

**A two–step Taylor–Galerkin  
algorithm applied to  
Lagrangian dynamics**

---

**X M Carreira**

**UNIVERSITY OF WALES SWANSEA**

## UNIVERSITY OF WALES SWANSEA

*Date*            October 2006  
*Author*        Xosé Manuel Carreira Rodríguez  
*Supervisor*   Prof. Javier Bonet  
*Title*            A two–step Taylor–Galerkin algorithm applied to  
                    Lagrangian dynamics  
*Department*   Civil Engineering  
*Degree*        M.Sc.

THE AUTHOR ASSURES THAT THE THESIS AND THE SOFTWARE WERE PRODUCED INDEPENDENTLY AND NO SOURCES OR ASSISTANCE OTHER THAN THOSE INDICATED WERE USED.

THE AUTHOR ATTESTS THAT PERMISSION HAS BEEN OBTAINED FOR THE USE OF ANY COPYRIGHTED MATERIAL APPEARING IN THIS THESIS (OTHER THAN BRIEF EXCERPTS REQUIRING ONLY PROPER ACKNOWLEDGEMENT IN SCHOLARLY WRITING) AND THAT ALL SUCH USE IS CLEARLY ACKNOWLEDGED.

PERMISSION IS GRANTED TO COPY, DISTRIBUTE AND MODIFY THE THESIS AND THE SOFTWARE FOR NON–COMMERCIAL USE WITH THE PROPER ACKNOWLEDGEMENT.

Signature of the author,

# Index

<b>Abstract</b>	<b>4</b>
<b>Acknowledgements</b>	<b>6</b>
<b>1. INTRODUCTION</b>	<b>8</b>
<b>2. THE CONTINUOUS PROBLEM</b>	<b>9</b>
2.1 Preliminaries	9
2.2 Governing equations	10
2.3 Variationally consistent origin of the equations	11
2.4 Constitutive model	13
2.5 Compact wave equation	14
2.6 Shock waves and Rankine–Hugoniot jump conditions	15
2.7 Eigenvalue structure	17
<b>3. THE DISCRETIZED PROBLEM</b>	<b>20</b>
3.1 Necessity of a high order conservative method	20
3.2 Possible numerical approaches	21
3.3 Shock capturing vs. front tracking	22
3.4 A first fractional idea	22
3.5 Time discretization (Taylor)	23
3.6 Spatial discretization (Galerkin)	24
3.7 Conservative properties of the scheme	26
3.8 Involutions	26
3.9 Internal energy	26
3.10 The entropy-like energy variable	27
3.11 The incompressible problem	28
3.12 Error in pressure	28
<b>4. ELEMENT TYPES</b>	<b>29</b>
4.1 Linear bar element	29
4.2 Constant strain linear triangles	33
4.3 Constant strain linear quadrilaterals	38

<b>5. VISCOUS FORMULATION</b>	<b>39</b>
5.1 Motivation	39
5.2 Viscous constitutive model	40
5.3 Conservation–Law with viscous terms	42
<b>6. STABILITY AND CONVERGENCE</b>	<b>44</b>
6.1 CFL condition	44
6.2 Stability criterion	44
6.3 Generalization of the stability condition	46
6.4 Convergence	46
<b>7. EXAMPLES</b>	<b>47</b>
7.1. Bar axially loaded	47
7.2. Rotating plates	51
7.2.1. Rectangular plate	51
7.2.2. Triangular plate	54
7.3 Deformation of a tube	57
7.4. Deformed plate with analytical solution	60
7.5. Buckling of a beam	64
7.6. Optimum viscosities	65
7.6.1. 1-D bar problem	68
7.6.2. 2-D bar problem	71
7.7. Bending of an infinite rectangular plate	71
7.8. Flexible foundation	74
<b>8. CONCLUSIONS</b>	<b>80</b>
<b>REFERENCES</b>	<b>81</b>
<b>APPENDIX: DESIGN AND IMPLEMENTATION OF THE CODE</b>	<b>84</b>
I. Algorithm	84
II. Diagram of procedures and functions	86
III. Truss 1-D code	87
IV. Triang 2-D code	101
V. Quad 2-D code	135

# Abstract

Several industrial applications involve high velocity dynamics of solids, for example forging, machining, crash-tests, collision modelling and many others. These problems involve large deformations and a complex material behaviour. A small-time step and a large mesh are required to model rapid dynamics accurately.

Explicit methods are the most appropriate ones to simulate fast impacts but not all of them have the ability to conserve mass, momentum (linear and angular) and energy. Methods which do not have good conservation properties develop large errors under long time-integrations. In structural dynamics, solids are usually modelled with a Lagrangian mesh and, in this case, mass conservation is automatically satisfied.

This dissertation is structured in eight chapters and an appendix. The first part of the thesis (Chapters 1 to 6) is devoted to present the theoretical background of the two-step Taylor-Galerkin algorithm in terms of linear momentum and stresses. The proposed method is momentum conservative. Energy fluctuations are found to be minimal and stable for long time. The method can be applied to solid dynamic problems that require good resolution of small wavelengths, such as high velocity impacts. It is both fast and accurate when simple linear elements (linear bars in 1-D, constant strain triangles and quadrilaterals in 2-D) are used and it can be an alternative to the classical displacement formulations. We also propose a viscous formulation which is aimed at eliminating the high frequencies in the solution and static solutions can be achieved if they are required.

The exceptional behaviour of the proposed methodology is demonstrated by eight numerical examples in the Chapter n° 7. The Chapter n° 8 summarizes the key conclusions and the future works. Finally, the appendix of this work provides the main details about the design and implementation of the method using an imperative language.

David Steinmann translated Melan's deflection theory into English in 1913 and began using it in 1920. Otmar Amman encouraged engineers to adopt a progressively greater degree of flexibility of stiffening girders. In 1940, four months after the completion, the first Tacoma Narrows bridge collapsed in a moderately strong storm, forcing structural engineers to re-evaluate their reliance on deflection theory.

*Adapted from The tower and the bridge. David P. Billington.  
Princeton paperbacks, 1985.*

# Acknowledgements

First of all, I gratefully acknowledge the kind support and the wise guidance of my supervisor, Prof. Javier Bonet, from the early days of the course. It was a privilege to work with the excellent staff of Swansea University.

I owe recognition to all the professionals from whom I have learnt all I know about structures, continuum mechanics and computing.

I am indebted to Barrié de la Maza Foundation for the generous financial support granted. Special thanks to all the people that gave me assistance to study in the UK: Fernando González Laxe and Montse Orta (Barrié de la Maza Foundation, Spain), Ignasi Colominas, Santiago Hernández Ibáñez and Alfonso Orro (Civil Eng., Univ. of A Coruña, Spain), Nieves Pedreira Souto (Computer Science, Spanish Open Univ.), Alberto Sánchez Martín (IDOM, Spain) and David González Rodríguez (Civil and Env. Eng., M.I.T., USA).

I would specially like to thank my colleagues of the MSc, MRes and PhD programmes, with whom I had many productive discussions. Thank you for all the good times we spent together in this beautiful ugly city.

Finally, I wish to express my gratitude to my parents and to my girlfriend for their unconditional help and motivation.

# Chapter 1

## Introduction

Non-linear finite element analysis is an essential component of nowadays computational mechanics. The field of dynamics of solids encloses a large variety of problems. The combination of classical displacement formulation and implicit methods – Newmark’s family of algorithms [22] is the most popular – gives fast and accurate results for most of the common civil engineering problems; for example stiffness dominated problems such as seismic problems and low velocity impacts.

Nevertheless, in mass dominated problems, for example high velocity impact dynamics, we find phenomena such as the propagation of shock waves and very large deformations that cannot be solved with the classical approach.

The classical approach presents two weak points. The first drawback of the classical displacement formulations is related to propagation of short wavelengths. Errors in the velocity of propagation of these waves together with numerical damping make displacement based finite elements unsuitable for shock propagation problems. The problem of shocks formation and propagation is also present in other areas such as fluid dynamics.

Secondly, low order elements such as triangles in 2-D and tetrahedra in 3-D cannot be used because of volumetric locking and inaccuracy in bending dominated situations.

To avoid these problems a new formulation is created in terms of linear momentums (or velocities) and stresses. The shock wave must be represented for short time steps so the advantage of simplicity and speed of the explicit methods becomes more important than the drawback of conditional stability.

Taylor–Galerkin finite element schemes provide a good compromise between accuracy and speed of computations. The basic Taylor–Galerkin

algorithm was proposed by Donea [3], Zienkiewicz [27] and others for first-order systems of hyperbolic equations.

The method has been refined since its initial formulation. Taylor–Galerkin family of algorithms has been first successfully applied in a wide variety of diffusion problems, fluid dynamics problems. The two–step Taylor–Galerkin was originated at Swansea [15], [16], [24]. The procedure has been used efficiently by Morgan et al [20], [21] in solving electromagnetic wave problems. Nowadays the method is being developed to its fullest for solid dynamics applications [11].

