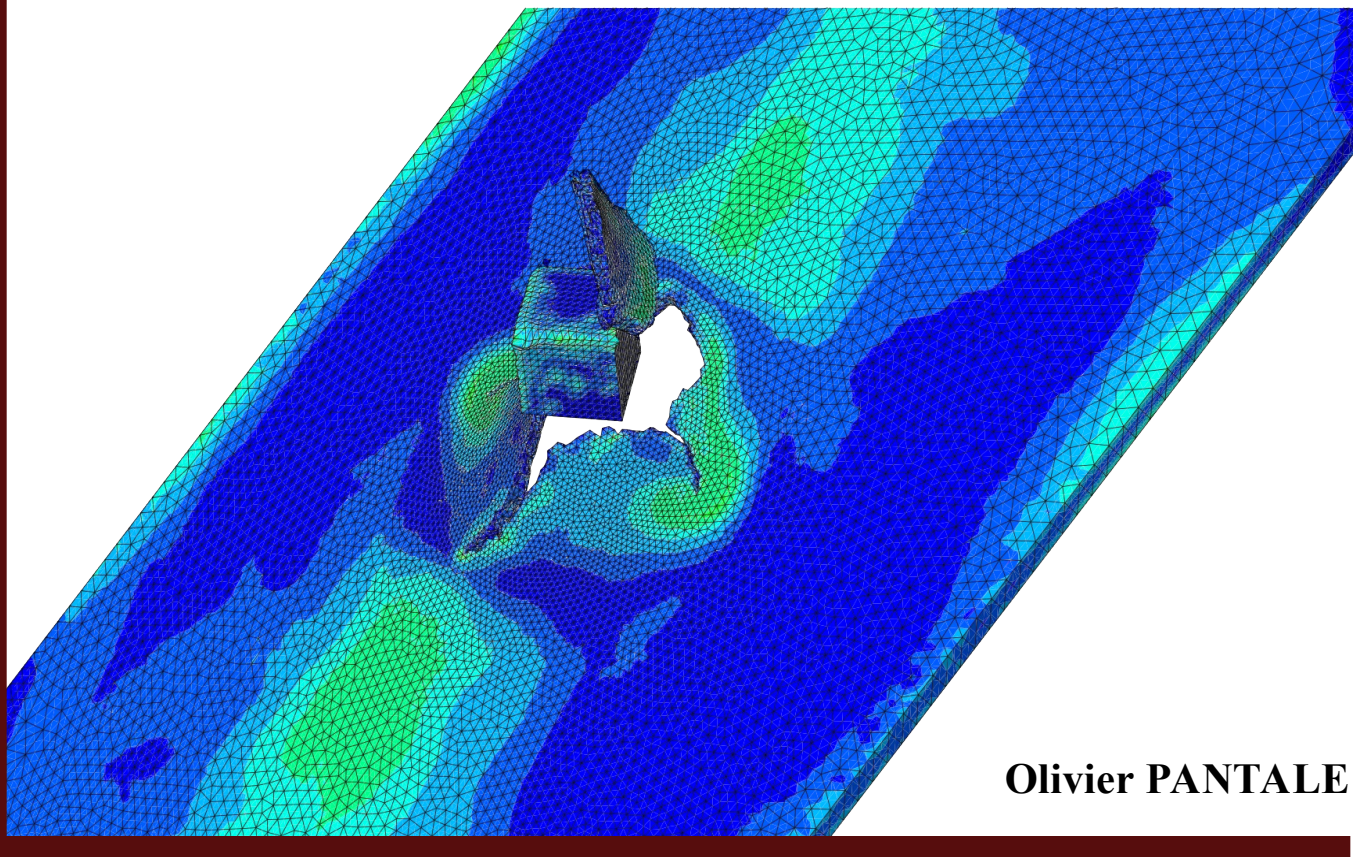
DynELA Finite Element Code v.3.0



Olivier PANTALE

Contents

List of Figures

Part I

DynELA FEM code theory

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 $\square$ ($\text{tr}\ \square = \sum \square_{ii}$)
dev $\square$	Deviatoric part of a tensor $\square$ ($\text{dev}\ \square = \square - \frac{1}{3}\text{tr}\ \square\, 1$)
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	
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
$\boldsymbol{\phi}$	

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

Part II

DynELA Programming

Chapter 1

DynELA programming language

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 language.

1.1 Introduction

1.1.1 Calling the python interpreter

After the compilation phase of the code, the DynELA Finite Element Code can be launch using the following command:

python <'model.py'>: where model.py is the source Python 3 file defining the DynELA Finite Element Code. The model.py file contains all the definitions of the Finite Element Model using a Python 3 language and calling DynELA methods written in C++.

