file name for data store.

1.9 Solvers

The DynELA Finite Element Code currently have only one FEM solver: an Explicit solver based on a Chung-Hulbert algorithm. Declaration of such a solver is done by creating an instance of the Explicit solver class, using the following syntax:

```
1 solver = dnl.Explicit('Solver') # Declaration of the explicit solver
2 solver.setTimes(0, stopTime)    # Definition of start and end time
3 model.add(solver)               # Link solver to the current model
```

In this example, we create an explicit solver named solver at line 1, defines the starting and ending times of the computation at line 2 and attach the created solver to the current FEM model at line 3.

The Explicit class contains some useful methods as reported here after:

setComputeTimeStepFrequency(*int*) : Defines the frequency (*i.e.* every how many increments) of re-computing the critical time step of the model.

setIncrements(*int*, *int*) : Defines the start and stop increments for the solver.

setTimes(*float*, *float*) : Defines start and stop time of the current job.

setTimeStepMethod(*method*) : Defines the method used to compute the time-step which can be one of the three following methods: Courant, PowerIteration or PowerIterationUnder.

setTimeStepSafetyFactor(*float* safetyfactor) : Defines the time-step safety factor (normally within the range $[0:1]$).

1.9.1 Parallel solver

In DynELA Finite Element Code it is possible to control the number of cores to use for the solver through the Parallel class. If no options are given to the parallel solver, it assumes that only 1 core will be used for the computation. There are currently few options in the parallel class and the following ones can be used to control the parallelization of the Explicit solver:

setCores (*int*) : Set the number of cores to use for the solver.

getCores ( ) : Get the number of cores used by the solver.

The control of the number of cores to use in an explicit solve is done using the following syntax, where model refers to the DynELA Finite Element Model.

```
# Parallel solver with two cores
model.parallel.setCores(2)
```

1.9.2 Solving procedure

Running solver is done through the solve method of the DynELA class using the following syntax:

```
# Run the main solver
model.solve()
```

At the beginning of the solving procedure, a call to the initSolve method of the same DynELA class is performed. It is also possible make a explicit call to the initSolve method (*i.e.* separated from the one hidden in the solve method). This is useful if someone wants to modify values of different fields before running the solver for example[(6)].

Commands available for running the solver are the following ones:

initSolve ( ) : Runs the init solve of the DynELA Finite Element Code.

solve ( ) : Runs the solver of the DynELA Finite Element Code.

1.10 Vectorial SVG meshes and contourplots

Since DynELA Finite Element Code version 3.0, it is possible to export SVG files (Scalable Vector Graphics) from the Python command language to produce high quality figures (and of course reproducible results, as everything is generated from script). This can be done before running the Solver to export the initial mesh of the structure for example, or after the solver phase to produce contourplot results. One can then produce some mesh exports as presented in Figure II.1.2 or contourplots exports for a given data (here we used the von Mises field) as presented in Figure II.1.3.

As an illustration, the following commented piece of code has been used to produce both plots presented in Figures II.1.2 and II.1.3.

[(6)] With such an approach, it is possible for example to modify all the fields of all points of the structure and to input the ones coming from another FEM code for example. It is also possible to modify for example only the temperature of all nodes of the structure thanks to an analytical model and run the solver on the basis of those new temperatures, *i.e.* to make an hybrid FEM/analytical simulation.

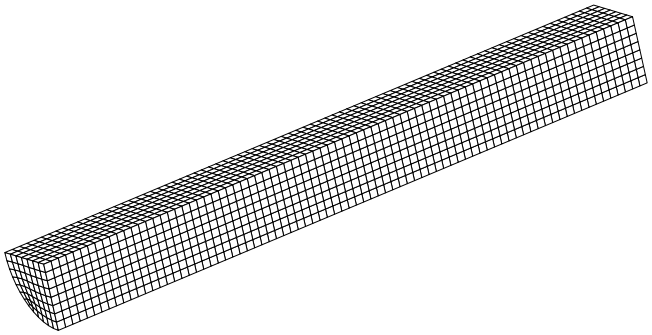


***Figure II.1.2** – Export of a mesh in SVG format*

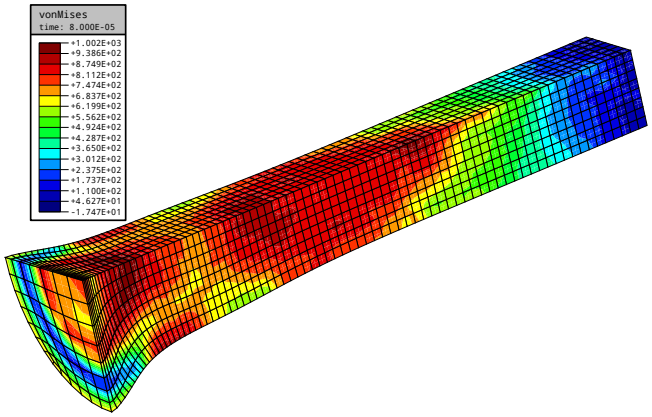


***Figure II.1.3** – Export of a von Mises contourplot in SVG format*

```
1 svg = dnl.SvgInterface('SVG')        # Creates an instance of SvgInterface object
2 svg.setLegendPosition(100, 280)      # Position the legend
3 svg.rotate(dnl.Vec3D(0, 1, 0), 190) # Rotation 190 degree around Y axis
4 svg.rotate(dnl.Vec3D(1, 0, 0), -70) # Rotation -70 degree around X axis
5 svg.rotate('Y', -60)                # Rotation -60 degree around Y axis
6 svg.write('mesh.svg')               # Exports the mesh
7 svg.write('vonMises.svg', dnl.Field.vonMises) # Exports the contourplot
```

More details concerning the syntax of the SvgInterface class are presented in the following subsections.

1.10.1 Initialization of an SVG object

Creation of a SVG file under DynELA Finite Element Code is done using through the instantiation of an object SvgInterface object using the following syntax:

```
1 svg = dnl.SvgInterface('SVG object')
```

In this piece of code, svg is an instance of the object SvgInterface and '*SVG object*' is the associated name of the object created by the constructor of the SvgInterface class.

Concerning SvgInterface, DynELA Finite Element Code creates a window with the prescribed size 1600×1600, where the coordinates of the top-left corner are $(0, 0)$ and the coordinates of the bottom-right corner are $(1600, 1600)$. All drawing options where some coordinates are requested must conform to this definition.

1.10.2 Drawing options

1.10.2.1 View orientation and rotations

resetView ( ) : This resets the current view to the original one where $O, \vec{x}, \vec{y}$ is the drawing plane and $\vec{z}$ is pointing forward.

rotate (*string*, *float*) : Defines the rotation of the global structure around one of the 3 base axis referred by the *string* value 'X', 'Y' or 'Z' with an angle α defined by a *float* value.

rotate (*Vec3D*, *float*) : Defines the rotation of the global structure around the axis $\vec{v}$ defined by a *Vec3D* with an angle α defined by a *float* value.

1.10.2.2 Mesh and data output

setMeshDisplay (*bool* = [true]) : Set or unset the drawing of the mesh.

setMeshThickness (*float* = [1.0]) : Defines the line thickness for drawing the mesh.

setPatchLevel (*int* = [1]) : Defines the level of decomposition of the elements for drawing, *i.e.* the number of recursive decomposition of the polygons [1 to 4]. The default value is usually sufficient for all plots, increasing it enhances the precision of the plot, but drastically increases also the size of the produced file.

setAutoRangeValues (*bool* = [true]) : Set or unset the automatic computation of minimum and maximum values for fields range. This function must be switched to *false* in addition to the function setBounds of the colorMap class (see 1.10.2.4).

1.10.2.3 Legends and notations

setInfoDisplay (*bool* = [false]) : Set or unset the drawing of the bloc of informations.

setInfoPosition (*int*, *int*) : Defines the position of the bloc of informations on the current drawing (default position is $(50, 1200)$).

setLegendDisplay (*bool* = [true]) : Set or unset the drawing of the legend.

setLegendPosition (*int*, *int*) : Defines the position of the legend on the current drawing (default position is $(30, 30)$).

setTitleDisplay (*bool* = [true]) : Set or unset the drawing of the title.

setTitlePosition (*int*, *int*) : Defines the position of the title on the current drawing (default position is $(50, 1550)$).

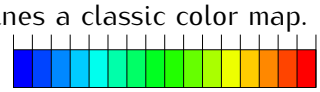
1.10.2.4 Color Map Options

Color Map used for SVG plots can be changed to one of the predefined color-maps using the following syntax (ex: changing for a gray map):

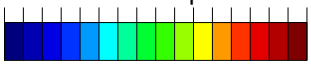
```
1 svg.colorMap.setGrayMap() # set the color map to Gay
```

The following 6 maps are available in DynELA Finite Element Code.

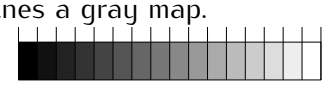
setColorMap ( ) : defines a classic color map.



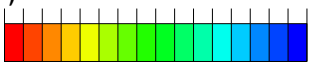
setDeepColorMap ( ) : defines a deep-color map (this is the default map).



setGrayMap ( ) : defines a gray map.

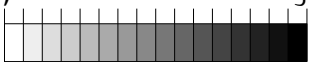


setReverseColorMap ( ) : defines a reversed classic color map (from red to blue).



setReverseDeepColorMap ( ) : defines a reversed deep-color map (from red to blue).

setReverseGrayMap ( ) : defines a reversed gray map (from white to black).



In DynELA Finite Element Code, maps are defined by a given range and number of isovalues. Those parameters can be defined using the following methods:

setLevels (*int* = [16]) : maps are decomposed into the given number of colors, *i.e.* isovalues levels.

setBounds (*float, float*) : maps are automatically ranged to fit the current field range. But this method allows to fix this range[7]. Out of bounds zones will be drawn in black for low values and white for high values.

On the other hand, the following methods returns some informations on the current map:

getMax ( ) : get the maximum value of the range.

getMin ( ) : get the minimum value of the range.

getLevels ( ) : get the current number of levels.

1.10.3 Creation of the SVG file

Effective creation of the SVG file is done by calling the write method of the SvgInterface class with the following syntax:

```
1 svg.write('mesh.svg') # Without field, this draws only the mesh
2 svg.write('temp.svg', dnl.Field.temperature)
```

Here, on line number 1, '*temp.svg*' is the name of the SVG file to create for exporting the mesh. On line number 2, '*temp.svg*' is the name of the SVG file to create and dnl.Field.temperature is the name of the associated field to output (see 1.7 for definitions).

[7] In order to use this method it is mandatory to use the method setAutoRangeValues of the SvgInterface class and set it to *false*.

Chapter 2

Curves utility

2.1 Introduction and presentation of the script 31
2.2 Configuration file to define the plots 32

This chapter deals about the curves utility of the DynELA Finite Element Code. This curve utility is a Python library used to produce plots (vector or bitmap plots) from DynELA Finite Element Code history files by reading a configuration file describing the way to produce those plots from a simple syntax language. The core of the library uses the matplotlib library implemented using the Python3 language. With this curve utility library, it is therefore possible to produce output graphics and curves directly from the python model file at the end of the solve.

2.1 Introduction and presentation of the script

2.1.1 Usage of the Python script

The Curve library is called from a Python script with the following minimal form.

```
1 import dnlCurves              # Import the dnlCurve library
2 curves = dnlCurves.Curves()   # Creates an instance of the Curve object
3 curves.plotFile('Curves.ex')  # Plot the curves defined by the 'Curves.ex' file
```

2.1.2 Datafile format

The datafile format is the plot file used by DynELA Finite Element Code. This file is a text file with $\vec{x}$ and $\vec{y}$ datas defined by two or more series of floating point data on two or more columns separated by spaces. This file should contain two header lines on top and should conform to the following for a datafile containing one data.

```
#DynELA_plot history file
#plotted :timeStep
1.0596474E-06 1.0596474E-06
1.0058495E-04 1.0572496E-06
\ldots\ldots
1.0000000E-02 1.4838518E-07
```

In this file, the very first line is ignored by the script, while the second one is used to define the default name of the variable to be plotted (here 'timeStep' in this case). Here after is a example of a datafile containing more columns.

```
#DynELA_plot history file
#plotted :s11 s22 s33
1.0596474E-06 1.0596474E+06 2.0596474E+06 -1.0596474E+06
1.0058495E-04 1.0572496E+06 2.0572496E+06 -1.0572496E+06
\ldots\ldots
1.0000000E-02 1.4838518E+07 2.4838518E+07 -1.4838518E+07
```

In this case, one can specify which columns will be used for plotting the curves as it will be presented just later.