The purpose of this work is to describe the basic lines of a two–step Taylor–Galerkin algorithm formulated in terms of linear momentums and first Piola–Kirchhoff stresses for solid dynamic problems. The proposed interpolation shape functions for both stresses and velocities are linear.

# Chapter 2

## The Continuous Problem

### 2.1. Preliminaries

The motion of a body can be described by a deformation mapping.

$$x_i = x_i(X_j, t) \tag{2.1}$$

$\mathbf{X}$  is the material coordinate in the reference configuration and  $\mathbf{x}$  denotes the position of the particle  $\mathbf{X}$  at time  $t$  in the deformed (current) configuration.

In the reference configuration the volume of the body is  $V_0$  and density  $\rho_0$ . At a given time  $t$ , the body has a volume  $V$  and a density  $\rho$ .

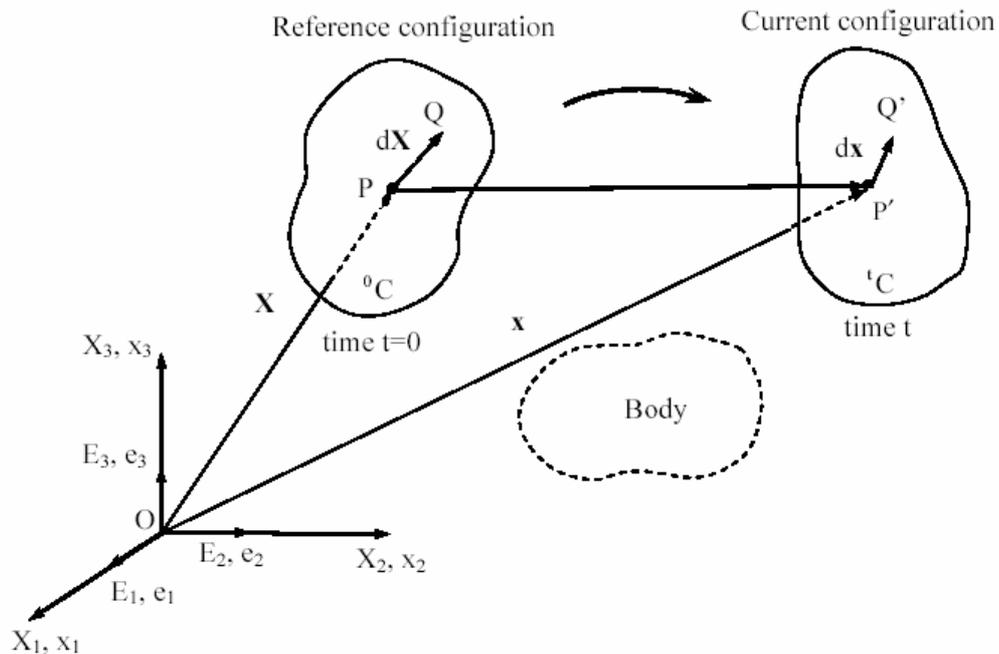


Figure n° 1: Motion of a deformable body

The Jacobian matrix of the transformation gives the deformation gradient tensor  $\mathbf{F}$ .

$$F_{ij} = \frac{\partial x_i}{\partial X_j} \quad (2.2)$$

The determinant of  $\mathbf{F}$  is usually denoted by  $J$  and it relates volume and density in the reference and current configuration.

$$\rho = \frac{\rho_0}{J} \quad (2.3)$$

$$dV = JdV_0 \quad (2.4)$$

The material velocity is given as

$$v_i = v_i(X_j, t) = \frac{\partial x_i(X_j, t)}{\partial t} \quad (2.5)$$

## 2.2. Governing equations

There are three groups of governing equations.

► Conservation of momentum.

$$\frac{\partial p_i}{\partial t} = \frac{\partial P_{ij}}{\partial X_j} \quad (2.6)$$

Where  $i, j=1:3$  for a 3-D Cartesian system of coordinates.

Here  $\mathbf{p}$  is the linear momentum and  $\mathbf{P}$  is the first Piola–Kirchhoff stress tensor.

$$p_i = \rho_0 v_i = \rho_0 \frac{\partial x_i}{\partial t} \quad (2.7)$$

In matrix notation the equation can be written like this,

$$\frac{\partial \mathbf{p}}{\partial t} = \text{DIV}(\mathbf{P}) \quad (2.8)$$

► Deformation–linear momentum relation.

$$\frac{\partial F_{ij}}{\partial t} = \frac{\partial}{\partial X_k} \left( \frac{1}{\rho_0} p_i \delta_{jk} \right) \quad (2.9)$$

Where  $i, j=1:3$  for a 3-D Cartesian system of coordinates.

Here  $\delta$  is the Kronecker delta  $\delta_{jk} = \begin{cases} = 1 \text{ if } j = k \\ = 0 \text{ if } j \neq k \end{cases}$

In matrix notation the equation is (2.10).

$$\frac{\partial \mathbf{F}}{\partial t} = \text{DIV} \left( \frac{\mathbf{p}}{\rho_0} \otimes \mathbf{I} \right) \quad (2.10)$$

► Constitutive model.

$$P_{ij} = \frac{\partial \psi}{\partial F_{ij}} \quad (2.11)$$

$\psi$  is the density of energy per undeformed volume.

### 2.3. Variationally consistent origin of the equations

The origin of the equations is explained in this section. The second governing equation (2.9) can be obtained easily deriving the deformation gradient tensor with respect to time

$$\frac{\partial F_{ij}}{\partial t} = \frac{\partial}{\partial t} \left( \frac{\partial x_i}{\partial X_j} \right) = \frac{\partial}{\partial X_j} \left( \frac{\partial x_i}{\partial t} \right) = \frac{\partial}{\partial X_j} \left( \frac{p_i}{\rho_0} \right) = \frac{\partial}{\partial X_k} \left( \frac{p_i}{\rho_0} \delta_{jk} \right) \quad (2.12)$$

The origin of the first governing equation (2.6) is given by the Hamilton's variational principle. We introduce the Lagrangian  $\mathcal{L}$ ,

$$\mathcal{L} = K - \Pi \quad (2.13)$$

$K$  represents the kinetic energy and  $\Pi$  is the potential energy. The action integral,  $\mathcal{S}$ , is defined as the integral of the Lagrangian over the time interval considered,

$$S = \int_{t_0}^t (K - \Pi) dt \quad (2.14)$$

Hamilton's principle states that the equations of motion can be obtained by making the action integral stationary with respect to all possible motions compatible with the boundary conditions. By stationary, we mean that the action does not vary to first order for infinitesimal deformations. This is  $DS[\delta\mathbf{x}] = 0$  for all compatible  $\delta\mathbf{x}$ . Calculating the linearization of  $S$  and rearranging,

$$DS[\delta\mathbf{x}] = \int_{t_0}^t \int_{V_0} \left( \rho_0 \frac{\partial \mathbf{v}}{\partial t} - \text{DIV}(\mathbf{P}) \right) \delta\mathbf{x} dV_0 dt = 0 \quad \forall \delta\mathbf{x} \quad (2.15)$$

So,

$$\rho_0 \frac{\partial \mathbf{v}}{\partial t} = \text{DIV}(\mathbf{P}) \quad (2.16)$$

And this can be rewritten like this using Einstein notation.

$$\frac{\partial p_i}{\partial t} = \frac{\partial P_{ij}}{\partial X_j} \quad (2.17)$$

### Internal potential energy

If we assume no external potentials, the only potential energy is due to deformation and for a hyperelastic neo-Hookean material this is given by the following function of the density of energy per undeformed volume:

$$\psi(\mathbf{F}) = \frac{1}{2} \mu [J^{-2/3} (\mathbf{F} : \mathbf{F}) - 3] + \frac{1}{2} \kappa (J - 1)^2 \quad (2.18)$$

So, the internal potential energy is the integral of the density of energy per undeformed volume in the reference domain  $\Omega$ .

$$\Pi_{INT} = \int_{\Omega} \psi(\mathbf{F}) d\Omega \quad (2.19)$$

Sometimes, it is convenient to split the potential into two parts, the volumetric and the deviatoric energy.

$$\Pi_{INT} = \Pi_{VOL} + \Pi_{DEV} \quad (2.20)$$

$$\Pi_{VOL} = \frac{\kappa}{2} \int_{\Omega} (J-1)^2 d\Omega \quad (2.21)$$

$$\Pi_{DEV} = \frac{\mu}{2} \int_{\Omega} [J^{-2/3} (\mathbf{F} : \mathbf{F}) - 3] d\Omega \quad (2.22)$$

### External potential energy

The external potential energy includes the work done by the external body forces  $\mathbf{f}_{BODY}$  and surface forces  $\mathbf{f}_{SURF}$ .

$$\Pi_{EXT} = \int_{\Omega} \mathbf{f}_{BODY} \cdot \mathbf{x} d\Omega + \int_{\Gamma} \mathbf{f}_{SURF} \cdot \mathbf{x} d\Gamma \quad (2.23)$$

Here,  $\mathbf{x}$  is the material position.  $\Gamma$  symbolizes the boundary of the domain in the reference configuration. In this project, no external body forces were considered.