1.1.2 Formalism of a DynELA python file

In order to build a Finite Element Model, it is necessary to import the dnlPython interpreter from the .py script. Conforming to this formalism, the minimal piece of Python code to setup a Finite Element Model is given hereafter.

```
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                         # and conforming to the DynELA language and Python 3
6 model.solve()           # Runs the solver
```

In the preceding piece of code, line 2 is used to load into memory the module dnlPython containing the interface to all C++ methods of the DynELA Finite Element Code based on the use of the SWIG Python interface. Therefore, all public methods of the DynELA Finite Element Code written in C++ can be called from the Python script to build the FEM model, lanch teh solver, produce output results...

In the minimal proposed example, 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[1], while line 6, the solver of the DynELA Finite Element Code library is called to solve the problem and produce the results.

As the interpretor of the DynELA Finite Element Code is based on Python 3 language, all kind of instructions valid in Python 3 can be used along with the specific DynELA instructions.

1.2 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 setup a FE 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.

[1] For the rest of this chapter, we will assume that the name of the instanciated DynELA object is model.

1.2.1 Definition of the Model

1.2.2 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)  # Node 1 [0.0, 0.0, 0.0]
2 model.createNode(2, 1.0, 2.0, -1.0) # Node 2 [1.0, 2.0, -1.0]
```

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.

1.2.3 Definitions of the elements

In the DynELA Finite Element Code, 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 tanks to the element definition as presented just below. Before creating the very first element of the structure, it is necessary to define the element shape using the DynELA.setDefaultElement() method as presented hereafter.

```
1 model.setDefaultElement(dnl.Element.ElQua4N2D) # Defines the default element
2 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.

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

1.2.4 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 to a name (a string use to identify the object), 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")
```

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 add(nodeset, start, end, increment). Hereafter are 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 to 4 to node set
model.add(nset, 1, 4, 2) # Add nodes number 1 and 3 to node set
```

1.2.5 Element sets

Application of materials, application of 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 (a string use to identify the object), and a list of elements. Creation of an ElementSet is done using the DynELA.ElementSet() method that returns an 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 nodes constituting the ElementSet with the generic DynELA.add() method with the following formalism add(elementset, start, end, increment). Hereafter are some self explaining examples to illustrate this process.

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

1.3 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.

1.3.1 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, 0.0, 0.0) # Defines the translation vector
model.translate(vector)           # Translates the whole model along [1.0, 0.0, 0.0]
model.translate(vector, nset)     # Translates the NodeSet ns along [1.0, 0.0, 0.0]
```

1.3.2 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 and apply this rotation to the whole structure (without specifying the NodeSet) or a NodeSet using the DynELA.rotate() method with the following syntax.

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

1.3.3 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 syntaxes.

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

1.4 Materials

1.4.1 General properties

General properties of materials in Dynela concerns the general constants such as Young's modulus, Poisson's ratio, density,... The complete list of parameters is given hereafter.

- youngModulus = The young modulus E of the material to define the elastic behaviour of the material.
- poissonRatio = The Poisson ratio ν of the material to define the elastic behaviour of the material.
- density = The density ρ of the material.
- heatCapacity = The heat capacity C_p of the material.
- taylorQuinney = The Taylor-Quinney coefficient η defining the amount of plastic energy converted into heat during a plastic transformation of the material.
- initialTemperature = The initial temperature T_0 of the material at the beginning of the computation.

After creating an instance of the object dnl.Material, on can apply the prescribed values to all 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
```

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
model.add(steel, eset)
```

1.4.2 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 in order
```

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)
```

1.5 Boundaries conditions

1.5.1 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)
```

1.5.2 Amplitude

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

1.5.3 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)
```

1.5.4 Initial speed

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

1.6 Data output

1.6.1 History files

```
dtHist = dnl.HistoryFile("dtHistory")
dtHist.setFileName(dnl.String("dt.plot"))
dtHist.add(dnl.Field.timeStep)
dtHist.setSaveTime(stopTime/nbrePoints)
model.add(dtHist)
```

```
# Declaration of the history files
vonMisesHist = dnl.HistoryFile("vonMisesHistory")
vonMisesHist.setFileName(dnl.String("vonMises.plot"))
vonMisesHist.add(histNS, dnl.Field.vonMises)
vonMisesHist.setSaveTime(stopTime/nbrePoints)
model.add(vonMisesHist)
```

1.6.2 Data files

```
model.setSaveTimes(0, stopTime, stopTime/nbreSaves)
```

1.7 Solvers

```
# Declaration of the explicit solver
solver = dnl.Explicit("Solver")
solver.setTimes(0, stopTime)
solver.setTimeStepSafetyFactor(1.0)
model.add(solver)
```

1.7.1 Parallel solver

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

1.7.2 Solving procedure

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

1.8 Utilities