2.2 Configuration file to define the plots

A configuration file is used to define the plots and is read by the curves.plotFile() method of the Curves object. Global parameters are defined using the keyword Parameters at the beginning of a line of the parameter file. The keyword parameter is followed by a set of commands used to set some global parameters for all the subsequent generated graphs by overwriting the default parameters, before generating the very first graph.

```
Parameters, xname=Displacement x, crop=True
```

Then all parameters can also be altered within the definition of a plot. A definition of a plot is done by a line without the Parameters keyword. The first element on this line is the name of the generated graph, followed by a set of parameters defining the generated plot.

```
# Plot of Temp.plot to Temp.svg file
Temp, name=$T$, Temp.plot
# Plot of column 1 of Stress.plot file to S11.svg file
S11, name=$\sigma_{xx}$, legendlocate=topleft, Stress.plot[0:1]
# Plot of column 2 of Stress.plot file to S22.svg file
S22, name=$\sigma_{yy}$, legendlocate=bottomleft, Stress.plot[0:2]
# Plot of dt.plot and Abaqus/Step.plot files to TimeStep.svg file
TimeStep, name=$\Delta t$, dt.plot, name=$Abaqus \Delta t$, Abaqus/Step.plot
```

In the proposed example we have the following.

- Line 2: plots the content of the Temp.plot file with legend T in a Temp.svg file using default parameters.

- Line 4: plots the content of the Stress.plot file with legend σ_{xx} in a S11.svg file using data from columns 0 for $\vec{x}$ and from column 1 for $\vec{y}$ and with a legend located in the top left part of the figure.
- Line 6: plots the content of the Stress.plot file with legend σ_{yy} in a S22.svg file using data from columns 0 for $\vec{x}$ and from column 2 for $\vec{y}$ with a legend located in the bottom left part of the figure.
- Line 8: plots the content of the dt.plot file with legend Δt and Abaqus/Step.plot file with legend $Abaqus\ \Delta t$ in a TimeStep.svg file.

As presented just before, LaTeX can be used for names and legends by using the escape character $ and remembering that space character has to be defined by the combination of an escape '\' character followed by the space ' ' so that '\ '.

Comments can be inserted as presented briefly earlier by using the '#' character. Everything after the '#' character and up to the end of the line is ignored and treated as comment. Blank lines are also allowed in the file to help human readability of the file.

2.2.1 Global parameters for a graph

outputformat = <'format'> : is used to define the format of the output file, *i.e.* pdf, svg, png,... (default value is svg file).

transparent = <True, False> : is used to define if the background should be or not transparent (default value is False).

crop = <True, False> : is used to define if the output image should be or not cropped (default value is False).

grid = <True, False> : is used to define if the output graph should contain or not a grid (default value is True).

title = <'name'> : defines the title of the graph on the top part of the plot (default value 'Default title of the graph')

xname, yname = <'name'> : defines the names of the $\vec{x}$ and $\vec{y}$ axis for the plot (default value are 'x-axis' and 'y-axis').

titlefontsize = <size> : defines the font size of the title on the top part of the plot (default value is 20).

xfontsize, yfontsize = <size> : defines the font size of the $\vec{x}$ and $\vec{y}$ axis names for the plot (default value is 20).

xlabelfontsize, ylabelfontsize = <size> : defines the font size of the values along the $\vec{x}$ and $\vec{y}$ axis for the plot (default value is 16).

xrange, yrange = <range> : remaps the range of the curves for the $\vec{x}$ and $\vec{y}$ axis to the given value. For example, setting xrange=10 will remap the x-data within the range $[0, 10]$.

xscale, yscale = <range> : multiply the $\vec{x}$ or the $\vec{y}$ data by a given factor. For example, setting xscale=10 will multiply all x-data by a factor of 10.

2.2.2 Plotting curve parameters

name = <'name'> : defines a new name for the next curve in the legend and overrides the one defined by the datafile.

removename = <'name'> : removes from the names of the curves a given pattern for the legends. Example: removename = -S11 will convert 'data-S11' into 'data'. This is not a very useful method and one should prefer the name command to redefine the name of the plotting curve.

linewidth = <'width'> : defines the line width for all subsequent plots (default value is 2).

marks = <'symbol'> : is used to define the next marker to use for all subsequent plots (default value is ''). If the value is void, markers are cycled through the default markers list.

marksnumber = <number> : defines the total number of markers on the curves for all subsequent plots (default value is 20).

markersize = <size> : defines the size of the marker symbols to use for all subsequent plots (default value is 10).

2.2.3 Legend definition

legendcolumns = <columns> : defines the number of columns to use for the legend (default value is 1).

legendshadow = <True, False> : is used to define if the output graph legend should contain or not a shadow box (default value is True).

legendanchor = <'position'> : defines the position of the legend anchor independently from the legend position parameter.

legendposition = <'position'> : defines the position of the legend position independently from the legend anchor parameter.

legendlocate = <'position'> : defines the position of the legend with the following four options: 'topleft', 'topright', 'bottomleft' and 'bottomright' (default value is 'topright').

legendfontsize = <size> : defines the font size of the text in the legend (default value is 16).

Chapter 3

Abaqus extractor utility

3.1 Introduction and presentation of the script 35
3.2 Syntax of the configuration file 35

This chapter deals about the Abaqus extractor utility of the DynELA Finite Element Code. This extractor is a Python library used to extract results from an Abaqus odb datafile by reading a configuration file describing the way to produce those extracts from a simple syntax language.

3.1 Introduction and presentation of the script

The AbaqusExtract library is called from a Python script with the following minimal form.

```
1 # import the AbaqusExtract library
2 import AbaqusExtract
3 # Extract data using datafile Extract.ex
4 AbaqusExtract.AbaqusExtract('Extract.ex')
```

Assuming that the Python script containing this piece of code is named 'Extract.py', therefore this script must be called from the Abaqus main program using the following command:

```
1 abaqus python Extract.py
```

3.2 Syntax of the configuration file

Different forms are used depending on the nature of the data to extract, but everything conforms to the Abaqus Python command syntax. The AbaqusExtract library uses some keywords to define the nature of the data to extract as proposed here after.

TimeHistory : is a keyword defining that the line contains the instructions to extract a time history from an Abaqus odb file.

job : is used to define the name of the odb file to use for the extraction of the results *i.e.* it's the name of the odb file without the extension .odb.

value : is the nature of the variable to extract conforming to the variables names of Abaqus.

name : is the given name of the plotting file to generate, *i.e.* the name of the plotting file without the .plot extension.

region : is used to define the location of the data to extract. This one is more complex, as it can be a global location, a node location or an integration point location as described hereafter.

operate : is an optional parameter defining what to do when multiple regions are defined.

As an example, the following piece of code gives some application examples.

```
1 # Extraction of a time history node data
2 TimeHistory, job=Torus, value=U1, name=dispX, region=Node PART-1.3
3 # Extraction of a time history integration point data
4 TimeHistory, job=Torus, value=MISES, name=vM, region=Element PART-1.25 Int Point 1
5 # Extraction of a time history global variable
6 TimeHistory, job=Torus, value=DT, name=timeStep, region=Assembly ASSEMBLY
```

In the previous example:

- line 2 is used to extract the nodal displacement U1 from the odb file Torus.odb and produce the dispX.plot file for node 3 of the PART-1 piece.
- line 4 is used to extract the integration point value MISES from the odb file Torus.odb and produce the vM.plot file for integration point 1 of element 25 of the PART-1 piece.
- line 6 is used to extract the global timestep DT from the odb file Torus.odb and produce the timeStep.plot file.

Concerning the definition of the region to use, it is possible to define more than 1 region by using the '+' sign in the definition of the region. When this is used, the optional parameter operate is used to define what to do with this multiple data. The operation is defined by:

operate = none : or no operate parameter defined, therefore, all values are reported to the plotting file separated by a white space.

operate = mean : the mean value is computed and reported to the plotting file.

operate = sum : the sum of the values is computed and reported= to the plotting file.

Part III

DynELA Samples

Chapter 1

DynELA single element sample cases

1.1 Uniaxial tensile tests 40
1.2 Uniaxial one element shear test 49

This chapter deals with some numerical applications of the DynELA Finite Element Code for dynamic applications in 2D, axi-symmetric and 3D cases. In the subsequent tests, if not specified, a Johnson-Cook constitutive law is used to model the behavior of the material. The Johnson-Cook hardening flow law is probably the most widely used flow law for the simulation of high strain rate deformation processes taking into account plastic strain, plastic strain rate and temperature effects. Since a lot of efforts have been made in the past to identify the constitutive flow law parameters for many materials, it is implemented in numerous Finite Element codes such as Abaqus [?]. The general formulation of the Johnson-Cook law $\sigma^y(\bar{\varepsilon}^p, \dot{\bar{\varepsilon}}^p, T)$ is given by the following equation:

$$\sigma^y = \left(A + B\bar{\varepsilon}^{p^n}\right)\left[1 + C\ln\left(\frac{\dot{\bar{\varepsilon}}^p}{\dot{\bar{\varepsilon}}_0}\right)\right]\left[1 - \left(\frac{T - T_0}{T_m - T_0}\right)^m\right] \tag{1.1}$$

where $\dot{\bar{\varepsilon}}_0$ is the reference strain rate, T_0 and T_m are the reference temperature and the melting temperature of the material respectively and A, B, C, n and m are the five constitutive flow law parameters. A 42CrMo4 steel following the Johnson-Cook behavior law has been selected for all those tests, and material properties are reported in Table III.3.1.

E (Gpa)	ν	A (MPa)	B (MPa)	C	n	m
206.9	0.3	806	614	0.0089	0.168	1.1
ρ (kg/m^3)	λ $(W/m°C)$	C_p $(J/Kg°C)$	η	$\dot{\bar{\varepsilon}}_0$ (s^{-1})	T_0 $(°C)$	T_m $(°C)$
7830	34.0	460	0.9	1.0	20	1540

***Table III.1.1** – Material parameters of the Johnson-Cook behavior for the numerical tests*

1.1 Uniaxial tensile tests

1.1.1 Element plane tensile test

The uniaxial one element tensile test is a numerical test where an plane element (with a square prescribed shape) is subjected to pure tensile as presented in figure III.1.1. The initial shape of

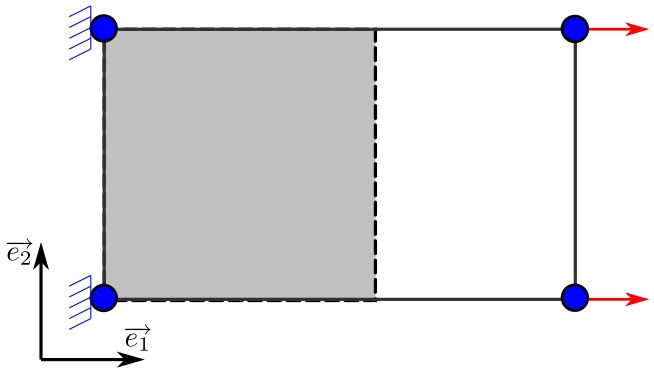


Figure III.1.1 *– Numerical model for the one element tensile test*

the specimen is $10\,mm \times 10\,mm$, the two left nodes of the element are encastred and a prescribed horizontal displacement $d = 10\,mm$ is applied on the two right nodes of the same element as illustrated in Figure III.1.1. As we are using an explicit integration scheme, the total simulation time is set to $t = 0.01\,s$.