### Kinetic energy

The total kinetic energy of the system is given by

$$K = \frac{1}{2} \int_{\Omega} \rho_0 \|\mathbf{v}\|^2 d\Omega = \frac{1}{2\rho_0} \int_{\Omega} \|\mathbf{p}\|^2 d\Omega \quad (2.24)$$

## 2.4. Constitutive model

The first Piola–Kirchhoff stress tensor is obtained from the density of energy function. If a Neo–Hookean model of the material is assumed, the stresses can be computed with the equation (2.26).

$$P_{ij} = \frac{\partial \psi}{\partial F_{ij}} \quad (2.25)$$

$$\mathbf{P} = \mu J^{-2/3} \left[ \mathbf{F} - \frac{1}{3} (\mathbf{F} : \mathbf{F}) \mathbf{F}^{-T} \right] + \kappa (J-1) J \mathbf{F}^{-T} \quad (2.26)$$

The stress has two parts: the volumetric (2.27) and the deviatoric part (2.28).

$$\mathbf{P}_{VOL} = \kappa (J-1) J \mathbf{F}^{-T} \quad (2.27)$$

$$\mathbf{P}_{DEV} = \mu J^{-2/3} \left[ \mathbf{F} - \frac{1}{3} (\mathbf{F} : \mathbf{F}) \mathbf{F}^{-T} \right] \quad (2.28)$$

The elastic coefficients are

$$\mu = \frac{E}{2(1+\nu)} \quad \kappa = \frac{E}{3(1-2\nu)} \quad (2.29)$$

For engineering design purposes, the results obtained can be output in terms of the Cauchy stress (2.30) and Almansi strain (2.31) tensors.

$$\boldsymbol{\sigma} = J \mathbf{P} \mathbf{F}^{-T} \quad (2.30)$$

$$\mathbf{e} = \frac{1}{2} (\mathbf{I} - \mathbf{F}^{-T} \mathbf{F}^{-1}) \quad (2.31)$$

## 2.5. Compact wave equation form

If we define the following two vectors, it is possible to put the first two governing equations (2.6) and (2.9) together in one equation (2.32).

$$\mathbf{U} = \begin{bmatrix} (\mathbf{p}) \\ (\mathbf{F}) \end{bmatrix} \quad \text{the vector of the unknowns}$$

$$\mathfrak{S} = - \begin{bmatrix} (\mathbf{P}) \\ \left( \frac{\mathbf{P}}{\rho_0} \otimes \mathbf{I} \right) \end{bmatrix} \quad \text{the matrix of the fluxes of the unknowns}$$

$$\frac{\partial U_i}{\partial t} + \frac{\partial \mathfrak{S}_{ij}}{\partial X_j} = 0 \quad (2.32)$$

This is our conservation-law formulation, which is similar to the equation of a wave. For 3-D the vectors have the structure shown in (2.33).

The main advantages of the compact form are two:

- The discretization is applied only to one equation instead of two expressions.
- The formulation is simpler because all the unknowns are together and it is more general. It could be used again for problems with a similar equation (i.e. diffusion problems, fluid dynamics...) by particularizing the content of the vector of unknowns and the vector of fluxes.

$$[U_i] = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ F_{11} \\ F_{12} \\ F_{13} \\ F_{21} \\ F_{22} \\ F_{23} \\ F_{31} \\ F_{32} \\ F_{33} \end{bmatrix} \quad [S_{ij}] = - \begin{bmatrix} P_{11} & P_{12} & P_{13} \\ P_{21} & P_{22} & P_{23} \\ P_{31} & P_{32} & P_{33} \\ p_1/\rho_0 & 0 & 0 \\ 0 & p_1/\rho_0 & 0 \\ 0 & 0 & p_1/\rho_0 \\ p_2/\rho_0 & 0 & 0 \\ 0 & p_2/\rho_0 & 0 \\ 0 & 0 & p_2/\rho_0 \\ p_3/\rho_0 & 0 & 0 \\ 0 & p_3/\rho_0 & 0 \\ 0 & 0 & p_3/\rho_0 \end{bmatrix} \quad (2.33)$$

Lax [12] was the first author to introduce the use of conservation form and he showed that the numerical schemes that follow this conservative form are able to represent discontinuities in the flow variables (shock waves).

It is also important to remark that the conservation of energy is not imposed although this methodology is found to have a good energy conservation property without a clear reason for it.

## 2.6. Shock waves and Rankine–Hugoniot jump conditions

Shock waves form an important class of solution to the elastodynamic equations. We will study these waves under the assumptions of homogeneity and isotropy and in the absence of body forces. The shock waves happen when the unknowns  $\mathbf{U}$  are discontinuous.

The original wave equation can be rewritten using the chain rule.

$$\frac{\partial U_i}{\partial t} + \frac{\partial \mathfrak{S}_{ij}}{\partial X_j} = 0 \quad (2.34)$$

$$\frac{\partial U_i}{\partial t} + \frac{\partial \mathfrak{S}_{ij}}{\partial U_k} \frac{\partial U_k}{\partial X_j} = 0 \quad (2.35)$$

$$\frac{\partial U_i}{\partial t} + A_{ijk} \frac{\partial U_k}{\partial X_j} = 0 \quad (2.36)$$

The tensor  $\mathbf{A}$  is usually called the *acoustic tensor* [6]. The system is quasi-linear if  $\mathbf{A}$  is constant and the system is hyperbolic if  $\mathbf{A}$  is diagonalizable. If  $\mathbf{A}$  were diagonal we could put the governing equation in the characteristic form (2.37).

$$\frac{\partial U_i}{\partial t} + \lambda_i \delta_{ijk} \frac{\partial U_k}{\partial X_j} = 0 \quad (2.37)$$

$$\delta \text{ is the Kronecker delta } \delta_{ijk} = \begin{cases} = 1 & \text{if } i = j = k \\ = 0 & \text{in other case} \end{cases}$$

This is now a set of independent equations and the solution of the equation is constant along the characteristic curves.

At a given interface defined by the normal  $\mathbf{N}$ , the resulting flux–Jacobian is given by

$$\mathbf{AN} = \frac{\partial(\mathfrak{S}\mathbf{N})}{\partial \mathbf{U}} = \begin{bmatrix} \frac{\partial \mathbf{P}}{\partial \mathbf{p}} & \frac{\partial \mathbf{P}}{\partial \mathbf{F}} \\ \frac{\partial \left( \frac{\mathbf{p}}{\rho_0} \otimes \mathbf{I} \right)}{\partial \mathbf{p}} & \frac{\partial \left( \frac{\mathbf{p}}{\rho_0} \otimes \mathbf{I} \right)}{\partial \mathbf{F}} \end{bmatrix} \mathbf{N} = \begin{bmatrix} \mathbf{0} & \frac{\partial \mathbf{P}}{\partial \mathbf{F}} \mathbf{N} \\ \frac{\mathbf{I} \otimes \mathbf{N}}{\rho_0} & \mathbf{0} \end{bmatrix} \quad (2.38)$$

The Riemann problem is simply the hyperbolic wave equation with special initial data. The data are piecewise constant with a single jump discontinuity at some position  $X_i$ .

$$\mathbf{U} = \begin{cases} \mathbf{U}^+ \\ \mathbf{U}^- \end{cases}$$

For linear hyperbolic systems the Riemann problem is solved in terms of the eigenvalues and eigenvectors of the acoustic tensor  $\mathbf{AN}$ .

$$\mathbf{AN}[U^+ - U^-] + c[U^+ - U^-] = \mathbf{0} \quad (2.39)$$

The Rankine–Hugoniot jump conditions above presented are obtained directly from the local conservation equations adding the jump. Developing the equation (2.39) we have:

$$\begin{aligned} [\mathbf{P}^+ - \mathbf{P}^-]N + c[\mathbf{p}^+ - \mathbf{p}^-] &= \mathbf{0} \\ \frac{1}{\rho_0}[\mathbf{p}^+ - \mathbf{p}^-] \otimes N + c[\mathbf{F}^+ - \mathbf{F}^-] &= \mathbf{0} \end{aligned} \quad (2.40)$$

If we include the entropy variable in the system the additional jump equation would be:

$$[\mathbf{P}^{T+} \mathbf{p}^+ - \mathbf{P}^{T-} \mathbf{p}^-]N + c[\mathbf{E}^+ - \mathbf{E}^-] = \mathbf{0} \quad (2.41)$$

In other words, singularities can only propagate along characteristics for a linear system.

## 2.7. Eigenvalue structure

The first two orthogonal eigenvectors of  $\mathbf{AN}$  will give the longitudinal and the transverse shock wave directions. The associated eigenvalues are the celerity  $c$  of the shock wave. The eigensystem is

$$\mathbf{AN}(\delta U) = -c(\delta U) \quad (2.42)$$

The symbol  $\delta$  refers to the change (jump) of the variable before and after the discontinuity.