```
# Plot the results as curves
import dnlCurves as cu
curves = cu.Curves()
curves.plotFile('Curves.ex')
```

Chapter 2

Curves utility

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
......
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
......
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

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 datafileby 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:

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 SQUARE-1.3
3 # Extraction of a time history integration point data
4 TimeHistory, job=Torus, value=MISES, name=vonMises, region=Element SQUARE-1.1 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 SQUARE-1 piece.
- line 4 is used to extract the integration point value MISES from the odb file Torus.odb and produce the vonMises.plot file for integration point 1 of element 1 of the SQUARE-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

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.1.1.

1.1 Uniaxial tensile tests

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

1.1.1 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

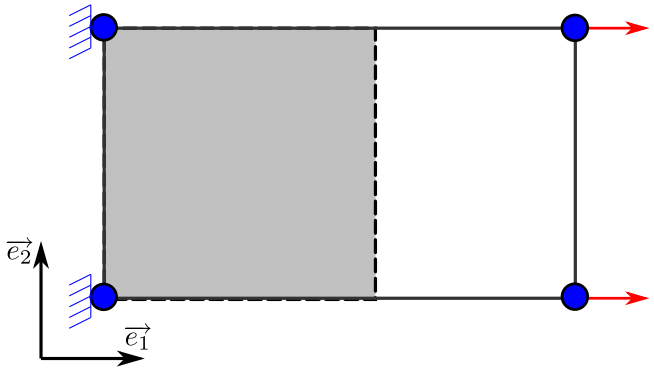


Figure III.1.1: Numerical model for the one element tensile test

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.1.1 are used and the material is assumed to follow the Johnson-Cook behavior described by equation 1.1.