All the properties of the constitutive law reported in Table III.3.1 are used and the material is assumed to follow the Johnson-Cook behavior described by equation 3.1.

Figure III.1.2 shows the comparison of the DynELA solver results (plotted in red) and the Abaqus numerical results (plotted in blue) concerning the evolution of the stress components σ_{11}, σ_{22}, σ_{12}, $\overline{\sigma}$, $\overline{\varepsilon}^p$ and T *vs.* the horizontal displacement of the right edge of the specimen along the horizontal axis. For the Abaqus model, the exact same mesh has been used. Comparison of Abaqus and DynELA results is made by averaging the DynELA results on the 4 integration points of the element. This has been done because DynELA Finite Element Code uses full integrated elements while Abaqus have reduced integrated elements only but in fact, the DynELA results at the 4 integration points are the same in this benchmark test.

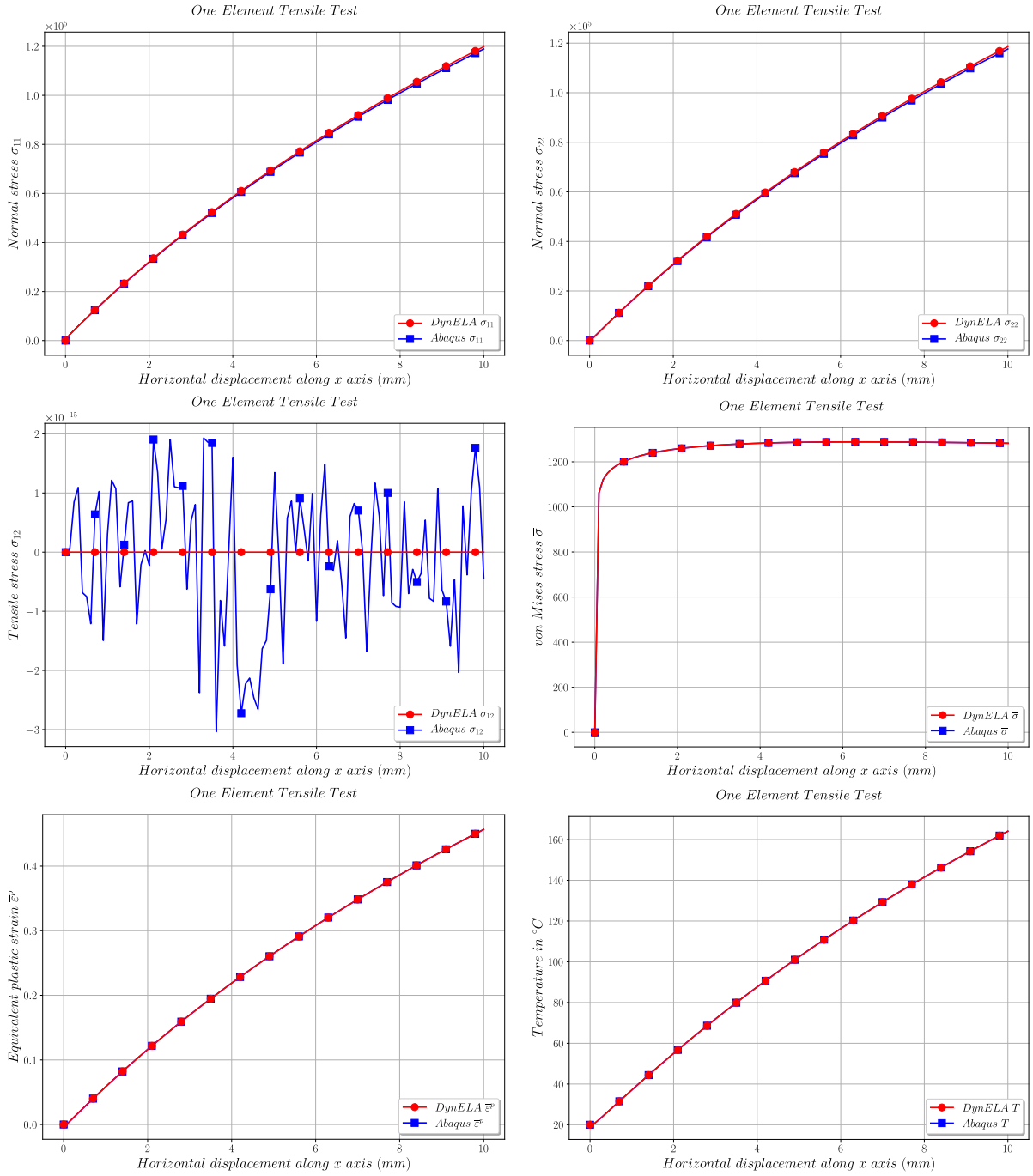


***Figure III.1.2** – Comparison of numerical and analytical results for the one element tensile test*

1.1.2 Element 3D tensile test

The uniaxial one element 3D tensile test is a numerical test where a 3D brick element (with a cubic prescribed shape) is subjected to radial tensile as presented in figure III.1.3. No boundary condition has been prescribed on the $\vec{z}$ direction, so that the reduction of the width during the test can occur. The initial shape of the specimen is $10\,mm \times 10\,mm \times 10\,mm$ and the the two left nodes of the element are restrained for their displacements along the $\vec{x}$ and $\vec{y}$ directions and a prescribed horizontal displacement $d = 10\,mm$ is applied on the two right nodes of the same element as illustrated in Figure III.1.3. As we are using an explicit integration scheme, the total simulation time is set to $t = 0.01\,s$.

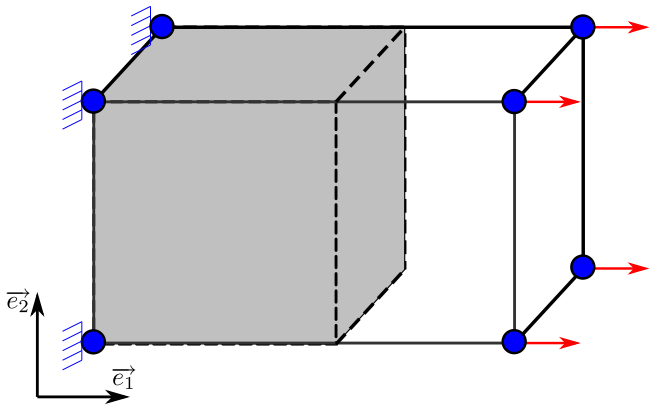


Figure III.1.3 – *Numerical model for the one element tensile test*

All the properties of the constitutive law reported in Table III.3.1 are used and the material is assumed to follow the Johnson-Cook behavior described by equation 3.1.

Figure III.1.4 shows the comparison of the DynELA solver results (plotted in red) and the Abaqus numerical results (plotted in blue) concerning the evolution of the stress components σ_{11}, σ_{22}, σ_{12}, $\overline{\sigma}$, $\overline{\varepsilon}^p$ and T *vs.* the horizontal displacement of the right edge of the specimen along the horizontal axis. For the Abaqus model, the exact same mesh has been used. Comparison of Abaqus and DynELA results is made by averaging the DynELA and the Abaqus results on the 8 integration points of the element.

1.1.3 Element radial tensile test

The uniaxial one element radial tensile test is a numerical test where an axisymmetric element (with a square prescribed shape) is subjected to pure radial tensile as presented in figure III.1.1. The initial shape of the specimen is $10\,mm \times 10\,mm$ and the the two left nodes of the element are encastred and a prescribed horizontal displacement $d = 10\,mm$ is applied on the two right nodes of the same element as illustrated in Figure III.1.1. As we are using an explicit integration scheme, the total simulation time is set to $t = 0.01\,s$.

All the properties of the constitutive law reported in Table III.3.1 are used and the material is assumed to follow the Johnson-Cook behavior described by equation 3.1.

Figure III.1.5 shows the comparison of the DynELA solver results (plotted in red) and the Abaqus numerical results (plotted in blue) concerning the evolution of the stress components σ_{11}, σ_{22}, σ_{12}, $\overline{\sigma}$, $\overline{\varepsilon}^p$ and T *vs.* the horizontal displacement of the right edge of the specimen along the horizontal axis. Again, comparison of Abaqus and DynELA results is made by averaging the DynELA results on the 4 integration points of the element because DynELA Finite Element Code uses full integrated

elements while Abaqus have reduced integrated elements only but in fact, the DynELA results at the 4 integration points are the same in this benchmark test.