The left hand side term in (2.42) is

$$\mathbf{AN}(\delta U) = \begin{bmatrix} (\underline{\mathcal{C}} : \delta \mathbf{F})N \\ \frac{\delta \mathbf{p} \otimes N}{\rho_0} \end{bmatrix} \quad (2.43)$$

Where  $\underline{\mathcal{C}}$  is the elastic constitutive tensor (2.44).

$$C_{ijkl} = \frac{\partial P_{ij}}{\partial F_{kl}} = \lambda \delta_{ij} \delta_{kl} + \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) \quad (2.44)$$

$\lambda$  and  $\mu$  are the Lamé elastic coefficients

$$\lambda = \frac{\nu E}{(1+\nu)(1-2\nu)} \quad \mu = \frac{E}{2(1+\nu)} \quad (2.45)$$

$E$  and  $\nu$  are the Young modulus and the Poisson coefficient.

The eigenvectors  $\mathbf{U}$  have the structure of the previously described vectors of unknowns. The system is called hyperbolic if the matrix  $\mathbf{AN}$  is diagonalizable with real eigenvalues. If  $\mathbf{P}$  describes a Neo–Hookean material, the two main eigenvalues of the system are

$$c_L = \sqrt{\frac{\lambda + 2\mu}{\rho_0}} \quad \text{for the longitudinal (or acoustic, or P) wave.} \quad (2.46)$$

$$c_T = \sqrt{\frac{\mu}{\rho_0}} \quad \text{for the transversal (or shear, or S) wave.} \quad (2.47)$$

It is possible to find two families of mechanisms that give a null eigenvalue ( $c = 0$ ).

$$\text{First mechanism:} \quad \delta \mathbf{p} = \mathbf{0} \quad \text{and} \quad \delta \mathbf{F} = -\delta \mathbf{F}^T \text{ skew symmetric} \quad (2.48)$$

$$\text{Second mechanism:} \quad \delta \mathbf{p} = \mathbf{0} \quad \text{and} \quad \delta \mathbf{F} \text{ leads to plane stress} \quad (2.49)$$

For the mechanism presented in (2.49)  $\delta \mathbf{F}$  must be such that

$$\underline{\mathbf{C}} : \delta \mathbf{F} = \langle T_1 \otimes T_1, T_2 \otimes T_2, T_1 \otimes T_2 + T_2 \otimes T_1 \rangle \quad (2.50)$$

Where  $\mathbf{T}_1 \cdot \mathbf{N} = \mathbf{T}_2 \cdot \mathbf{N} = \mathbf{T}_3 \cdot \mathbf{N} = \mathbf{0}$  and  $\langle \rangle$  means a linear combination of vectors.

In the following figure we illustrate qualitatively how different parts of the one–dimensional waves moving with velocities proportional to their amplitude finally develop into a shock wave. A similar figure can be found in [27].

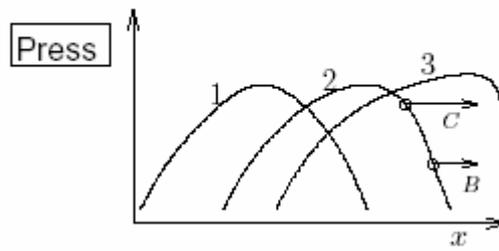


Figure n° 2: Progression of a 1-D wave and creation of a shock [4]

In a linear problem, all the points on the wave move at the same speed. For a nonlinear problem, the wave advects itself such the local speed depends on the wave amplitude. The process is called *nonlinear steepening* and eventually results in shock waves. A fundamental feature of nonlinear compared to linear conservation laws is that discontinuities can be developed even from smooth initial data.

## Chapter 3

# The Discretized Problem

### 3.1. Necessity of a high order conservative method

The governing equation can be expressed in the form (3.1) where the function  $f$  contains the terms with the spatial derivatives.

$$\frac{\partial U}{\partial t} = f(\mathfrak{I}) \quad (3.1)$$

The simplest idea is to

1. Take initial state of unknowns given on a grid.
2. Compute the flux.
3. Compute the spatial derivatives to get the right-hand side using finite differences.
4. Get the small change of the unknowns.
5. Update the unknowns.
6. Restart at step 2

Even for short time steps the explicit Euler method leads to conflicts because of the growing oscillations. Discontinuities lead to computational difficulties. Finite difference methods in which derivatives are approximated directly by finite differences can be expected to break near discontinuities.

Conservativeness can be used to check if the code is correct and conservative methods give better solutions than the schemes that are known to conserve poorly.

### 3.2. Possible numerical approaches

There are four principal families of numerical methods to solve numerically the hyperbolic problem. The main differences are summarized here:

a) Spectral methods:

- Representation by a finite set of harmonical functions (sinusoidal waves, wavelets, etc).
- Good for flows with small non-linear interactions.
- Typically used for flows with low Mach numbers.

b) Smoothed Particle Hydrodynamics (SPH):

- Representation by a finite set of particles that move freely.
- Grid is replaced by particle positions.
- Particle density translates into fluid density

c) Finite volume methods:

- Restriction: integration over control volume.
- Construction and reconstruction by polynomials.
- Derivatives can become finite differences or can be avoided.

d) Finite element methods:

- Representation by a finite set piecewise polynomials.
- Usually used on an unstructured grid to model the flow around complex bodies.
- Often used in engineering.

The main types of grids are four:

- Hybrid grid: moving with another speed.
- No grid: SPH (grid is replaced by particle positions).
- Eulerian grid: fixed in space.
- Lagrangian grid: moving with the body (makes the linear 1-D problem trivial).

The aim of this dissertation is to develop a finite element method approach (a two-step Taylor– Galerkin) with a Lagrangian mesh. Finite element methods work with elements instead of cells (finite volume) or points (finite difference). The primary problem is to obtain good numerical flux solutions that approximate the real fluxes reasonably well. Conservative high order methods such as two-step Taylor–Galerkin are effective for computing discontinuous solutions.

### 3.3. Shock capturing vs. front tracking

#### Front tracking

The positions of shocks are explicitly followed to allow a different numerical treatment in smooth regions and near discontinuities. A common finite volume method in smooth regions is combined with a high order explicit procedure for tracking the location of discontinuities.

The governing equations are supplemented with jump conditions (2.40) across discontinuities. These algorithms are complicated in more than one space dimension because discontinuity surfaces can interact and evolve in complicated ways.

#### Shock capturing

The aim is to capture discontinuities in the solution automatically, without explicit tracking them. Discontinuities are spread over one or more elements. This kind of scheme was chosen because it is simpler to implement than a front tracking method.

### 3.4. A first fractional idea

A possible explicit two-step procedure could be (3.2) and (3.3).

#### Predictor step:

$$\delta U^{n+\frac{1}{2}} = \delta U^n + \frac{1}{2} M^{-1} f^n \quad (3.2)$$

#### Corrector step:

$$\delta U^{n+1} = \delta U^n + \Delta t M^{-1} f^{n+\frac{1}{2}} \quad (3.3)$$

This would be an equivalent method to second order Runge–Kutta algorithm for ODEs extended to PDEs. However, it was found [27] that the

numerical results were inaccurate after some time due to overdiffusion. The two-step Taylor-Galerkin algorithm is based on this fractional idea but it balances convection and diffusion, what makes it suitable for long time simulations.

### 3.5. Time discretization (Taylor)

A Taylor expansion in time yields

$$U_i^{n+1} = U_i^n + \Delta t \left[ \frac{\partial U_i}{\partial t} \right]^n + \frac{\Delta t^2}{2} \left[ \frac{\partial^2 U_i}{\partial t^2} \right]^n + O(\Delta t^3) \quad (3.4)$$

With  $t_{n+1} = t_n + \Delta t$  and  $U_i^{n+1} = U_i(X_j, t_n)$

Using the wave equation to replace the time derivatives by space derivations and ignoring the third order term we can write

$$U_i^{n+1} = U_i^n - \Delta t \left[ \frac{\partial \mathfrak{S}_{ij}}{\partial X_j} \right]^n - \frac{\Delta t^2}{2} \left[ \frac{\partial}{\partial t} \left( \frac{\partial \mathfrak{S}_{ij}}{\partial X_j} \right) \right]^n \quad (3.5)$$

Or with a more compact notation

$$U_i^{n+1} = U_i^n - \Delta t \frac{\partial \mathfrak{S}_{ij}^n}{\partial X_j} - \frac{\Delta t^2}{2} \frac{\partial}{\partial t} \left( \frac{\partial \mathfrak{S}_{ij}^n}{\partial X_j} \right) \quad (3.6)$$

In the first step we use the following Taylor expansion. From here we recalculate the spatial derivative of the flux (3.8).

$$U_i^{n+\frac{1}{2}} = U_i^n - \frac{\Delta t}{2} \frac{\partial \mathfrak{S}_{ij}^n}{\partial X_j} \quad (3.7)$$

$$\frac{\partial \mathfrak{S}_{ij}^n}{\partial X_j} = -2 \frac{U_i^{n+\frac{1}{2}} - U_i^n}{\Delta t} \quad (3.8)$$

From the wave equation (2.32) we obtain the following operator

$$\frac{\partial}{\partial t} = -\frac{\partial \mathfrak{S}_{ij}^n}{\partial X_j} \frac{\partial}{\partial U_i} = -\frac{\partial \mathfrak{S}_{ij}^n}{\partial U_i} \frac{\partial}{\partial X_j} = -A_j \frac{\partial}{\partial X_j} \quad (3.9)$$

We also define the Jacobian vectors as

$$A_j = \frac{d\mathfrak{S}_{ij}^n}{dU_i} \quad (3.10)$$

So,

$$\frac{\partial}{\partial t} \left( \frac{\partial \mathfrak{S}_{ij}^n}{\partial X_j} \right) = -A_k^n \frac{\partial}{\partial X_k} \left( \frac{\partial \mathfrak{S}_{ij}^n}{\partial X_j} \right) \quad (3.11)$$

Hence,

$$U_i^{n+1} = U_i^n - \Delta t \frac{\partial \mathfrak{S}_{ij}^n}{\partial X_j} + \frac{\Delta t^2}{2} A_k^n \frac{\partial}{\partial X_k} \left( \frac{\partial \mathfrak{S}_{ij}^n}{\partial X_j} \right) \quad (3.12)$$

For 3-D the indices are  $i=1:12$ ,  $j=1:3$ ,  $k=1:3$ .