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.

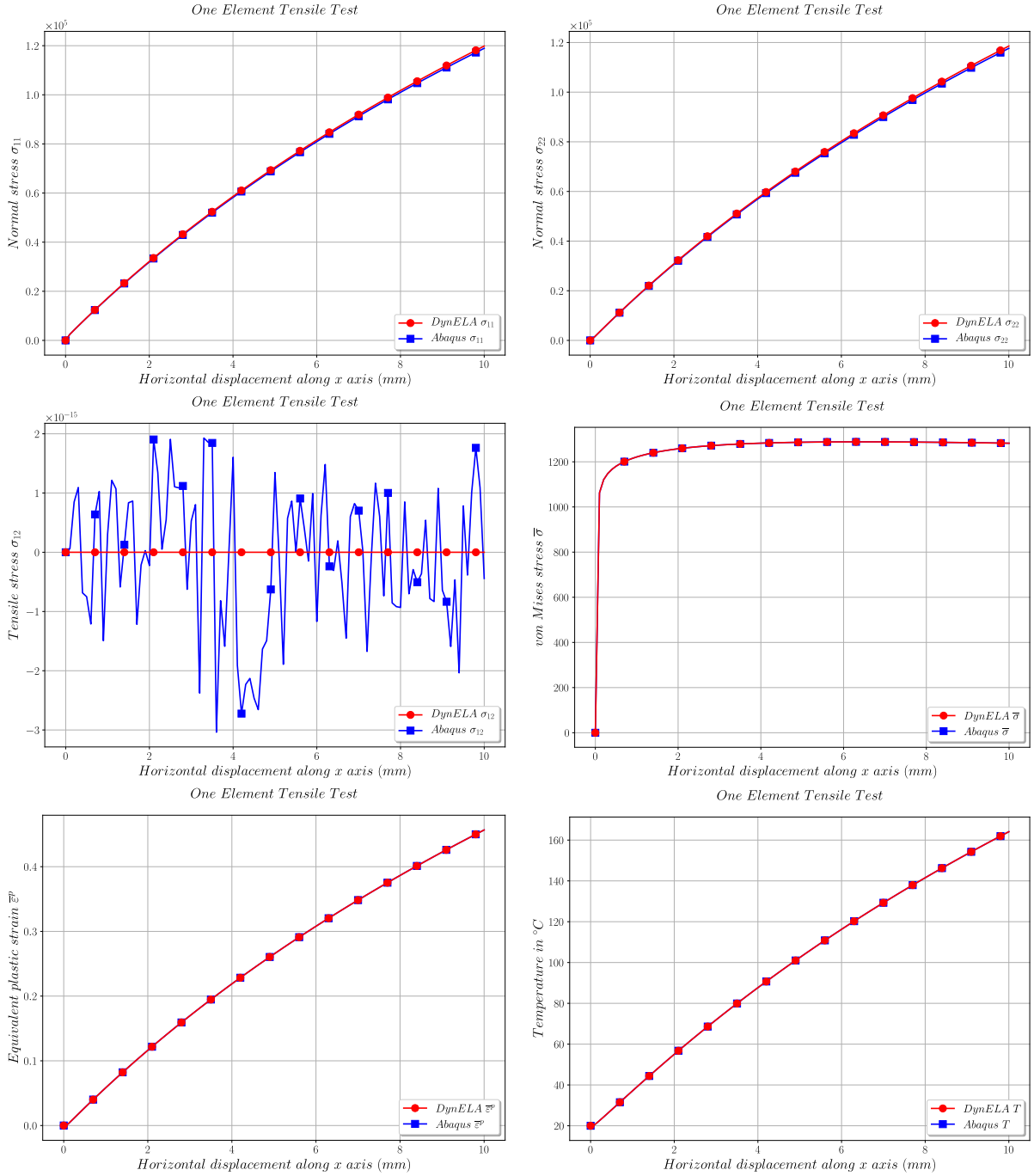


Figure III.1.2: Comparison of numerical and analytical results for the one element tensile test

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 $t = 0.01\,s$.

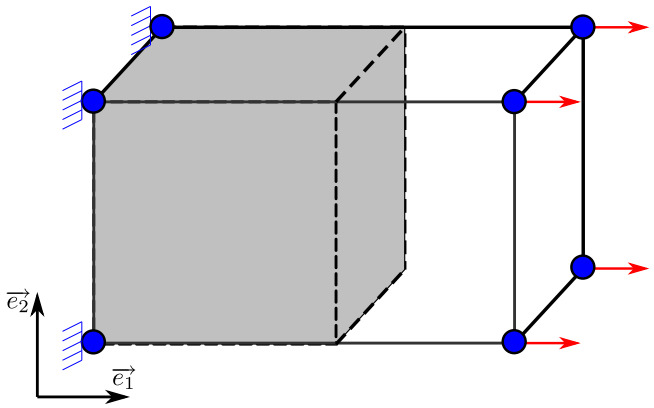


Figure III.1.3: Numerical model for the one element tensile test

All the properties of the constitutive law reported in Table III.1.1 are used and the material is assumed to follow the Johnson-Cook behavior described by equation 1.1.

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.1.1 are used and the material is assumed to follow the Johnson-Cook behavior described by equation 1.1.