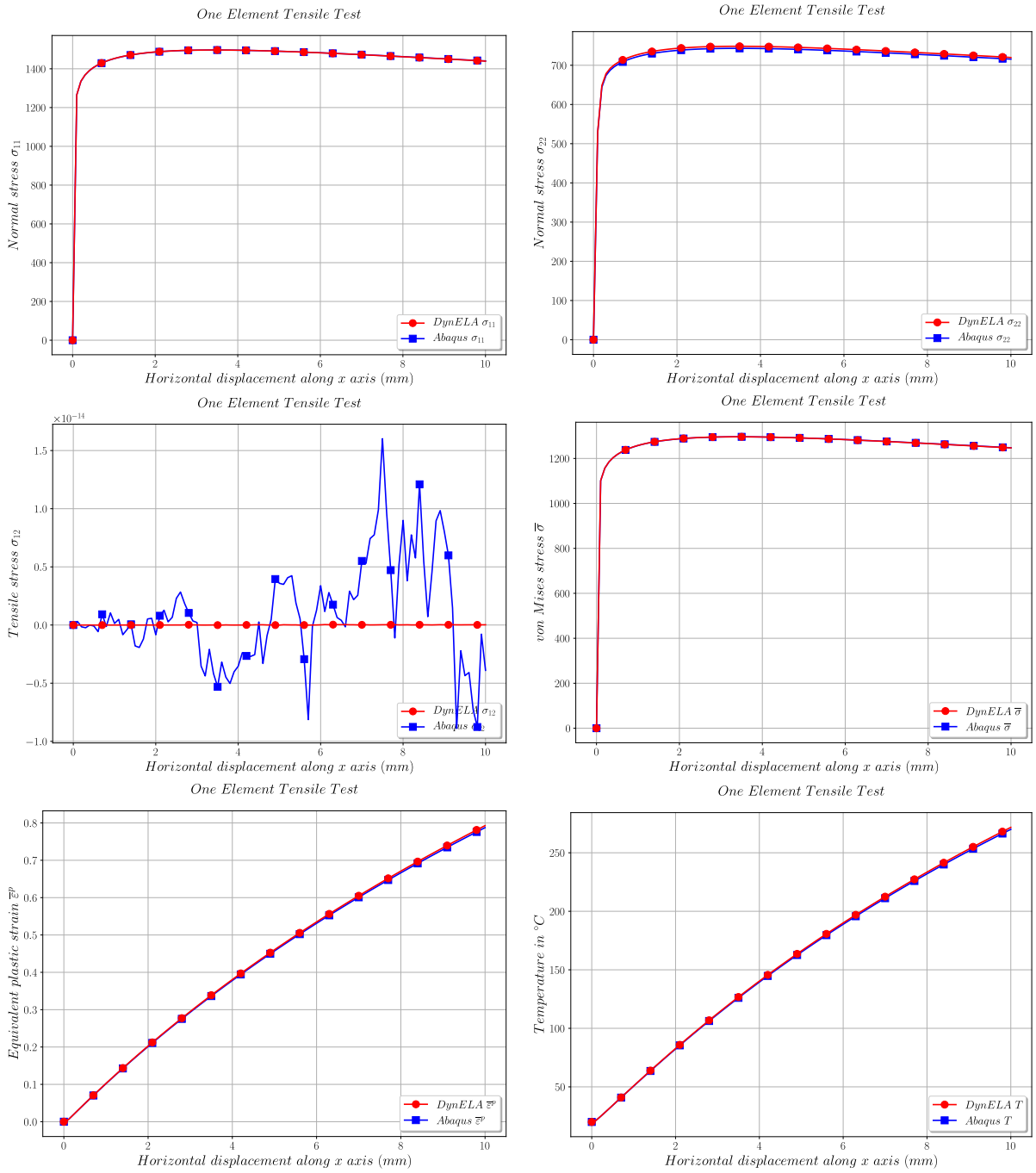


***Figure III.1.4** – Comparison of numerical and analytical results for the 3D one element tensile test*

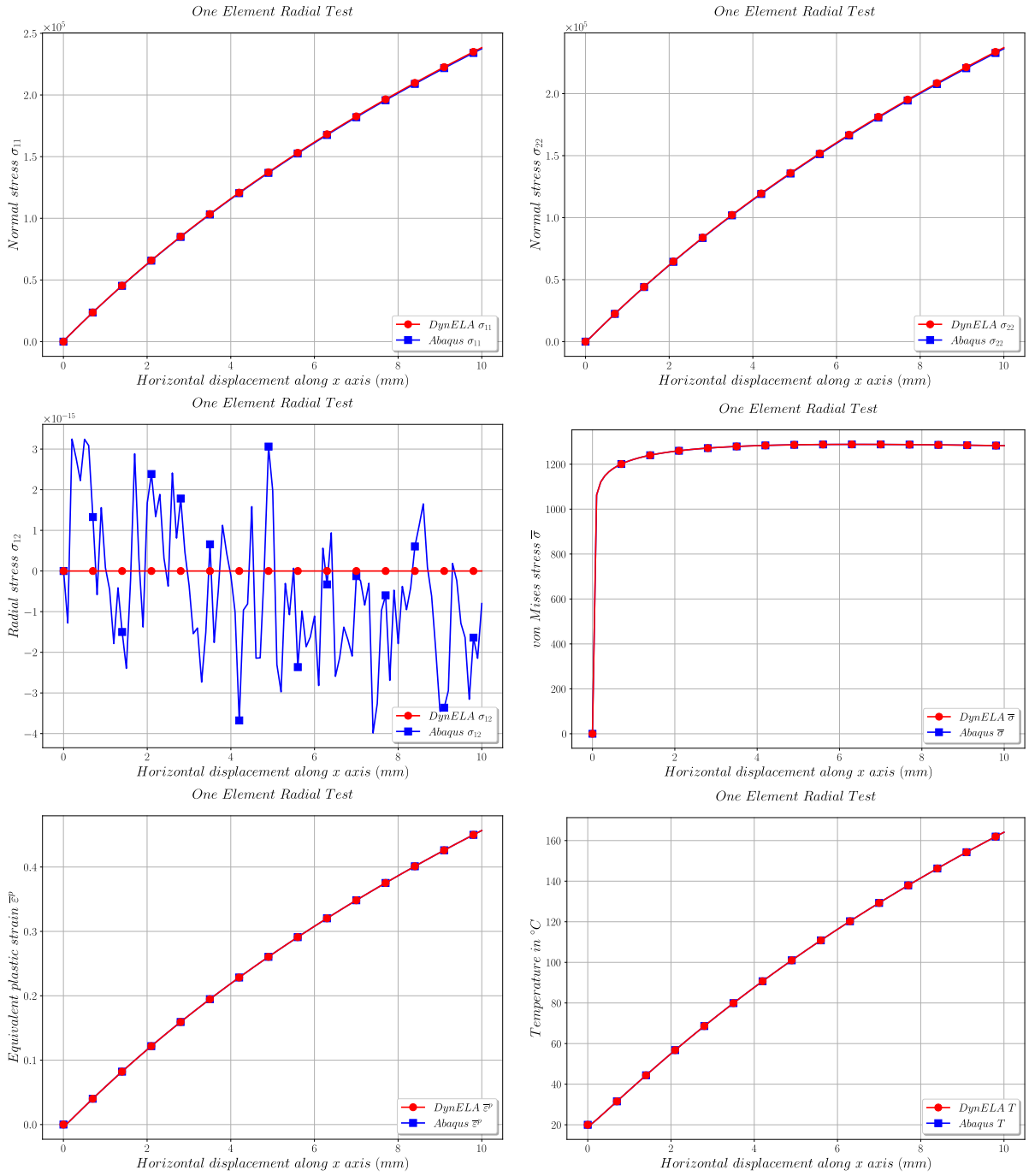


***Figure III.1.5** – Comparison of numerical and analytical results for the one element radial tensile test*

1.1.4 Element radial torus test

The uniaxial one element torus tensile test is a numerical test where an axisymmetric element (with a square prescribed shape) is subjected to radial tensile as presented in figure III.1.6. The difference

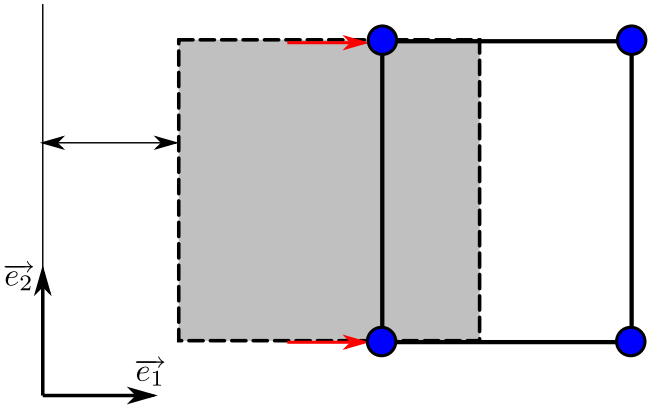


***Figure III.1.6** – Numerical model for the one element torus test*

with the previous test is that the left edge of the specimen in not aligned with the symmetry axis, the radial coordinate of the left edge is $r = 10\,mm$. The initial shape of the specimen is $10\,mm \times 10\,mm$ and a prescribed horizontal displacement $d = 10\,mm$ is applied on the two left nodes of the element as illustrated in Figure III.1.6. As we are using an explicit integration scheme, the total simulation time is set to $t = 0.01\,s$. All the properties of the constitutive law reported in Table III.3.1 are used and the material is assumed to follow the Johnson-Cook behavior described by equation 3.1.

As a global comparison, Figure III.1.7 show the comparison of the right edge displacement *vs.* the left edge displacement for both the DynELA and the Abaqus simulations.

***Figure III.1.7** – Comparison of the right edge displacement for the one element torus test*

Figure III.1.9 shows the comparison of the DynELA solver results (plotted in red) and the Abaqus numerical results (plotted in blue) concerning the evolution of the stress components σ_{11}, σ_{22}, σ_{12}, $\overline{\sigma}$, $\overline{\varepsilon}^p$ and T *vs.* the horizontal displacement of the right edge of the specimen along the horizontal axis. As the Abaqus software only provides under-integrated elements, we are using a 2×2 elements mesh for the Abaqus model, as presented in Figure III.1.8, to compare the results. On abaqus, the

mean value of the results of the 4 elements is computed and compared to the mean value of the 4 integration points of the DynELA simulation.

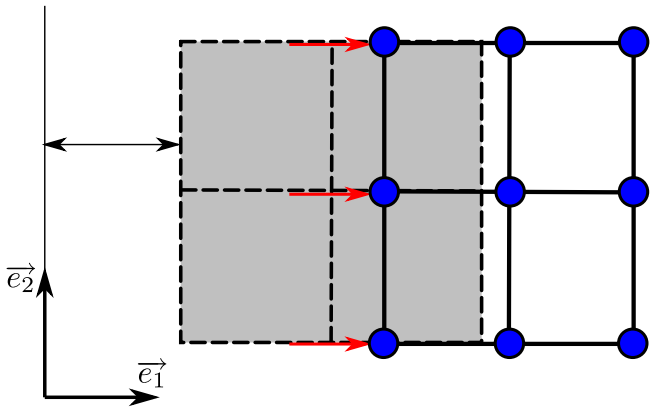


Figure III.1.8 – *Numerical model for the one element torus test under Abaqus*

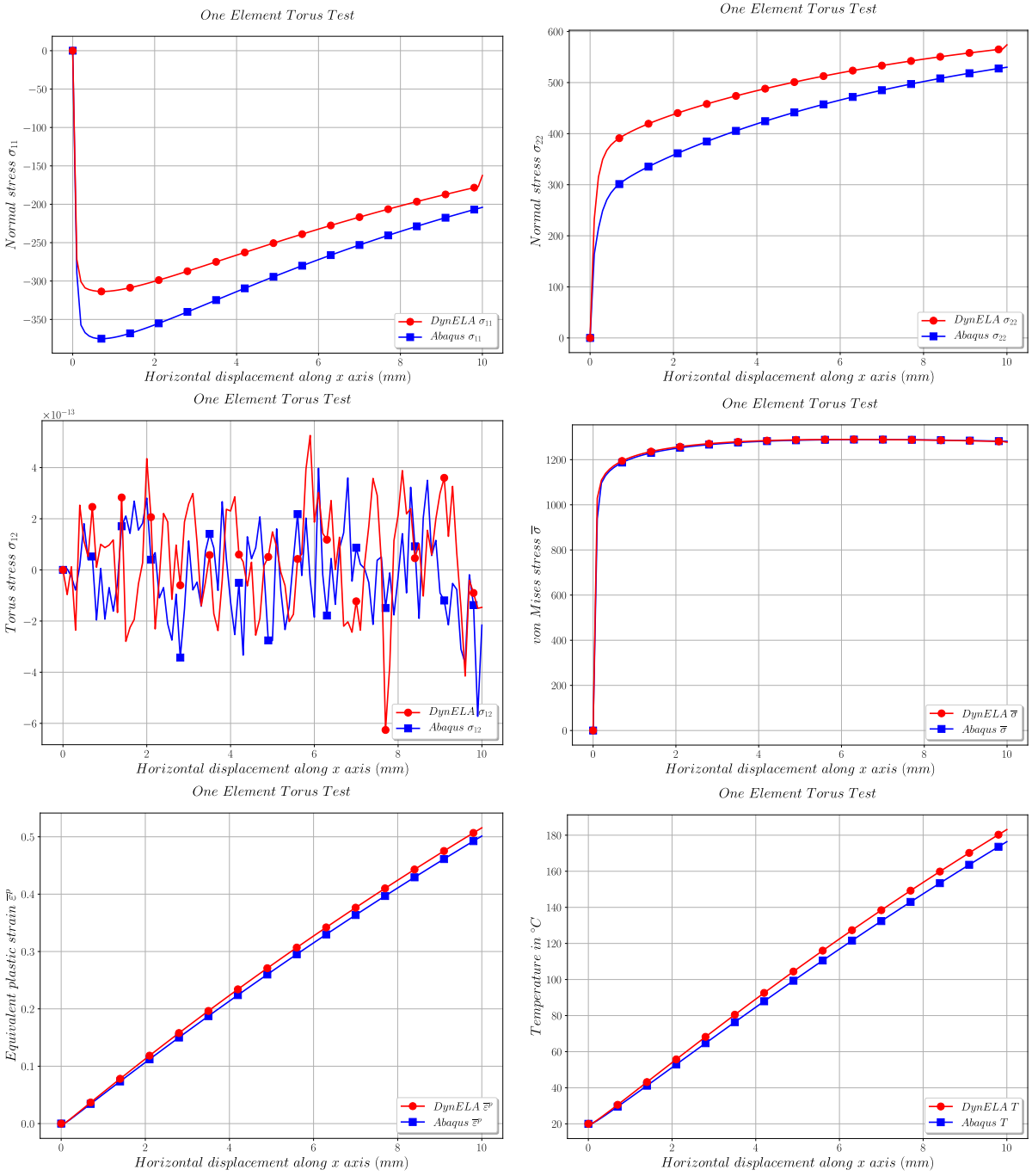


Figure III.1.9 *– Comparison of numerical results for the one element torus tensile test*

1.2 Uniaxial one element shear test

The uniaxial one element shear test is a numerical test where an element (with a square prescribed shape) is subjected to pure shear test as presented in figure III.1.10. The initial shape of the specimen

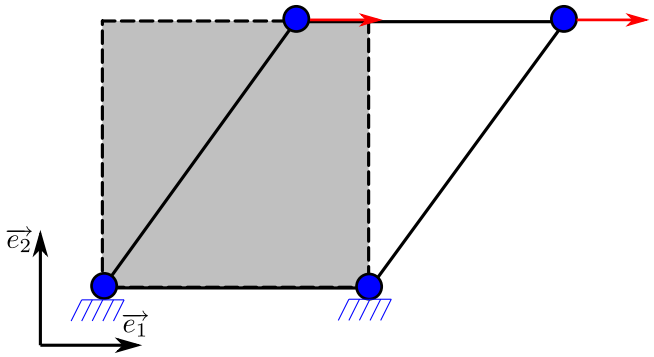


Figure III.1.10 – *Numerical model for the one element shear test*

is $10\,mm \times 10\,mm$ and the the two bottom nodes of the element are encastred and a prescribed horizontal displacement $d = 100\,mm$ is applied on the two upper nodes of the same element as illustrated in Figure III.1.10. As we are using an explicit integration scheme, the total simulation time is set to $t = 0.01\,s$. Mechanical properties of the material are reported in Table III.3.1.