In an analogous way we operate over the flux matrix.

$$\mathfrak{S}_{ij}^{n+\frac{1}{2}} = \mathfrak{S}_{ij}^n + \frac{\Delta t}{2} \frac{\partial \mathfrak{S}_{ij}^n}{\partial t} = \mathfrak{S}_{ij}^n - \frac{\Delta t}{2} A_k^n \frac{\partial \mathfrak{S}_{ij}^n}{\partial X_k} \Rightarrow A_k^n \frac{\partial \mathfrak{S}_{ij}^n}{\partial X_k} = -2 \frac{\mathfrak{S}_{ij}^{n+\frac{1}{2}} - \mathfrak{S}_{ij}^n}{\Delta t} \quad (3.13)$$

Then,

$$U_i^{n+1} = U_i^n - \Delta t \frac{\partial \mathfrak{S}_{ij}^n}{\partial X_j} - \Delta t \frac{\partial}{\partial X_j} \left( \mathfrak{S}_{ij}^{n+\frac{1}{2}} - \mathfrak{S}_{ij}^n \right) \quad (3.14)$$

And we obtain the approximation (3.15) with second order accuracy.

$$U_i^{n+1} - U_i^n = -\Delta t \frac{\partial \mathfrak{S}_{ij}^{n+\frac{1}{2}}}{\partial X_j} \quad (3.15)$$

### 3.6. Spatial discretization (Galerkin)

Applying the shape functions as weighting functions in each side of the equation (3.15), we get

$$\int_{\Omega} (U_i^{n+1} - U_i^n) N_c d\Omega = -\Delta t \int_{\Omega} \frac{\partial \mathfrak{S}_{ij}^{n+\frac{1}{2}}}{\partial X_j} N_c d\Omega \quad (3.16)$$

Using Gauss divergence theorem (in 1-D this is just integration by parts) it is possible to separate the contribution of the boundary from the internal contribution.

$$\int_{\Omega} (U_i^{n+1} - U_i^n) N_c d\Omega = -\Delta t \int_{\Gamma} \mathfrak{S}_{ij}^{n+\frac{1}{2}} N_c d\Gamma + \Delta t \int_{\Omega} \mathfrak{S}_{ij}^{n+\frac{1}{2}} \frac{\partial N_c}{\partial X_j} d\Omega \quad (3.17)$$

With the index  $c$  ranging from 1 to  $N_{\text{nodes}}$ .

For the spatial discretization we interpolate using  $C_0$  shape functions:

$$U_i = [U_i]_a N_a \quad \mathfrak{S}_{ij} = [\mathfrak{S}_{ij}]_a N_a \quad (3.18)$$

With the mute index  $a=1: N_{\text{nodes}}$ . Thus, we lead to the general system of discretized equations (3.19). A more compact way is (3.20).

$$\begin{aligned} \int_{\Omega} ([U_i^{n+1}]_a - [U_i^n]_a) N_a N_c d\Omega &= -\Delta t \int_{\Gamma} [\mathfrak{S}_{ij}^{n+\frac{1}{2}}]_a N_a N_c d\Gamma \\ + \Delta t \int_{\Omega} [\mathfrak{S}_{ij}^{n+\frac{1}{2}}]_a N_a \frac{\partial N_c}{\partial X_j} d\Omega \end{aligned} \quad (3.19)$$

$$M\delta U = f \quad U^{n+1} = U^n + \delta U \quad (3.20)$$

Equation (3.7) is usually called the *predictor step* and equation (3.20) is the *corrector step*. In the first step, the balance area of the convective flows for one element have to be calculated on the nodes of each element so, the information goes from the nodes to the element (Fig. 3). In the second step, the balance area for one node is calculated with the help of all elements which are defined with this node so, the information goes from the elements to the nodes.

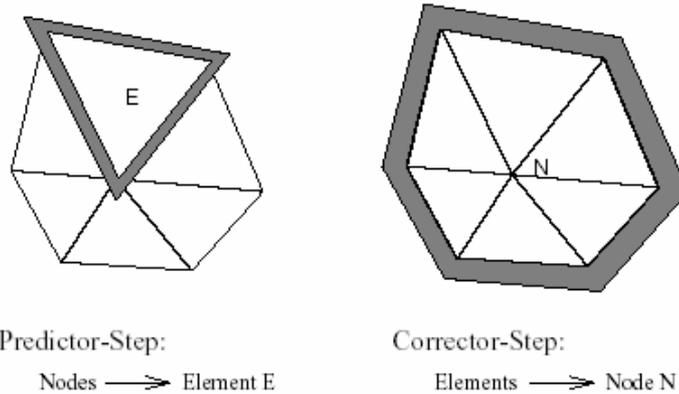


Figure n° 3: Balance areas [14]

### 3.7. Conservative properties of the scheme

The scheme proposed is conservative. The derivation of the Rankine–Hugoniot jump conditions used the conservation law of  $\mathbf{U}$  across the shock to derive the shock speed. With a non-conservative scheme, the shock moves with the wrong speed.

Lax–Wendroff theorem ensures that if the numerical solution of a conservative scheme converges, it converges towards a weak solution.

Nevertheless, Lax–Wendroff does not guarantee that weak solutions obtained using conservative methods satisfy the entropy condition.

### 3.8. Involutions

The components of the deformation gradient tensor  $\mathbf{F}$  must satisfy the compatibility conditions. The involutions can be expressed through the following equation:

$$\mathbf{ROT}(\mathbf{F}) = \nabla_x \times \mathbf{F} = \mathbf{0} \quad (3.21)$$

These conditions were not considered to solve the system because the code would lose its simplicity. So, it is convenient to define an average rotation error norm of the element.

$$e_{\mathbf{ROT}(\mathbf{F})} = \frac{1}{N} \sum_{Elem} \|\nabla_x \times \mathbf{F}_E\| \quad (3.22)$$

For short term the error remains bounded. It is probable that the involutions are necessary to make the system symmetric and to guarantee the energy stability in problems with long term iterations, very non–linear systems and coarse meshes.

### 3.9. Internal energy

The internal energy is an important variable to test the quality of the solution.

$$E_{INT} = K + \Pi_{INT} \quad (3.23)$$

The total kinetic energy of the system is given by

$$K = \sum_a \frac{1}{2\rho_0} M_a \|\mathbf{p}_a\|^2 \quad (3.24)$$

The Lobatto integration (integration by nodes) is less expensive to calculate the internal energy than the Gaussian integration, so a possible calculation of the internal energy is (3.25).

$$\Pi_{INT} = \sum_a M_a \left\{ \frac{\kappa}{2} (J_a - 1)^2 + \frac{\mu}{2} (J_a^{-2/3} (\mathbf{F}_a : \mathbf{F}_a) - 3) \right\} \quad (3.25)$$

### 3.10. The entropy-like energy variable

We may express the Helmholtz free-energy function in terms of the internal energy  $E$ , the absolute temperature  $\theta$  and the entropy per unit of reference volume  $\eta$ :

$$\psi = E - \eta\theta \quad (3.28)$$

The Clasius–Plank inequality principle is a strong form of the second law of thermodynamics that states that the internal dissipation  $D_{INT}$  or local production of entropy is non-negative.

$$D_{INT} = \mathbf{P}^T \nabla \mathbf{p} - \frac{D\psi}{Dt} - \eta \frac{D\theta}{Dt} \geq 0 \quad (3.29)$$

The process is reversible if the internal dissipation is zero ( $D_{INT} = 0$ )

It would be interesting to introduce a physical entropy-like energy variable  $E$ . Entropy can be viewed as the quantitative measure of microscopic randomness and disorder.

$$\frac{D}{Dt} \int_{\Omega} E d\Omega = \int_{\Gamma} \frac{1}{\rho_0} \mathbf{t} \cdot \mathbf{p} d\Gamma \quad (3.30)$$

$$\frac{\partial E}{\partial t} = \frac{1}{\rho_0} \frac{\partial}{\partial X_j} (p_i P_{ij}) \quad (3.31)$$

Lin and Szeri [7] stated that entropy gradients are related with shock formation. From a physical point of view, it is easy to imagine that when waves travel into a field with higher sound speed ahead (larger entropy), the velocity gradient of a compression wave front must grow and exceed a critical value in order to evolve into a shock (otherwise it relaxes and there

is no shock). When waves travel into a field with lower sound speed ahead (lower entropy), all compression wave fronts evolve into shocks.