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.

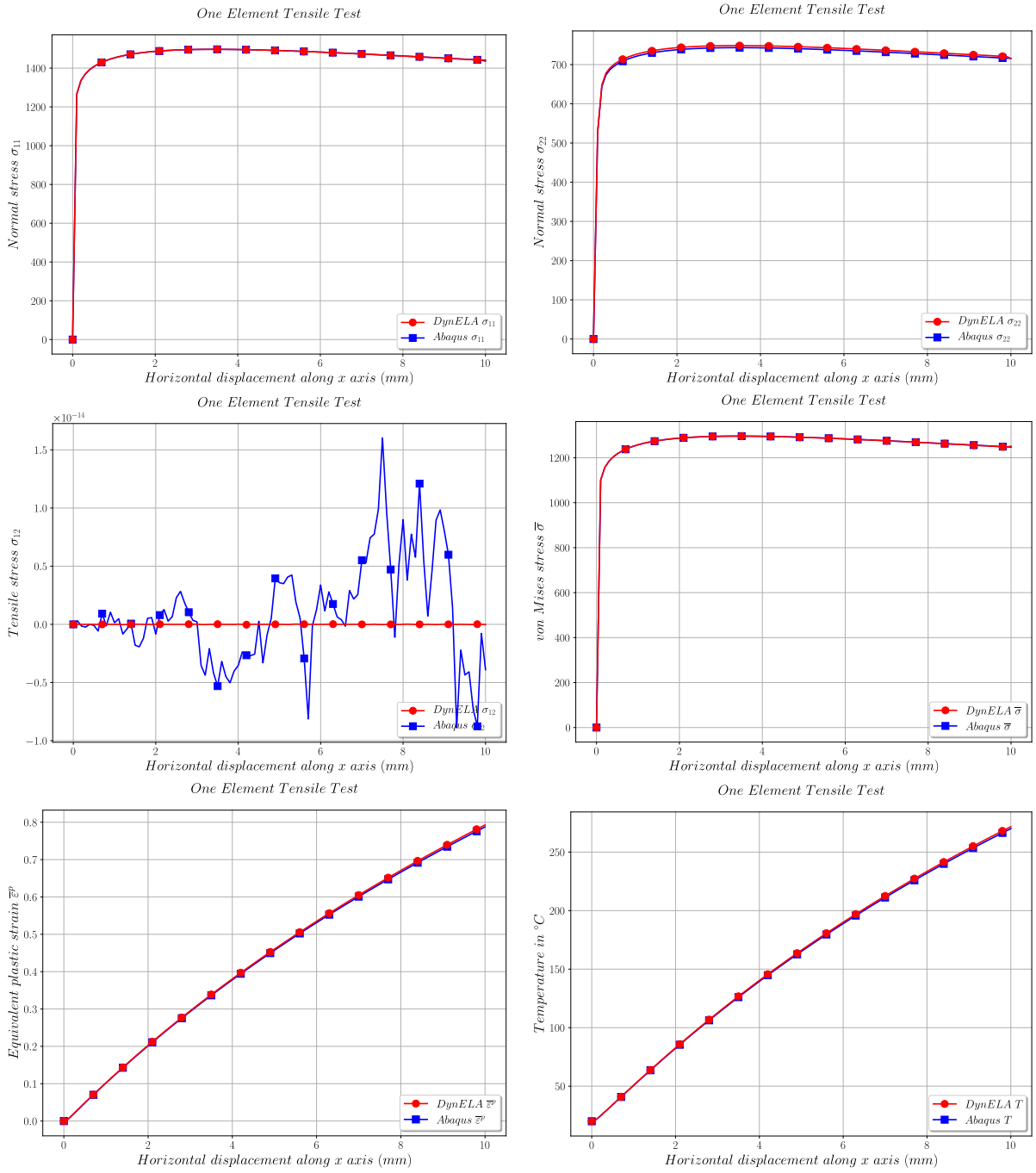


Figure III.1.4: Comparison of numerical and analytical results for the 3D one element tensile test

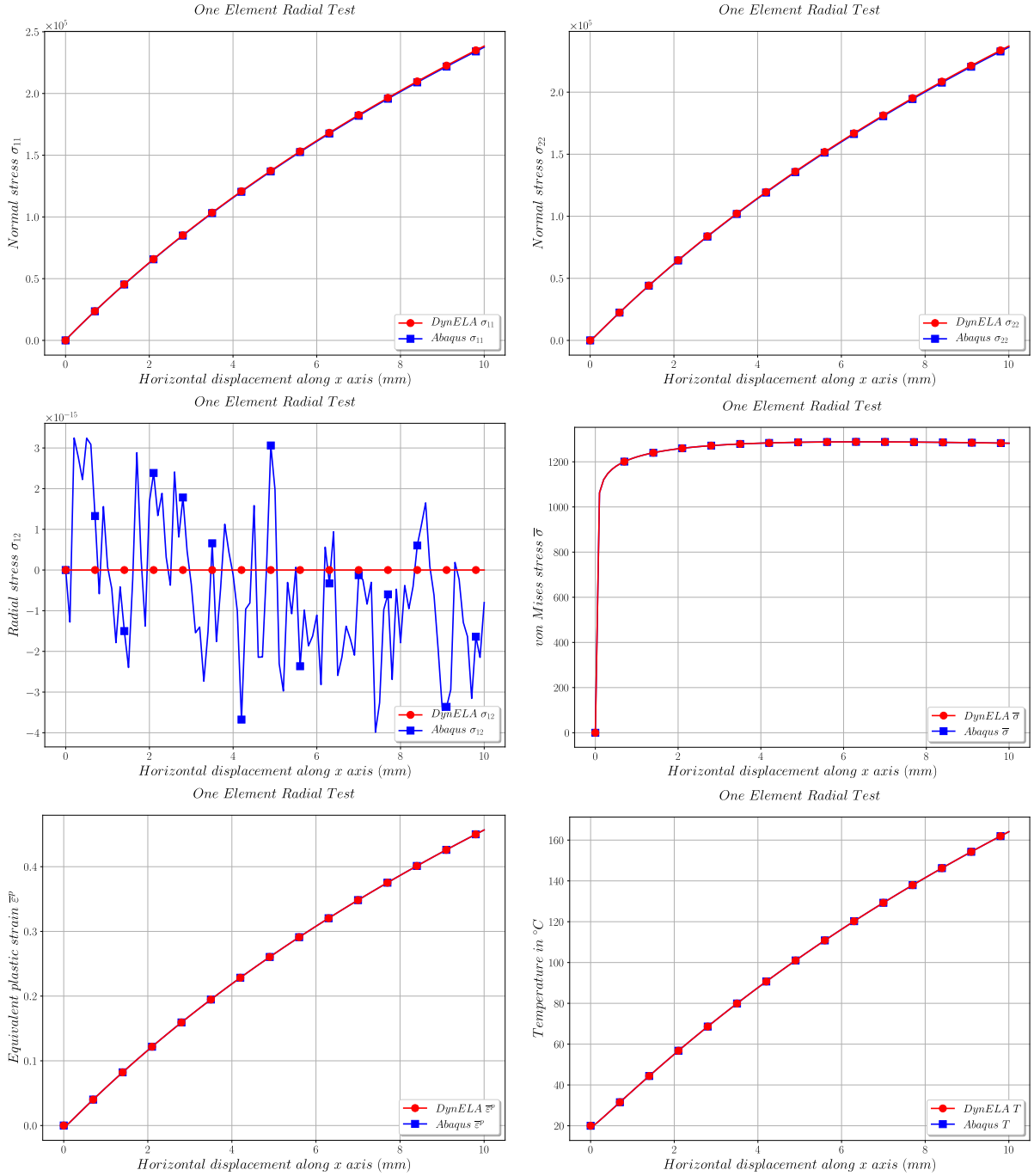


Figure III.1.5: Comparison of numerical and analytical results for the one element radial tensile test

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

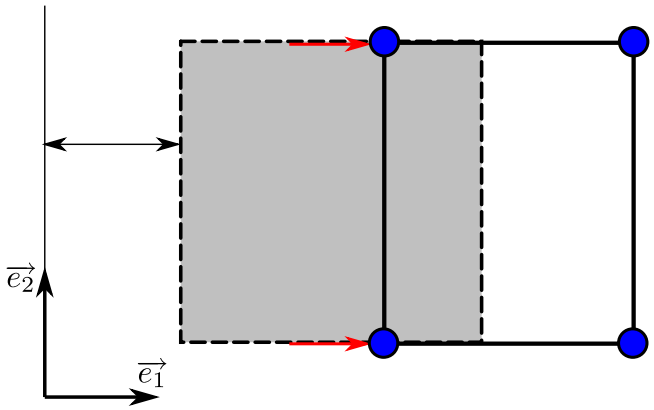


Figure III.1.6: Numerical model for the one element torus test

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.1.1 are used and the material is assumed to follow the Johnson-Cook behavior described by equation 1.1.