Comparison of Abaqus and DynELA results is made by averaging the DynELA results on the 4 integration points of the element. This has been done because DynELA Finite Element Code uses full integrated elements while Abaqus have reduced integrated elements only but in fact, the DynELA results at the 4 integration points are the same in this benchmark test.

1.2.1 Elastic case

In this case, only the elastic properties of the constitutive law reported in Table III.3.1 are used and the material is assumed to be hyper-elastic. As we are using a Jaumann objective rate within the DynELA Finite Element Code, one can obtain, using an analytical development, the following results for the proposed case:

$$\boldsymbol{\sigma} = G \begin{bmatrix} 1-\cos e & \sin e & 0 \\ & \cos e - 1 & 0 \\ sym & & 0 \end{bmatrix}, \tag{1.2}$$

where $G = \mu = \frac{E}{2(1+\nu)}$ is the shear modulus of the material and e is the elongation along the horizontal axis with $e_{max} = 10$ conforming to the prescribed boundaries conditions. Figure III.1.11 shows the comparison of the DynELA solver results (plotted in red) and both the analytical (plotted in blue) and the Abaqus numerical results (plotted in green) concerning the evolution of the stress components σ_{11}, σ_{22}, σ_{12} and $\overline{\sigma}$ *vs.* the horizontal displacement of the top edge of the specimen along the horizontal axis. A perfect match between all those results can be seen from this later.

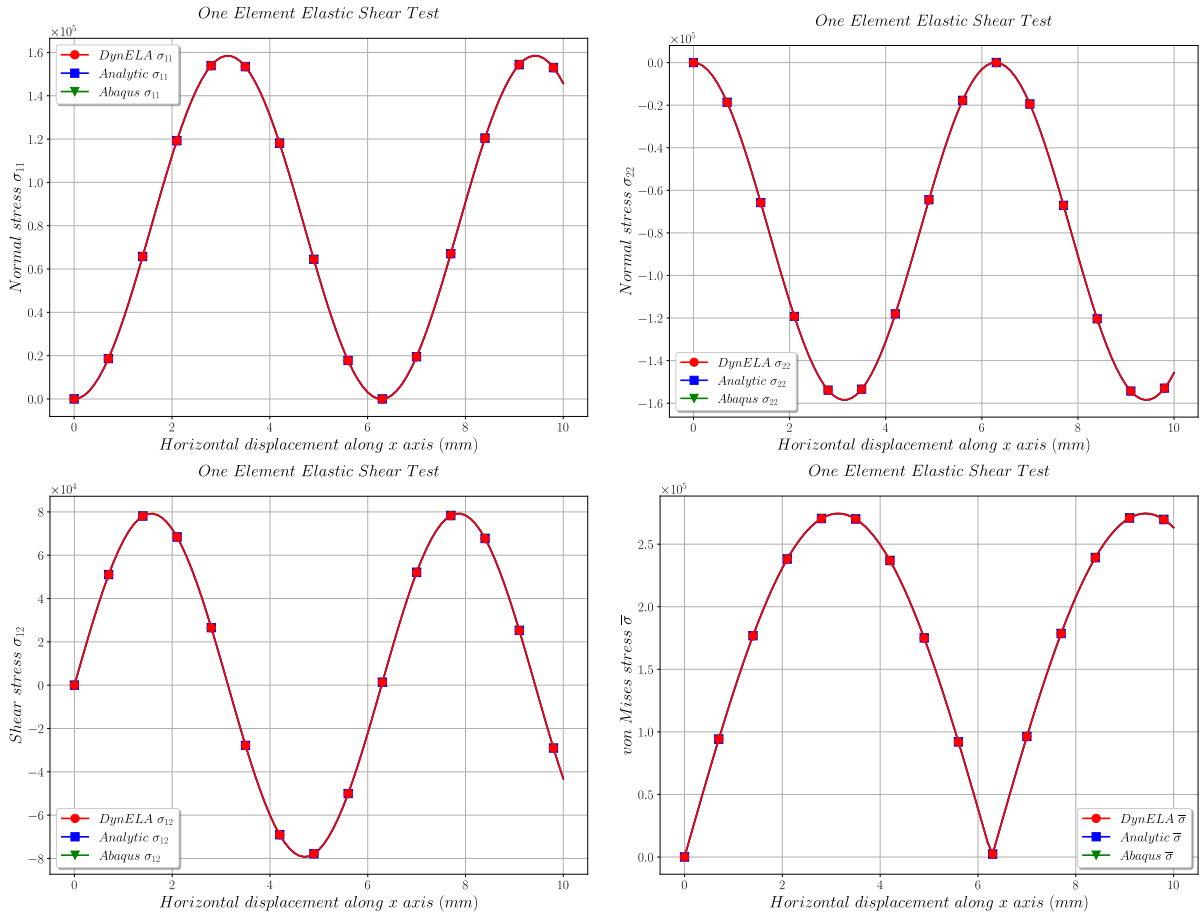


Figure III.1.11 *– Comparison of numerical and analytical results for the one element elastic shear test*

1.2.2 Johnson-Cook plastic behaviour

In this case, all the properties of the constitutive law reported in Table III.3.1 are used and the material is assumed to follow the Johnson-Cook behavior described by equation 3.1. Figure III.1.12 shows the comparison of the DynELA solver results (plotted in red) and the Abaqus numerical results (plotted in blue) concerning the evolution of the stress components σ_{11}, σ_{22}, σ_{12}, $\overline{\sigma}$, $\overline{\varepsilon}^p$ and T *vs.* the horizontal displacement of the top edge of the specimen along the horizontal axis.

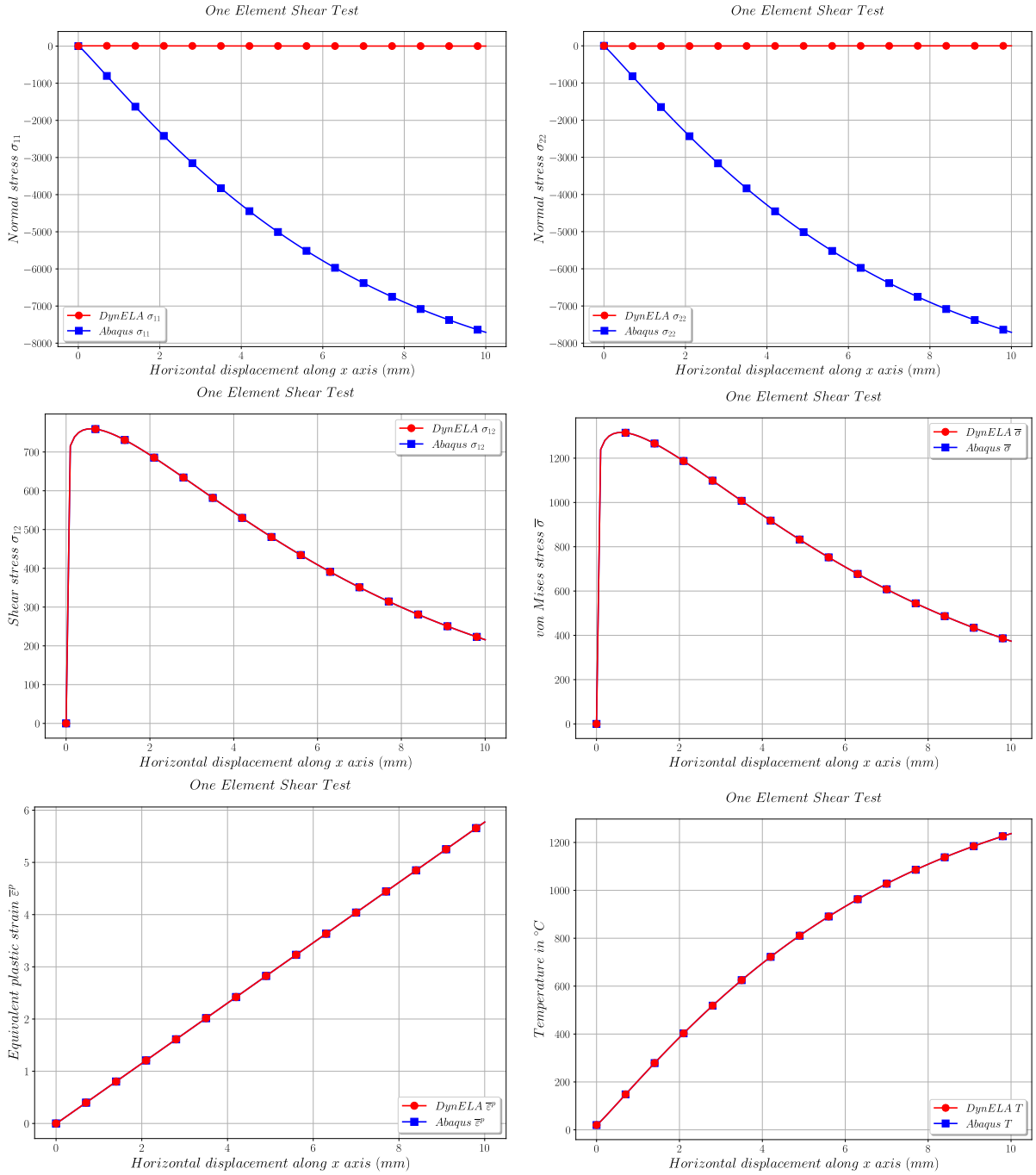


***Figure III.1.12** – Comparison of numerical and analytical results for the one element shear test*

Chapter 2

DynELA plasticity sample cases

2.1 Necking of a circular bar 54

This chapter deals with some numerical applications of the DynELA Finite Element Code for plasticity applications in 2D, axi-symmetric and 3D cases. In the subsequent tests, if not specified, a Johnson-Cook constitutive law is used to model the behavior of the material. The Johnson-Cook hardening flow law is probably the most widely used flow law for the simulation of high strain rate deformation processes taking into account plastic strain, plastic strain rate and temperature effects. Since a lot of efforts have been made in the past to identify the constitutive flow law parameters for many materials, it is implemented in numerous Finite Element codes such as Abaqus [?]. The general formulation of the Johnson-Cook law $\sigma^y(\bar{\varepsilon}^p, \dot{\bar{\varepsilon}}^p, T)$ is given by the following equation:

$$\sigma^y = \left(A + B\bar{\varepsilon}^{p^n}\right)\left[1 + C\ln\left(\frac{\dot{\bar{\varepsilon}}^p}{\dot{\bar{\varepsilon}}_0}\right)\right]\left[1 - \left(\frac{T - T_0}{T_m - T_0}\right)^m\right] \tag{2.1}$$

where $\dot{\bar{\varepsilon}}_0$ is the reference strain rate, T_0 and T_m are the reference temperature and the melting temperature of the material respectively and A, B, C, n and m are the five constitutive flow law parameters. A 42CrMo4 steel following the Johnson-Cook behavior law has been selected for all those tests, and material properties are reported in Table III.3.1.