### 3.11. The incompressible problem

The expressions provided are well suited for compressible and nearly incompressible materials. However, to work with incompressible material it is necessary to introduce a new variable, the pressure  $p$

$$p = \kappa(J - 1) \quad (3.32)$$

And now the volumetric potential is re-written like this,

$$\Pi_{VOL} = \int_{\Omega} p(J - 1)d\Omega - \frac{1}{2\kappa} \int_{\Omega} p^2 d\Omega \quad (3.33)$$

The pressure is interpolated with the shape functions (3.34) and the volumetric 1<sup>st</sup> Piola–Kirchhoff stresses are calculated with the expression (3.35). This little change in the formulation was not implemented and the code should be slightly modified in case models of fully incompressible materials were required. It can be found more in Lahiri et al. [10], [11].

$$p = N_a p_a \quad (3.34)$$

$$\mathbf{P}_{VOL} = p\mathbf{F}^{-T} \quad (3.35)$$

### 3.12. Error in pressure

There are various error–estimators presented in the engineering literature. The Z2 estimator described by Zienkiewicz and Zhu [28] is one of the most effective ones and it is commonly used for linear and non–linear materials. We have implemented the simplest nodal error estimation based on the jump of the pressure in each node in the predictor step. However, a Z2 error–estimate or a similar one is strongly recommended.

$$e_{press} \Big|_a = \max(p_a^{elem}) - \min(p_a^{elem}) \quad \forall elem \in a \quad (3.36)$$

# Chapter 4

## Element Types

### 4.1. Linear bar element

We can use the following simplified model for a 1-D rod problem,

$$\begin{aligned}\frac{\partial p}{\partial t} &= \frac{\partial P}{\partial X} \\ \frac{\partial F}{\partial t} &= \frac{1}{\rho_0} \frac{\partial p}{\partial X}\end{aligned}\tag{4.1}$$

The one-dimensional constitutive model becomes

$$P = \frac{2}{3} \mu \frac{F^{2/3} - F^{-2/3}}{F} + \kappa(F - 1)\tag{4.2}$$

For small deformations the relation between P and F is linear.

$$P = \left( \frac{4}{3} \mu + \kappa \right) (F - 1)\tag{4.3}$$

If the Poisson modulus is zero, in fact, we have the classical linear elastic equation.

$$P = E(F - 1)\tag{4.4}$$

Putting these two equations in the compact wave equation form we have

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathfrak{S}}{\partial X} = 0\tag{4.5}$$

Where,

$$\mathbf{U} = \begin{bmatrix} p \\ F \end{bmatrix} \quad \mathfrak{S} = - \begin{bmatrix} P \\ \frac{1}{\rho_0} p \end{bmatrix} \quad (4.6)$$

We assume that the rod is discretized in  $(N/2-1)$  elements, this is  $N$  degrees of freedom, and all the bar elements have the same length  $L_E$  and uniform material properties.

For nodes inside the rod, this is  $a=2:(N/2-1)$ , the Taylor predictor is given by (4.7). For nodes on the boundary, in order to maintain the second order accuracy, the best way is (4.8) and (4.9).

$$\left[ U_i^{n+\frac{1}{2}} \right]_a = \left[ U_i^n \right]_a - \frac{\Delta t}{2} \left[ \frac{\partial \mathfrak{S}_i^n}{\partial X} \right]_a = \left[ U_i^n \right]_a - \frac{\Delta t}{4L_E} (\left[ \mathfrak{S}_i^n \right]_{a+1} - \left[ \mathfrak{S}_i^n \right]_{a-1}) \quad (4.7)$$

$$\left[ U_i^{n+\frac{1}{2}} \right]_1 = \left[ U_i^n \right]_1 - \frac{\Delta t}{2} \left[ \frac{\partial \mathfrak{S}_i^n}{\partial X} \right]_1 = \left[ U_i^n \right]_1 - \frac{\Delta t}{4L_E} (-\left[ \mathfrak{S}_i^n \right]_3 + 4\left[ \mathfrak{S}_i^n \right]_2 - 3\left[ \mathfrak{S}_i^n \right]_1) \quad (4.8)$$

$$\left[ U_i^{n+\frac{1}{2}} \right]_N = \left[ U_i^n \right]_N - \frac{\Delta t}{2} \left[ \frac{\partial \mathfrak{S}_i^n}{\partial X} \right]_N = \left[ U_i^n \right]_N - \frac{\Delta t}{4L_E} (3\left[ \mathfrak{S}_i^n \right]_N - 4\left[ \mathfrak{S}_i^n \right]_{N-1} + \left[ \mathfrak{S}_i^n \right]_{N-2}) \quad (4.9)$$

The consistent mass matrix for one element is (4.10). The lumped system is (4.11).

$$\mathbf{M}_E = \int_L N^T N dX = \frac{L_E}{6} \begin{bmatrix} 2 & 0 & 1 & 0 \\ 0 & 2 & 0 & 1 \\ 1 & 0 & 2 & 0 \\ 0 & 1 & 0 & 2 \end{bmatrix} \quad (4.10)$$

$$\mathbf{M}_L = \frac{L_E}{2} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (4.11)$$



$$\begin{bmatrix} \delta U_1 \\ \delta U_2 \\ \delta U_3 \\ \delta U_4 \\ \delta U_5 \\ \\ \delta U_{N-4} \\ \delta U_{N-3} \\ \delta U_{N-2} \\ \delta U_{N-1} \\ \delta U_N \end{bmatrix} = \frac{\Delta t}{L_E} \begin{bmatrix} 3\mathfrak{I}_1 - \mathfrak{I}_3 \\ 3\mathfrak{I}_2 - \mathfrak{I}_4 \\ (\mathfrak{I}_1 - \mathfrak{I}_5)/2 \\ (\mathfrak{I}_2 - \mathfrak{I}_6)/2 \\ (\mathfrak{I}_3 - \mathfrak{I}_7)/2 \\ \\ (\mathfrak{I}_{N-6} - \mathfrak{I}_{N-2})/2 \\ (\mathfrak{I}_{N-5} - \mathfrak{I}_{N-1})/2 \\ (\mathfrak{I}_{N-4} - \mathfrak{I}_N)/2 \\ -\mathfrak{I}_{N-3} + 3\mathfrak{I}_{N-1} \\ -\mathfrak{I}_{N-2} + 3\mathfrak{I}_N \end{bmatrix} \quad (4.16)$$

The boundary conditions to model an impact on a rod are:

*Fixed end:*

$$U_{N-1} = 0 \quad \text{No velocity} \quad (\text{strong way})$$

*Free end:*

$$\mathfrak{I}_1 = 0 \quad \text{No traction} \quad (\text{weak way})$$

Initial condition:

*Free end:*

$$U_1 = p_0 \quad \text{Initial impact}$$

It is important to see that the lumped system basically coincides in 1-D with the central finite difference algorithm:

$$\frac{U_i^{n+1} - U_i^n}{\Delta t} + \frac{\mathfrak{I}_{i+2}^{n+1/2} - \mathfrak{I}_{i-2}^{n+1/2}}{2h} = 0 \quad (4.17)$$

$$h(U_i^{n+1} - U_i^n) = \frac{\Delta t}{2} (-\mathfrak{I}_{i-2}^{n+1/2} + \mathfrak{I}_{i+2}^{n+1/2}) \quad (4.18)$$

In the case of small deformation and linear elasticity the Courant's stability condition (4.19) is similar to the Euler condition.

$$\Delta t \leq \beta \frac{h_E}{c} \quad (4.19)$$

$h_E$  is the smallest characteristic length of the element.

$\beta$  is a number between 0.5 (consistent) and 1 (lumped)  
 $c$  is the speed in each element

If no transversal effect is considered, the maximum celerity in the bar is the classical result (4.20).

$$c \leq \sqrt{\frac{E}{\rho}} \quad (4.20)$$

## 4.2. Constant strain linear triangles

To create the system we need to particularize the  $\mathbf{M}$  matrix and the right hand side vector  $\mathbf{f}$  for the linear shape functions of the first order triangular element.