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.

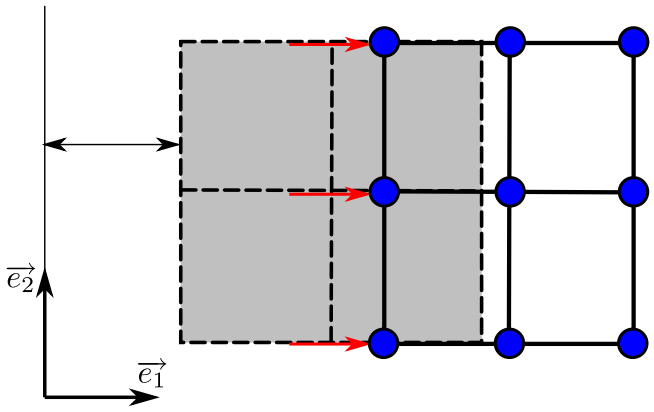


Figure III.1.8: Numerical model for the one element torus test under Abaqus

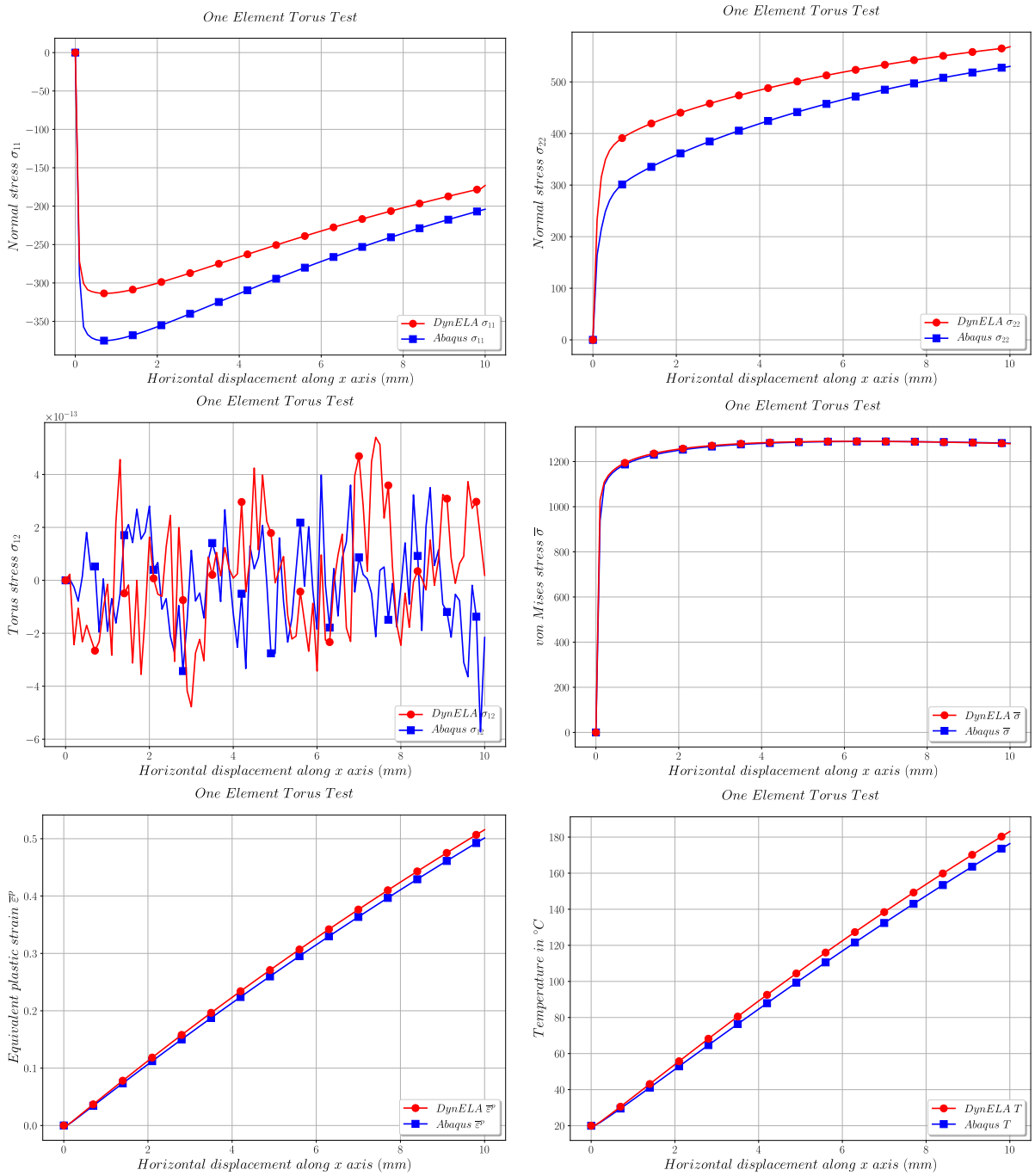


Figure III.1.9: Comparison of numerical and analytical results for the one element torus tensile test

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

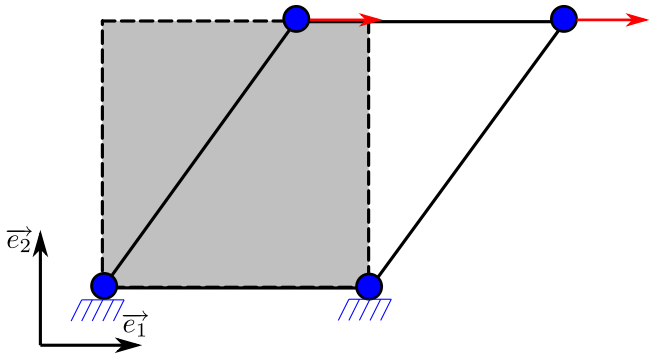


Figure III.1.10: Numerical model for the one element shear test

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.1.1.

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.1.1 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.