E (Gpa)	ν	A (MPa)	B (MPa)	C	n	m
206.9	0.3	806	614	0.0089	0.168	1.1
ρ (kg/m^3)	λ $(W/m°C)$	C_p $(J/Kg°C)$	η	$\dot{\bar{\varepsilon}}_0$ (s^{-1})	T_0 $(°C)$	T_m $(°C)$
7830	34.0	460	0.9	1.0	20	1540

***Table III.2.1** – Material parameters of the Johnson-Cook behavior for the numerical tests*

2.1 Necking of a circular bar

2.1.1 Axisymmetric Bar Necking

The necking of a circular bar test is useful to evaluate the performance of VUMAT subroutine for materials in presence of plasticity and large deformation [16,17]. Because of the symmetric structure, an axisymmetric quarter model of the specimen is established. Dimensions of the specimen are reported in Figure III.2.1. The loading is realized through an imposed total displacement of $7\,mm$ along the $\vec{z}$ axis on the left side of the specimen while the radial displacement of the same edge is supposed to remain zero. On the opposite side the axial displacement is restrained while the radial displacement is free. The mesh consists of 400 CAX4RT elements with a refined zone of 200 elements on the right side on $1/3$ of the total height. Again, the total simulation time is set to $t = 0.01\,s$ because of the explicit integration scheme adopted.

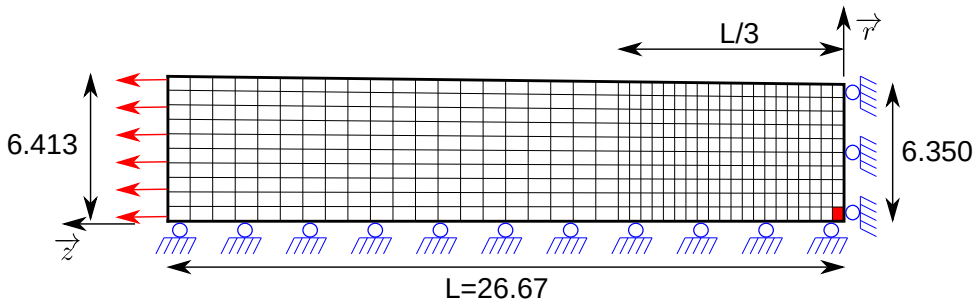


Figure III.2.1 *– Numerical model for the necking of a circular bar*

Figures III.2.2 and III.2.3 shows the temperature T and the von Mises stress $\overline{\sigma}$ contourplots of the deformed rod for both the DynELA Finite Element Code and Abaqus. The distributions of the temperatures and stresses are almost the same for both models. The maximum temperature T is located into the center element of the model (the red element in Figure III.2.1) and the models give quite the same results as reported in Table III.2.2 for $\overline{\varepsilon}^p$, T and the final dimensions of the specimen D_f (final diameter of the necking zone).

Figure III.2.4 shows the evolution of the the final dimensions of the specimen L_f (final length) and R_f (final radius of the impacting face), the equivalent plastic strain $\overline{\varepsilon}^p$, the temperature T, the von Mises stress $\overline{\sigma}$ and the timeStep Δt for the different models for the element at the center of the impacting face (the red element in Figure III.2.1).

As reported in this figure and according to the results presented in Table III.2.2, a quite good agreement between the results is obtained.

code	D_f (mm)	T $(^\circ C)$	$\overline{\varepsilon}^p$
DynELA	2.71	642.78	2.07
Abaqus	2.35	658.91	2.12

Table III.2.2 *– Comparison of numerical results for the necking of a circular bar test*

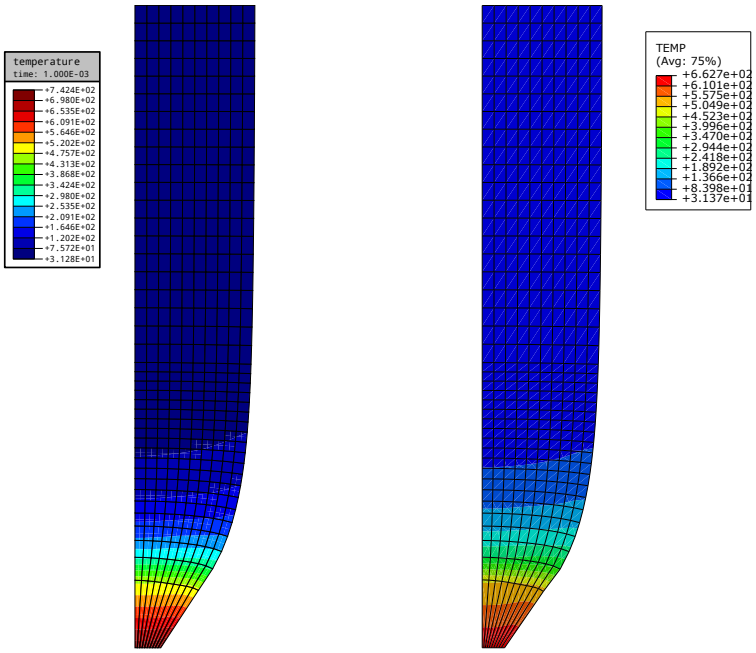


Figure III.2.2 – *Temperature T contourplot for the necking of a circular bar (DynELA left and Abaqus right)*

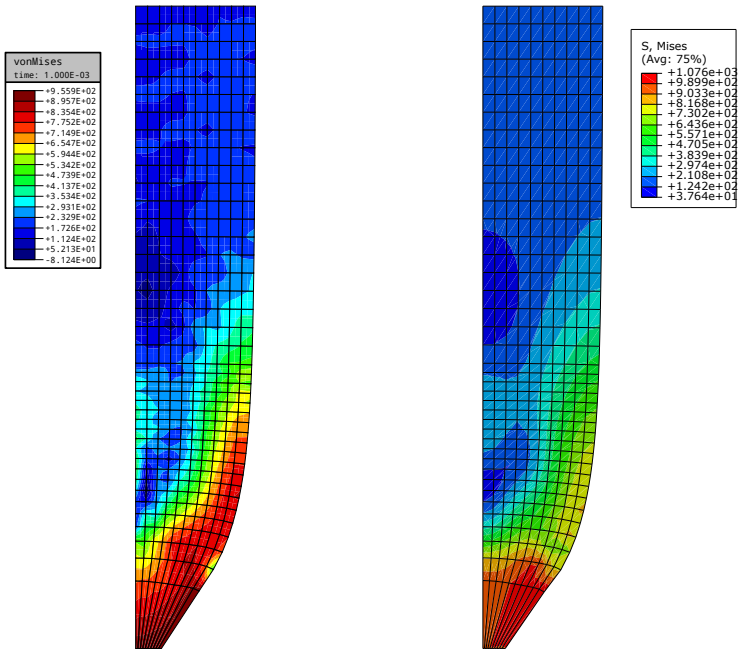


Figure III.2.3 – *von Mises stress $\overline{\sigma}$ contourplot for the necking of a circular bar (DynELA left and Abaqus right)*

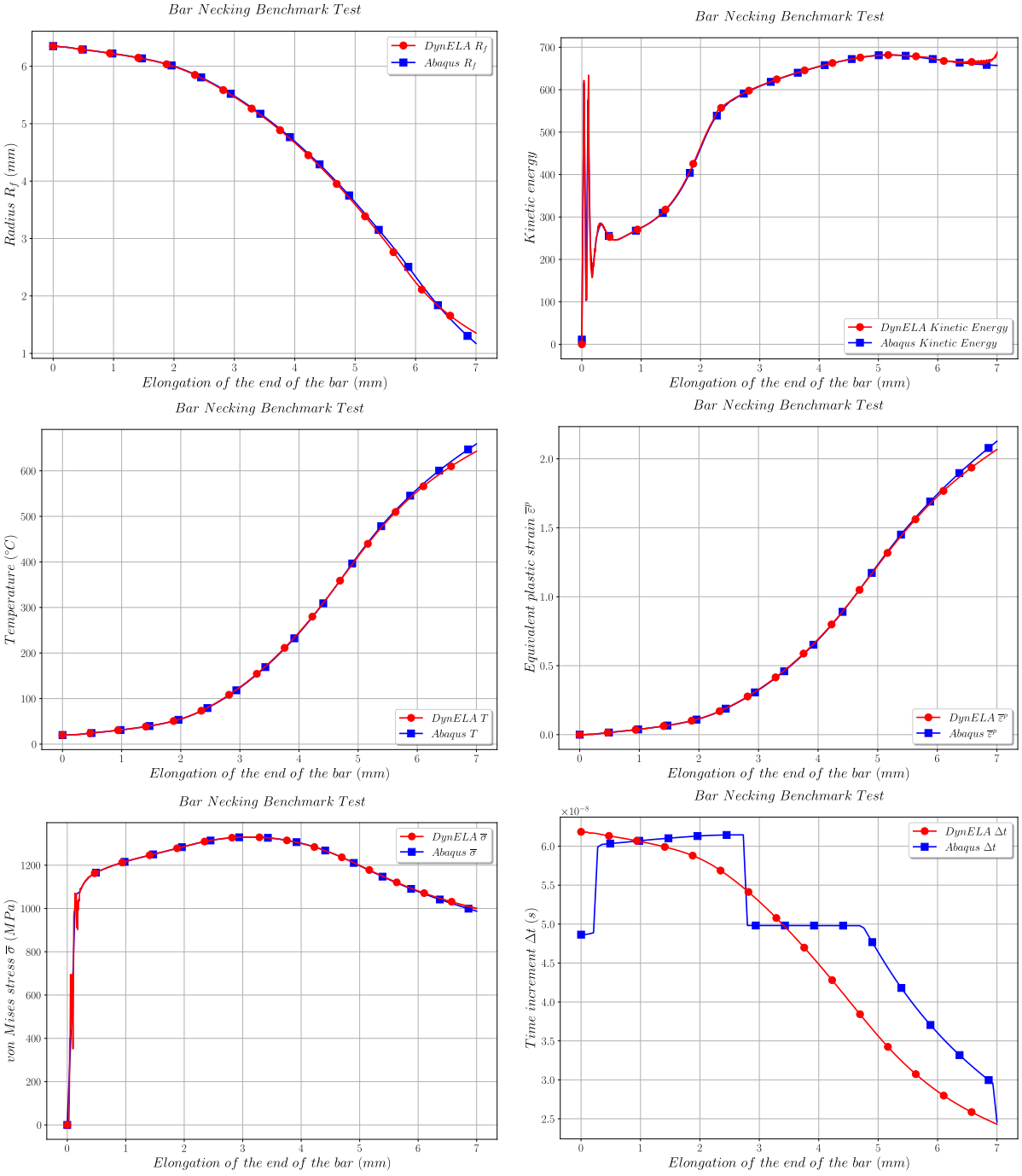


***Figure III.2.4** – Comparison of numerical and analytical results necking of a circular bar test*

Chapter 3

DynELA impact sample cases

3.1 Taylor impact sample 58

This chapter deals with some numerical applications of the DynELA Finite Element Code for impact applications in 2D, axi-symmetric and 3D cases. In the subsequent tests, if not specified, a Johnson-Cook constitutive law is used to model the behavior of the material. The Johnson-Cook hardening flow law is probably the most widely used flow law for the simulation of high strain rate deformation processes taking into account plastic strain, plastic strain rate and temperature effects. Since a lot of efforts have been made in the past to identify the constitutive flow law parameters for many materials, it is implemented in numerous Finite Element codes such as Abaqus [?]. The general formulation of the Johnson-Cook law $\sigma^y(\bar{\varepsilon}^p, \dot{\bar{\varepsilon}}^p, T)$ is given by the following equation:

$$\sigma^y = \left(A + B\bar{\varepsilon}^{p^n}\right)\left[1 + C\ln\left(\frac{\dot{\bar{\varepsilon}}^p}{\dot{\bar{\varepsilon}}_0}\right)\right]\left[1 - \left(\frac{T - T_0}{T_m - T_0}\right)^m\right] \tag{3.1}$$

where $\dot{\bar{\varepsilon}}_0$ is the reference strain rate, T_0 and T_m are the reference temperature and the melting temperature of the material respectively and A, B, C, n and m are the five constitutive flow law parameters. A 42CrMo4 steel following the Johnson-Cook behavior law has been selected for all those tests, and material properties are reported in Table III.3.1.