### Unknowns and flux vectors

For 2-D the vectors are

$$[U_i] = \begin{bmatrix} p_1 \\ p_2 \\ F_{11} \\ F_{12} \\ F_{21} \\ F_{22} \end{bmatrix} \quad [\mathfrak{S}_{ij}] = - \begin{bmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \\ p_1/\rho_0 & 0 \\ 0 & p_1/\rho_0 \\ p_2/\rho_0 & 0 \\ 0 & p_2/\rho_0 \end{bmatrix} \quad (4.21)$$

### Predictor step

In the prediction the same amount (4.22) is added in each node of the element.

$$\delta U^{n+\frac{1}{2}} = -\frac{\Delta t}{2} [\mathfrak{S}_{ij}]_c \frac{\partial N_c}{\partial X_j} = -\frac{\Delta t}{2} \mathfrak{S}_c \mathbf{B}_{cj} \quad (4.22)$$

### Consistent mass matrix

The consistent mass matrix per element is

$$M_E = \int_A N_c N_a dA \quad (4.23)$$

Where  $J$  is the Jacobian of the local – global transformation. If the coordinates are given in counter clockwise numeration, the determinant is positive.

For the triangular element, the mass integrals are (4.24), (4.25) and (4.26).

$$\int_A N_a^2 dA = \frac{A_e}{6} \quad (4.24)$$

$$\int_A N_a N_b dA = \frac{A_e}{12} \quad \text{with } a \neq b \quad (4.25)$$

$$2A_e = \text{Det}(J) = \text{Det} \left\{ \begin{array}{c} \partial X_i \\ \partial \xi_j \end{array} \right\} \quad (4.26)$$

So, the elemental consistent mass matrix  $\mathbf{M}_E$  is

$$\mathbf{M}_E = \frac{A_E}{12} \begin{bmatrix} 2\mathbf{I}_{4 \times 4} & \mathbf{I}_{4 \times 4} & \mathbf{I}_{4 \times 4} \\ \mathbf{I}_{4 \times 4} & 2\mathbf{I}_{4 \times 4} & \mathbf{I}_{4 \times 4} \\ \mathbf{I}_{4 \times 4} & \mathbf{I}_{4 \times 4} & 2\mathbf{I}_{4 \times 4} \end{bmatrix} \quad (4.27)$$

However, it is preferable to use the lumped matrix instead of the consistent one because having the same order of accuracy the algorithm becomes fully explicit with the lumped matrix.

$$\mathbf{M}_E = \frac{A_E}{3} \mathbf{I}_{18 \times 18} \quad (4.28)$$

### Right hand side vector

- Internal amount of flux:

$$\begin{aligned} \Delta t \int_A \frac{\partial N_c}{\partial X_j} N_a \left[ \mathfrak{F}_{ij}^{n+\frac{1}{2}} \right]_a dX_1 dX_2 &= \Delta t \sum_E \int_{AE} B_{cj} N_a \left[ \mathfrak{F}_{ij}^{n+\frac{1}{2}} \right]_a dX_1 dX_2 = \\ \Delta t \sum_E B_{cj} \frac{1}{3} \sum_a \left[ \mathfrak{F}_{ij}^{n+\frac{1}{2}} \right]_a A_E & \end{aligned} \quad (4.29)$$

The elemental area can be obtained from the Jacobian of the local–global transformation. For the linear triangular element this is

$$\text{Det}(\mathbf{J}) = (X_{1|_1} - X_{1|_3})(X_{2|_2} - X_{2|_3}) - (X_{1|_2} - X_{1|_3})(X_{2|_1} - X_{2|_3}) \quad (4.30)$$

The determinant is positive for counter clockwise numeration of local nodes.

The  $\mathbf{B}$  matrix components (4.31) are constant.

$$\begin{aligned}
B_{11} &= \frac{\partial N_1}{\partial X_1} = \frac{1}{2A_e} ([X_2]_2 - [X_2]_3) \\
B_{12} &= \frac{\partial N_1}{\partial X_2} = \frac{1}{2A_e} ([X_1]_3 - [X_1]_2) \\
B_{21} &= \frac{\partial N_2}{\partial X_1} = \frac{1}{2A_e} ([X_2]_3 - [X_2]_1) \\
B_{22} &= \frac{\partial N_2}{\partial X_2} = \frac{1}{2A_e} ([X_1]_1 - [X_1]_3) \\
B_{31} &= \frac{\partial N_3}{\partial X_1} = \frac{1}{2A_e} ([X_2]_1 - [X_2]_2) \\
B_{32} &= \frac{\partial N_3}{\partial X_2} = \frac{1}{2A_e} ([X_1]_2 - [X_1]_1)
\end{aligned} \tag{4.31}$$

The internal amount of flux generate by the local node  $c$  of the element is

- Amount of flux lost through the contour:

$$\left[ f_{EXT} \right]_c = -\Delta t \int_{\partial A} N_c N_a \left[ \mathfrak{S}_{ij}^{n+\frac{1}{2}} \right]_a N_j^{c-a} dL = -\Delta t \int_{\partial A} N_c N_a dL \left[ \mathfrak{S}_{ij}^{n+\frac{1}{2}} \right]_a N_j^{c-a} \tag{4.32}$$

For the triangular element the boundary integrals are (4.33) and (4.34).

$$\int_l N_a^2 dL = \int_l N_c^2 dL = \frac{1}{3} L_{a-c} \tag{4.33}$$

$$\int_l N_a N_c dL = \frac{1}{6} L_{a-c} \tag{4.34}$$

Where  $L_{a-c}$  is the length of the side a-c of integration. The external normal vector to the side a-c is (4.35)

$$\mathbf{N}^{a-c} = \text{sign}\{\text{Det}(\mathbf{J})\} \begin{bmatrix} X_2|_c - X_2|_a \\ -(X_1|_c - X_1|_a) \end{bmatrix} \tag{4.35}$$

The amount of flux lost in the side a-c through the node a is

$$[\mathbf{f}_{EXT}^{a-c}]_a = -\Delta t \frac{L_{a-c}}{2} N_j^{a-c} \left( \frac{2}{3} [\mathfrak{S}_{ij}^{n+\frac{1}{2}}]_a + \frac{1}{3} [\mathfrak{S}_{ij}^{n+\frac{1}{2}}]_c \right) \quad (4.36)$$

And trough the node  $c$  is

$$[\mathbf{f}_{EXT}^{a-c}]_c = -\Delta t \frac{L_{a-c}}{2} N_j^{a-c} \left( \frac{1}{3} [\mathfrak{S}_{ij}^{n+\frac{1}{2}}]_a + \frac{2}{3} [\mathfrak{S}_{ij}^{n+\frac{1}{2}}]_c \right) \quad (4.37)$$

This is a trapezoidal integration of the flux. It is important to remark that a linear approximation such as (4.38) gives bad results and the scheme loses its conservativeness.

$$[\mathbf{f}_{EXT}^{a-c}]_a = -\Delta t \frac{L_{a-c}}{2} N_j^{a-c} \left( [\mathfrak{S}_{ij}^{n+\frac{1}{2}}]_a \right) \quad [\mathbf{f}_{EXT}^{a-c}]_c = -\Delta t \frac{L_{a-c}}{2} N_j^{a-c} \left( [\mathfrak{S}_{ij}^{n+\frac{1}{2}}]_c \right) \quad (4.38)$$

It is only necessary to compute the amount of flux lost through the sides placed on the boundary of the region. The integrals over internal sides are going to be cancelled and the external flux is added at the end of the loop.

Adding the two terms we get the right hand side vector of the system

$$[\mathbf{f}]_c = [\mathbf{f}_{INT}]_c + [\mathbf{f}_{EXT}]_c \quad (4.39)$$

### First Piola–Kirchhof stress tensor

It is important to remind that the Piola–Kirchhoff is obtained in a slightly different way for plane strain conditions. For large deformation we use the formula (4.40). For small deformations, it is possible to use the linear approximation (4.41).

$$\mathbf{P} = \mu J^{-\frac{2}{3}} \left[ \mathbf{F} - \frac{1}{3} (1 + \text{tr}(\mathbf{C})) \mathbf{F}^{-T} \right] + \kappa (J - 1) \mathbf{J} \mathbf{F}^{-T} \quad (4.40)$$

With  $J = \det(\mathbf{F})$ .

$$\mathbf{P} = \mu \left[ \mathbf{F} + \mathbf{F}^T - \frac{2}{3} (1 + \text{tr}(\mathbf{F})) \mathbf{I} \right] + \kappa \text{tr}(\mathbf{F}) \mathbf{I} \quad (4.41)$$

The corresponding stress tensor  $\mathbf{P}$  is symmetric for small deformations.

Boundary conditions

There are two main types of boundary conditions:

- Boundaries without displacement (constant zero velocity). As a consequence, we have a traction (reaction) on that boundary.

$$[\mathfrak{I}_{ij} N_j] = - \begin{bmatrix} T_1 \\ T_2 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad (4.42)$$

The wall boundary conditions are imposed in a *strong way*, this is, via the unknowns.

Type of wall	Strong conditions
Wall parallel to X	$p_y=0$
Wall parallel to Y	$p_x=0$

Sometimes, it is possible to determine the value of the deformation and it can be convenient to impose those values of  $\mathbf{F}$  in a *strong way* in order to achieve a more precise result.