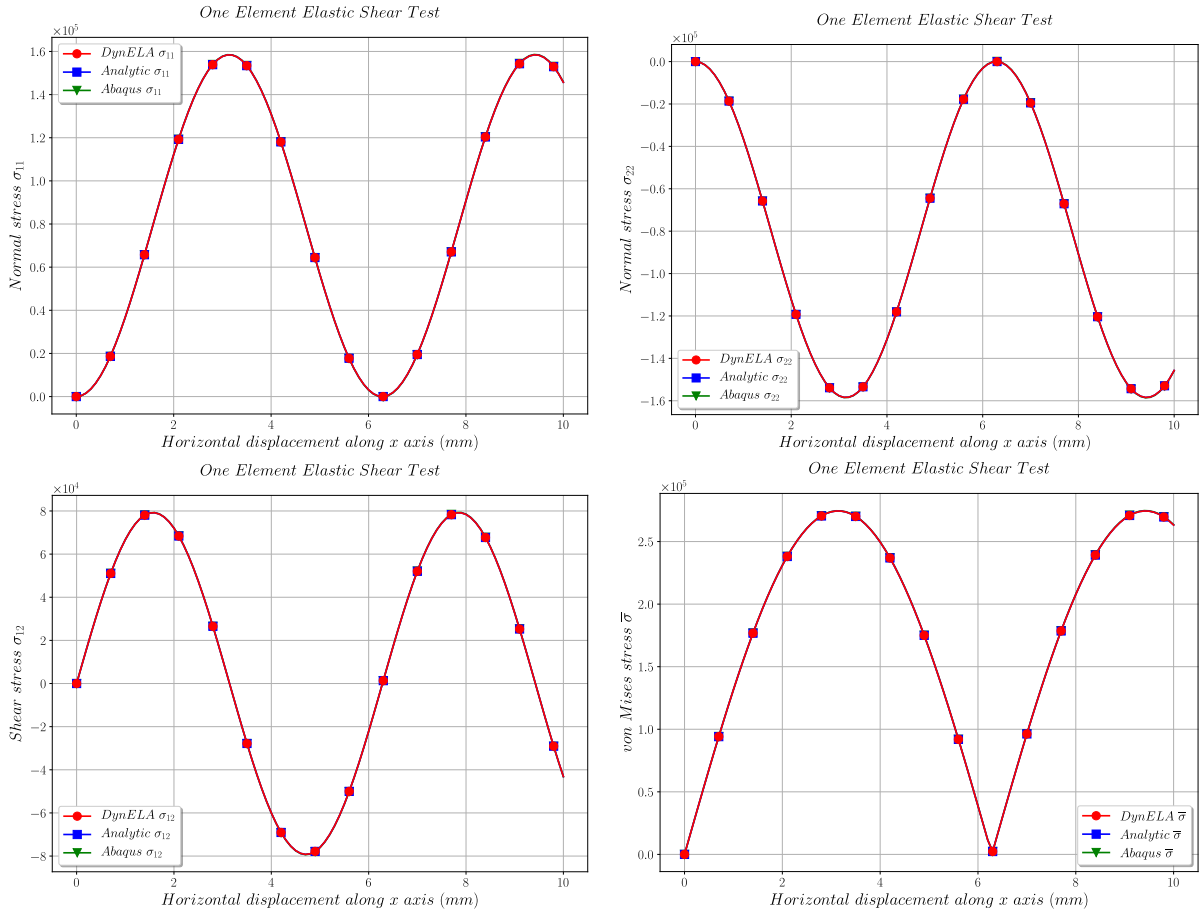


Figure III.1.11: Comparison of numerical and analytical results for the one element elastic shear test

1.2.2 Johnson-Cook plastic behaviour

In this case, all the properties of the constitutive law reported in Table III.1.1 are used and the material is assumed to follow the Johnson-Cook behavior described by equation 1.1. 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.

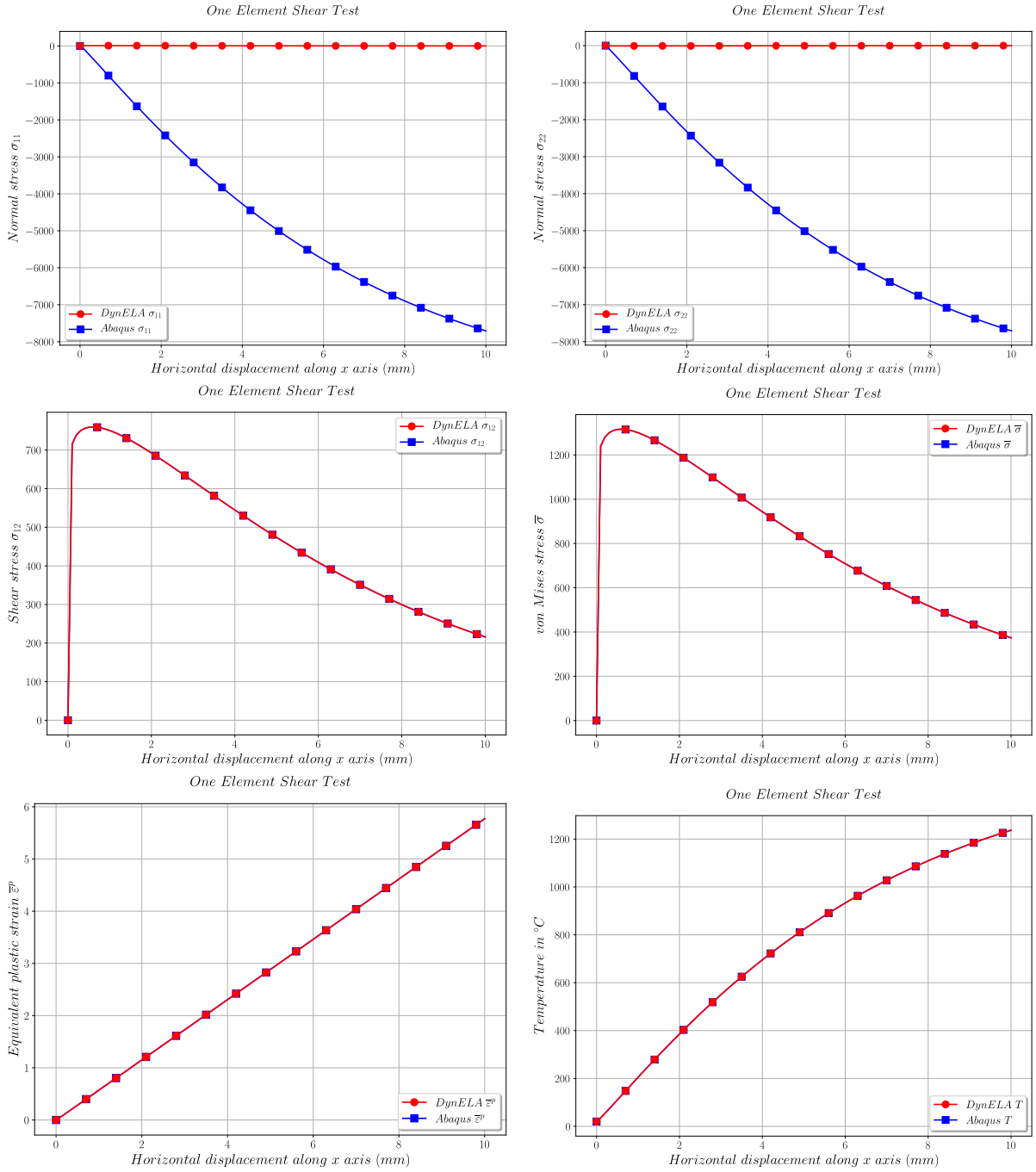


Figure III.1.12: Comparison of numerical and analytical results for the one element shear test

[1]

Bibliography

[1] 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. 39