E (Gpa)	ν	A (MPa)	B (MPa)	C	n	m
206.9	0.3	806	614	0.0089	0.168	1.1
ρ (kg/m^3)	λ $(W/m°C)$	C_p $(J/Kg°C)$	η	$\dot{\bar{\varepsilon}}_0$ (s^{-1})	T_0 $(°C)$	T_m $(°C)$
7830	34.0	460	0.9	1.0	20	1540

***Table III.3.1** – Material parameters of the Johnson-Cook behavior for the numerical tests*

3.1 Taylor impact sample

3.1.1 Axisymmetric Taylor impact

The performance of the proposed code is validated under high deformation rate with the simulation of the Taylor impact test [18]. In the Taylor impact test, a cylindrical specimen is launched to impact a rigid target with a prescribed initial velocity. The numerical model, reported in Figure III.3.1 is established as axisymmetric. The height is $32.4\,mm$ and the radius is $3.2\,mm$. The axial displacement is restrained on the right side of the specimen while the radial displacement is free (to figure a perfect contact without friction of the projectile onto the target). A predefined velocity of $V_c = 287\,m/s$ is imposed on the specimen. The mesh consists of 250 elements (5×50 elements). The total simulation time for the Taylor impact test is $t = 8.0\,10^{-5}s$.

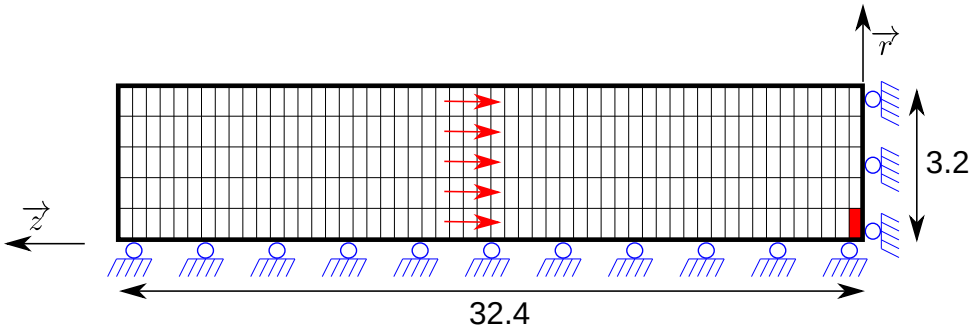


***Figure III.3.1** – Numerical model for the Axisymmetric Taylor impact test*

Figure III.3.2 shows the temperature contourplot of the deformed rod for both the DynELA Finite Element Code and Abaqus. The distributions of the temperatures are almost the same for both models. The maximum temperature T is located into the center element of the model (the red element in Figure III.3.1) and the models give quite the same results as reported in Table III.3.2 for $\overline{\varepsilon}^p$, T and the final dimensions of the specimen L_f (final length) and D_f (final diameter of the impacting face).

Figure III.3.3 shows the evolution of the the final dimensions of the specimen L_f (final length) and R_f (final radius of the impacting face), the equivalent plastic strain $\overline{\varepsilon}^p$, the temperature T, the von Mises stress $\overline{\sigma}$ and the timeStep Δt for the different models for the element at the center of the impacting face (the red element in Figure III.3.1).

As reported in this figure and according to the results presented in Table III.3.2, a quite good agreement between the results is obtained.

code	L_f (mm)	D_f (mm)	T $(°C)$	$\overline{\varepsilon}^p$
DynELA	26.52	11.15	582.34	1.78
Abaqus	26.56	11.16	590.96	1.81

***Table III.3.2** – Comparison of numerical results for the Axisymmetric Taylor impact test*

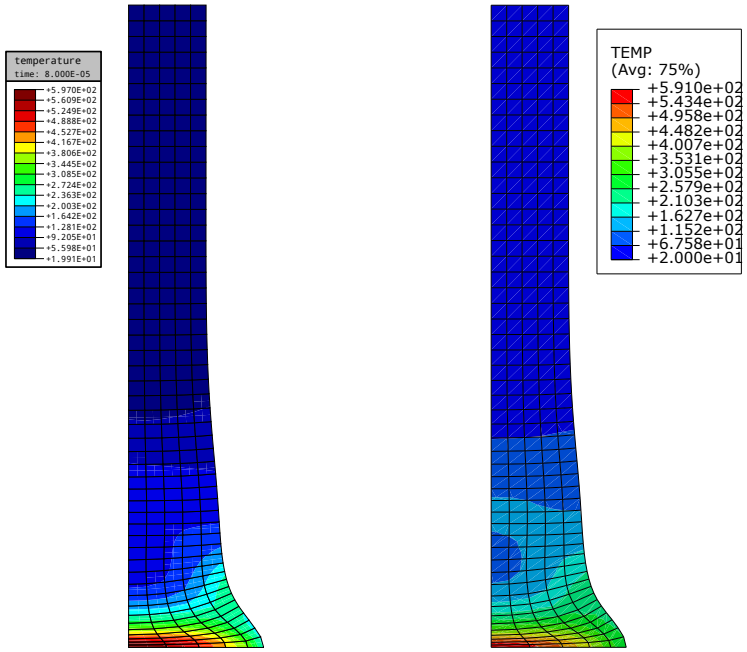


Figure III.3.2 *– Temperature contourplots for Axisymmetric Taylor impact test (DynELA left and Abaqus right)*

3.1.2 3D Taylor impact

The performance of the proposed code is validated under high deformation rate with the simulation of the Taylor impact test [18]. In the Taylor impact test, a cylindrical specimen is launched to impact a rigid target with a prescribed initial velocity. The numerical model, reported in Figure III.3.4 is established as axisymmetric. The height is $32.4\,mm$ and the radius is $3.2\,mm$. The axial displacement is restrained on the right side of the specimen while the radial displacement is free (to figure a perfect contact without friction of the projectile onto the target). A predefined velocity of $V_c = 287\,m/s$ is imposed on the specimen. The mesh consists of 4455 elements (55×81 elements). The total simulation time for the Taylor impact test is $t = 8.0\,10^{-5}s$.

Figure III.3.5 shows the temperature contourplot of the deformed rod for both the DynELA Finite Element Code and Abaqus. The distributions of the temperatures are almost the same for both models. The maximum temperature T is located into the center element of the model (the red element in Figure III.3.4) and the models give quite the same results as reported in Table III.3.3 for $\overline{\varepsilon}^p$, T and the final dimensions of the specimen L_f (final length) and D_f (final diameter of the impacting face).

Figure III.3.6 shows the evolution of the the final dimensions of the specimen L_f (final length) and R_f (final radius of the impacting face), the equivalent plastic strain $\overline{\varepsilon}^p$, the temperature T, the von Mises stress $\overline{\sigma}$ and the timeStep Δt for the different models for the element at the center of the impacting face (the red element in Figure III.3.4).

As reported in this figure and according to the results presented in Table III.3.3, a quite good agreement between the results is obtained.

code	L_f (mm)	D_f (mm)	T $(°C)$	$\overline{\varepsilon}^p$
DynELA	26.52	11.18	597.10	1.84
Abaqus	26.55	11.22	597.01	1.84

***Table III.3.3** – Comparison of numerical results for the 3D Taylor impact test*

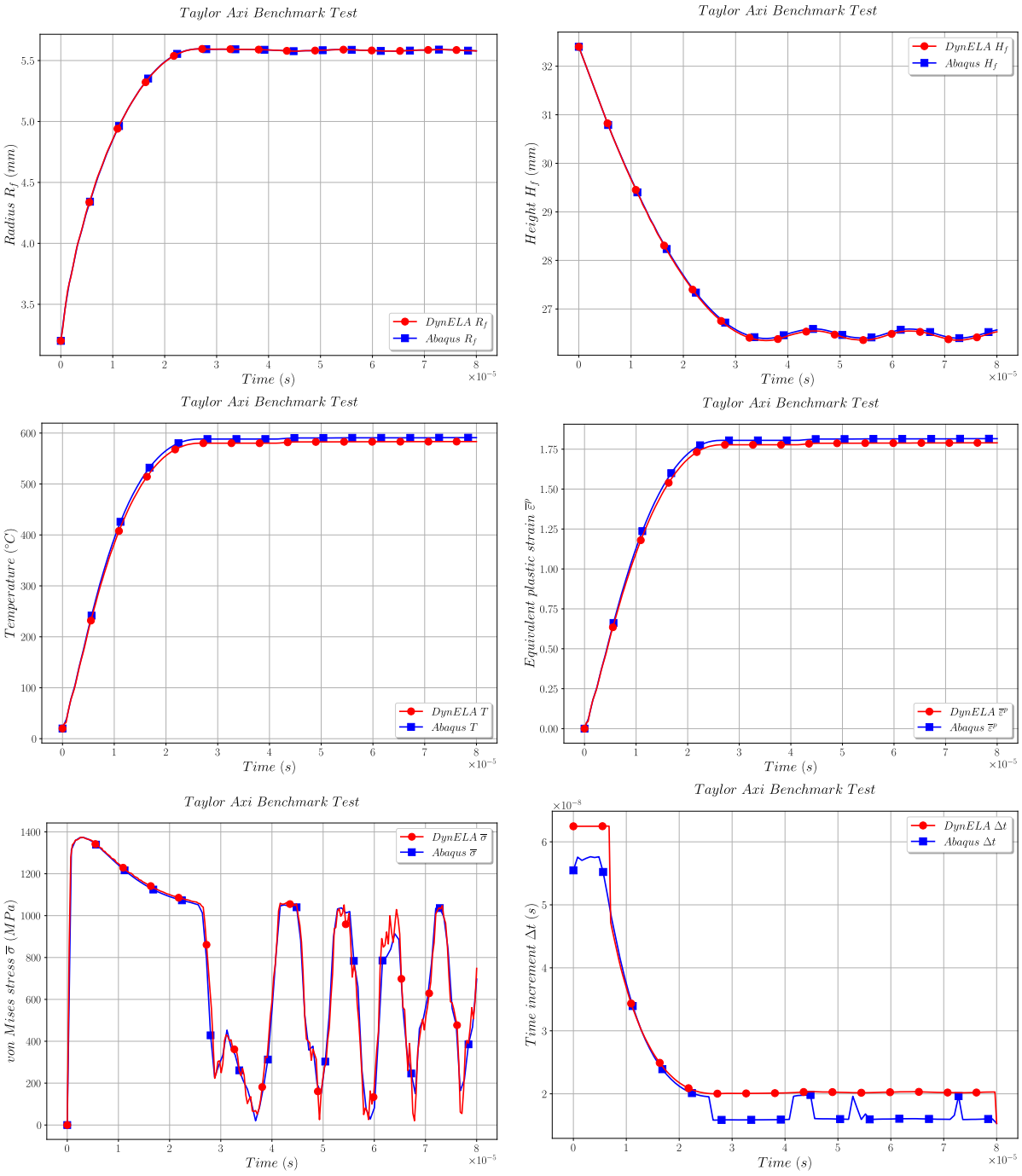


***Figure III.3.3** – Comparison of numerical and analytical results for the Axisymmetric Taylor impact test*

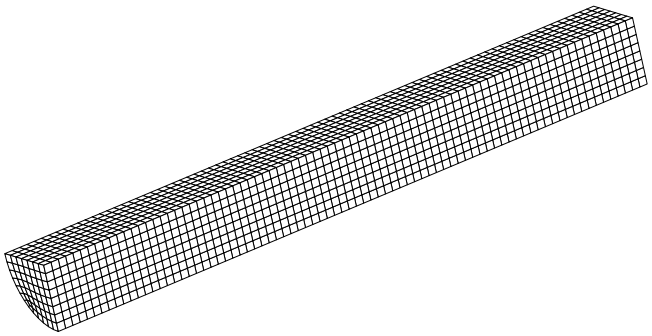


Figure III.3.4 *– Numerical model for the 3D Taylor impact test*

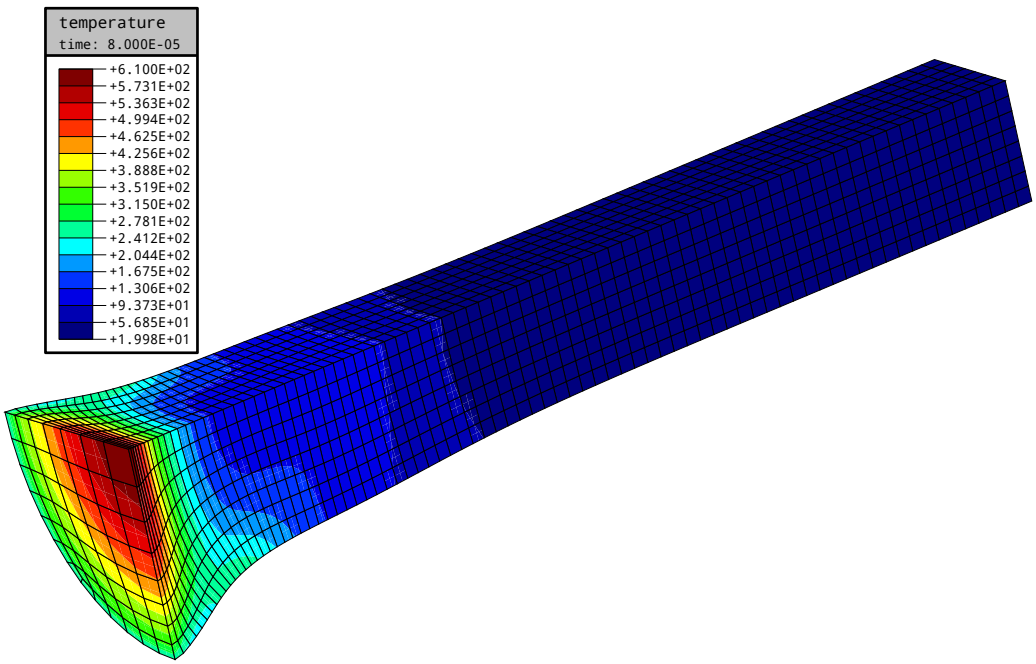


Figure III.3.5 *– Temperature contourplot for the 3D Taylor impact test*

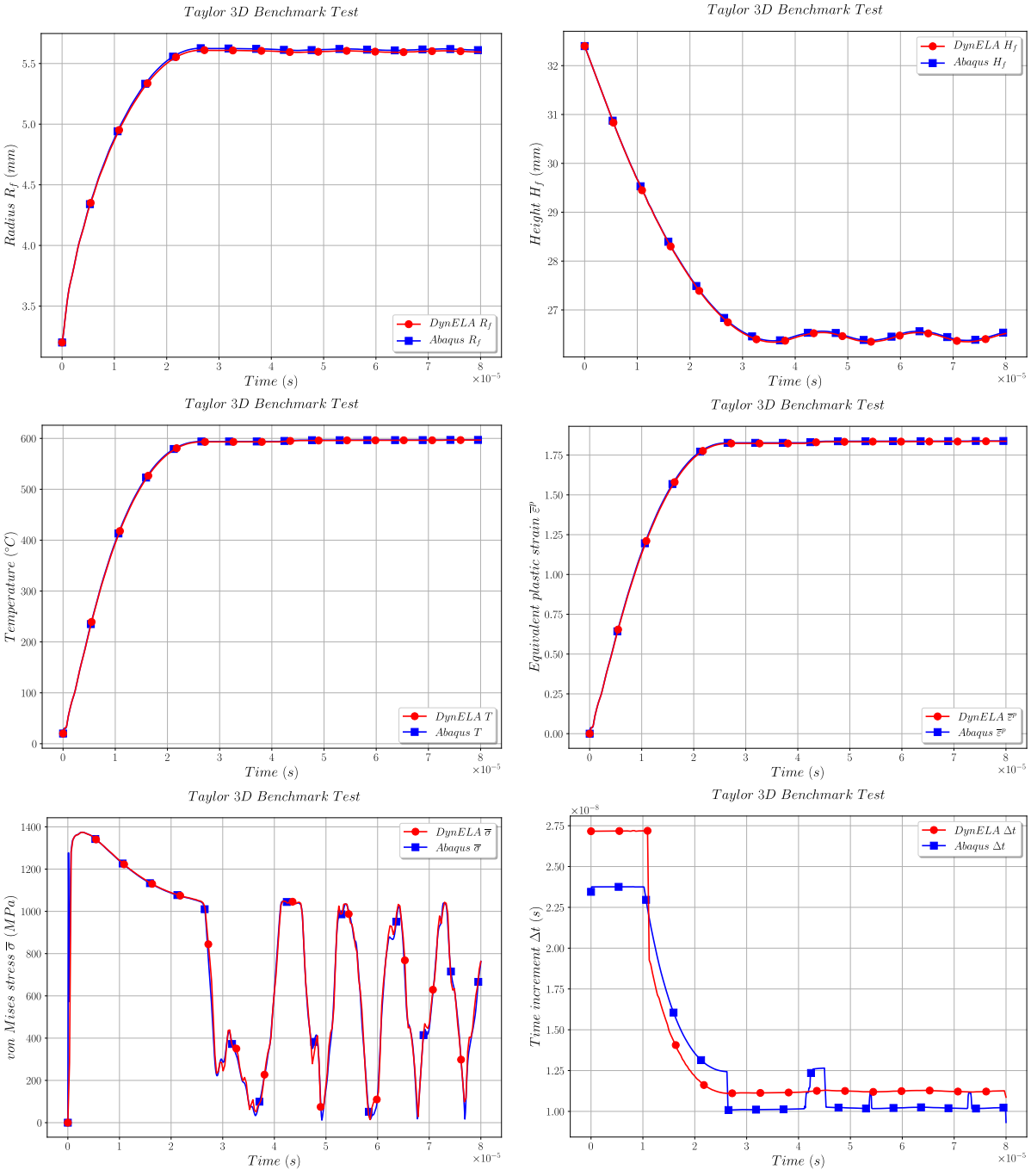


Figure III.3.6 *– Comparison of numerical and analytical results for the 3D Taylor impact test*

Bibliography

[1] Olivier Pantalé. An object-oriented programming of an explicit dynamics code: application to impact simulation. *Advances in Engineering Software*, 33(5):297–306, May 2002. 5

[2] Olivier Pantalé, Serge Caperaa, and Roger Rakotomalala. Development of an object-oriented finite element program: application to metal-forming and impact simulations. *Journal of Computational and Applied Mathematics*, 168(1):341–351, July 2004. 5

[3] Olivier Pantalé. Parallelization of an object-oriented FEM dynamics code: influence of the strategies on the Speedup. *Advances in Engineering Software*, 36(6):361–373, June 2005. 5

[4] Laurent Menanteau, Olivier Pantalé, and Serge Caperaa. A methodology for large scale finite element models, including multi-physic, multi-domain and multi-timestep aspects. *European Journal of Computational Mechanics*, 15(7-8):799–824, January 2006. 5

[5] Ionel Nistor, Olivier Pantalé, and Serge Caperaa. Numerical propagation of dynamic cracks using X-FEM. *European Journal of Computational Mechanics*, 16(2):183–198, January 2007. 5

[6] Ionel Nistor, Olivier Pantalé, and Serge Caperaa. Numerical implementation of the eXtended Finite Element Method for dynamic crack analysis. *Advances in Engineering Software*, 39(7):573–587, July 2008. 5

[7] Olivier Pantalé. [Rp] Parallelization of an object-oriented FEM dynamics code. *ReScience C*, 6(1):#8, June 2020. 5

[8] Laurent Menanteau, Olivier Pantalé, and Serge Caperaa. A coupled electro-thermo-mechanical FEM code for large scale problems including multi-domain and multiple time-step aspects. ISBN:84-95999-71-4, Santorini Island, May 2005. 5

[9] Ionel Nistor, Olivier Pantalé, and Serge Caperaa. On the modeling of the dynamic crack propagation by extended finite element method: numerical implementation in DynELA code. In CIMNE, editor, *VIII International Conference on Computational Plasticity*, pages 303–306, Barcelona, 2005. 5

[10] Olivier Pantalé and Serge Caperaa. Strategies for a parallel 3D FEM code: Application to impact and crash problems. In *Coupled Problems 2005 - Santorini Island*, 2005. 5

[11] Olivier Pantalé and Serge Caperaa. Développement d'un code de calcul explicite en grandes transformations: Application à la coupe des métaux. In *Séminaire Optimus*, 2004. 5

[12] Olivier Pantalé, Roger Rakotomalala, and Serge Caperaa. Développement d'un code de calcul explicite ALE : Application à la coupe des métaux. In *14 ème congrés francais de mécanique*, page 951, Toulouse, September 1999. 5

[13] Olivier Pantalé, Serge Caperaa, and Roger Rakotomalala. Development of an object-oriented finite element program: application to metal-forming and impact simulations. In *ACOMEN 2002 Second international conference on advanced computational methods in engineering*, ISBN 2-930322-39-X, Liège University (Belgium), May 2002. 5

[14] Laurent Francis Menanteau. *Développement d'un module de prototypage virtuel multiphysique, multidomaine et multitemps : application aux convertisseurs de puissance*. PhD Thesis, Toulouse, INPT, January 2004. 5

[15] Ionel Nistor. *Identification expérimentale et simulation numérique de l'endommagement en dynamique rapide : application aux structures aéronautiques*. PhD Thesis, Toulouse, INPT, November 2005. 5

[16] J. P. Ponthot. Unified stress update algorithms for the numerical simulation of large deformation elasto-plastic and elasto-viscoplastic processes. *International Journal of Plasticity*, 18(1):91–126, January 2002. 54

[17] J. C. Simo and T.J.R. Hughes. *Computational Inelasticity*. Springer, July 1998. /home/pantale/SynologyDrive/Bibliotheque/Livres Techniques/J. C. Simo/Computational Inelasticity (90)/Computational Inelasticity - J. C. Simo.pdf. 54

[18] G. I. Taylor. JAMES FORREST LECTURE 1946. THE TESTING OF MATERIALS AT HIGH RATES OF LOADING. *Journal of the Institution of Civil Engineers*, 26(8):486–519, October 1946. 58, 59

Programming Language

dnlPython, 14
    DynELA(string), 16
DynELA
    add(HistoryFile), 25
    createElement(int, int, int,...), 17
    createNode(int, float, float, float), 16
    createNode(int, Vec3D), 16
    initSolve( ), 26
    setDefaultElement(Element), 17
    setSaveTimes(float, float, float), 23
    solve( ), 26
    writeVTKFile( ), 24

HistoryFile
    add(ElementSet), 24
    add(Field), 24
    add(NodeSet), 24
    getFileName( ), 25
    getSaveTime( ), 25
    getStartTime( ), 25
    getStopTime( ), 25
    setFileName(string), 25
    setSaveTime(float), 25
    setSaveTime(float, float, float), 25

NodeSet(string), 16

parallel
    getCores( ), 26
    setCores(int), 26

Solver
    setComputeTimeStepFrequency(int), 25
    setIncrements(int, int), 25
    setTimes(float, float), 25
    setTimeStepMethod(method), 25
    setTimeStepSafetyFactor(float), 25
SvgInterface
    resetView( ), 28
    rotate(string, float), 28
    rotate(Vec3D, float), 28
    setAutoRangeValues(bool), 28
    setInfoDisplay(bool), 28
    setInfoPosition(int, int), 28
    setLegendDisplay(bool), 28
    setLegendPosition(int, int), 28
    setMeshDisplay(bool), 28
    setMeshThickness(float), 28
    setPatchLevel(int), 28
    setTitleDisplay(bool), 28
    setTitlePosition(int, int), 28
    write(string), 29
    write(string, Field), 29