- Free boundaries, without tractions.

$$[\mathfrak{I}_{ij} N_j] = - \begin{bmatrix} 0 \\ 0 \\ p_1 N_1 / \rho_0 \\ p_1 N_2 / \rho_0 \\ p_2 N_1 / \rho_0 \\ p_2 N_2 / \rho_0 \end{bmatrix} \quad (4.43)$$

The free boundary conditions are imposed in a *weak way*, this is, using the fluxes.

Type of free boundary	Weak conditions
Free boundary parallel to X	$P_{xx}=0 \quad P_{xy}=0$
Free boundary parallel to Y	$P_{yx}=0 \quad P_{yy}=0$

### 4.3. Constant strain linear quadrilaterals

Here we produce the main differences in the formulation between linear triangles and linear quadrilaterals.

#### Consistent mass matrix

The elemental consistent mass matrix  $\mathbf{M}_E$  needs to be evaluated numerically. The lumped mass matrix is

$$\mathbf{M}_E = \frac{A_E}{4} \mathbf{I}_{24 \times 24} \quad (4.44)$$

The elemental area can be obtained from the Jacobian of the local–global transformation with one Gauss point of integration. The Jacobian matrix of local – global transformation is not constant and it depends on the local position  $(\xi, \eta)$ .

$$\mathbf{J} = \frac{1}{4} \begin{bmatrix} (1-\eta)(X_2 - X_1) + (1+\eta)(X_3 - X_4) & (1-\eta)(Y_2 - Y_1) + (1+\eta)(Y_3 - Y_4) \\ (1-\xi)(X_4 - X_1) + (1+\xi)(X_3 - X_2) & (1-\xi)(Y_4 - Y_1) + (1+\xi)(Y_3 - Y_2) \end{bmatrix} \quad (4.45)$$

#### Internal amount of flux

For the local node  $c$  of the element this is

$$\left[ \mathbf{f}_{INT} \right]_c = \Delta t \int_A B_{cj} N_a \left[ \mathfrak{S}_{ij}^{n+\frac{1}{2}} \right]_a \text{Det}(J) d\xi d\eta \quad (4.46)$$

The integral must be performed with four Gaussian points.

And the  $\mathbf{B}$  matrix components, this is the derivatives of the shape functions, are now dependant on the position:

$$B_{cj} = \frac{\partial N_c}{\partial X_j} \quad \mathbf{B} = \frac{1}{A_E} \begin{bmatrix} -(1-\eta)J_{22} + (1-\xi)J_{12} & (1-\eta)J_{21} - (1-\xi)J_{11} \\ (1-\eta)J_{22} + (1+\xi)J_{12} & -(1-\eta)J_{21} - (1+\xi)J_{11} \\ (1+\eta)J_{22} - (1+\xi)J_{12} & -(1+\eta)J_{21} + (1+\xi)J_{11} \\ -(1+\eta)J_{22} - (1-\xi)J_{12} & (1+\eta)J_{21} + (1-\xi)J_{11} \end{bmatrix} \quad (4.47)$$

# Chapter 5

## Viscous Formulation

### 5.1. Motivation

In problems which are highly nonlinear, such as high velocity impacts, the shock waves present high frequencies. Very non-linear cases are more sensitive to this problem due to the coupling within the different modes in the solution, which creates an exchange of energy between the high frequency modes and low frequency vibrations. In any case, the discontinuities are approximated with continuous functions, the solution can suffer oscillations which affect the stress level and they can corrupt the solution over the time.

A way to overcome this is to add an artificial viscosity to damp numerically the shock wave and to spread the discontinuity.

The artificial viscosity stabilizes the solution by dissipating the high frequency vibrations. The proposed viscous stabilizations is clearly momentum conserving and variationally consistent provided that the viscous stresses that are added are zero for rigid body displacements.

It is convenient to choose the same volumetric and deviatoric percentage of damping to simplify the problem.

$$\lambda_v = \alpha \frac{\lambda h}{c} \quad \mu_v = \alpha \frac{\mu h}{c} \quad (5.1)$$

Where

$$c = \sqrt{\frac{\lambda + 2\mu}{\rho_0}} \quad (5.2)$$

Like this, the dissipation is controlled by the non-dimensional parameter  $\alpha$ .

The dissipation term can be used with two objectives:

a) To filter the high frequency dispersion error.

It becomes especially effective in nonlinear cases. If the system is slightly damped it is possible to avoid the corruption of the solution.

b) To get static solutions.

In this case, it is important to realize that the transportiveness of the flow is controlled by the Peclet number.

$$Pe = \frac{\rho_0 c h_E}{\lambda_{VIS} + 2\mu_{VIS}} \quad (5.3)$$

- $Pe \rightarrow 0$ . No convection, pure diffusion. The flow is quite, it represents the pure static solution. The diffusion tends to spread the fluxes equally in all the directions and the initial conditions do not influence the process after some time.
- $Pe \rightarrow \infty$ . No diffusion, pure convection. It is the pure dynamic solution without any kind of damping. The flux is transported immediately and the initial conditions have a strong influence during the process.

## 5.2. Viscous constitutive model

A simple dissipative formulation can be derived by using a total Lagrangian viscous stress tensor.

$$\mathbf{P} = \mathbf{P}^{INV} + \mathbf{P}^{VIS} = \frac{\partial \psi}{\partial \mathbf{F}} + \mathbf{D} : \nabla \mathbf{p} \quad (5.4)$$

In Einstein notation this is

$$P_{ij} = \frac{\partial \psi}{\partial F_{ij}} + D_{ijkl} \frac{\partial p_k}{\partial X_l} \quad (5.5)$$

Where  $\underline{\mathbf{D}}$  is a fourth order dissipation tensor that can be derived from the total Lagrangian viscous fourth order tensor  $\underline{\mathbf{C}}$  in terms of the second Piola–Kirchhoff.

$$\begin{aligned}
 \mathbf{S} &= \underline{\mathbf{C}} : \dot{\mathbf{E}} \\
 \mathbf{F}^{-1} \mathbf{P} &= \underline{\mathbf{C}} : \left( \frac{1}{2} \left( \dot{\mathbf{F}}^T \mathbf{F} + \mathbf{F}^T \dot{\mathbf{F}} \right) \right) = \underline{\mathbf{C}} : \left( \frac{1}{2} \left( \left( \frac{\nabla \mathbf{p}}{\rho_0} \right)^T \mathbf{F} + \mathbf{F}^T \frac{\nabla \mathbf{p}}{\rho_0} \right) \right) \\
 &= \underline{\mathbf{C}} : \left( \frac{1}{2\rho_0} \left( (\nabla \mathbf{p})^T \mathbf{F} + \mathbf{F}^T \nabla \mathbf{p} \right) \right) = \frac{1}{2\rho_0} \text{tr} \left( \underline{\mathbf{C}}^T \left( (\nabla \mathbf{p})^T \mathbf{F} + \mathbf{F}^T \nabla \mathbf{p} \right) \right) = \\
 &= \frac{1}{2\rho_0} \left( \text{tr} \left( \underline{\mathbf{C}}^T \nabla \mathbf{p}^T \mathbf{F} \right) + \text{tr} \left( \mathbf{C}^T \mathbf{F}^T \nabla \mathbf{p} \right) \right) = \frac{1}{\rho_0} \text{tr} \left( \underline{\mathbf{C}}^T \mathbf{F}^T \nabla \mathbf{p} \right)
 \end{aligned} \tag{5.6}$$

$$\underline{\mathbf{D}} = \frac{1}{\rho_0} \mathbf{F} \underline{\mathbf{C}} \tag{5.7}$$

$$\begin{aligned}
 \underline{\mathbf{D}} &= \frac{1}{\rho_0} \mathbf{F} \underline{\mathbf{C}} = \frac{\lambda_v}{\rho_0} (\mathbf{F} \otimes \mathbf{F}) + \frac{\mu_v}{\rho_0} (\mathbf{F} (\mathbf{i} + \mathbf{i}^T) \mathbf{F}) \\
 i_{ijkl} &= \delta_{ik} \delta_{jl} \\
 D_{ijkl} &= \frac{\lambda_v}{\rho_0} F_{ij} F_{kl} + \frac{\mu_v}{\rho_0} (F_{im} F_{km}) \delta_{jl} + \frac{\mu_v}{\rho_0} F_{il} F_{kj}
 \end{aligned} \tag{5.8}$$

So finally the viscous first Piola–Kirchhoff stress is given by (5.9) where  $\mathbf{b}$  is the Finger deformation tensor (5.10).

$$\mathbf{P}^{vis} = \underline{\mathbf{D}} : \nabla \mathbf{p} = \frac{\lambda_v}{\rho_0} (\mathbf{F} : \nabla \mathbf{p}) \mathbf{F} + \frac{\mu_v}{\rho_0} (\mathbf{b} \nabla \mathbf{p} + \mathbf{F} (\nabla \mathbf{p})^T \mathbf{F}) \tag{5.9}$$

$$\mathbf{b} = \mathbf{F} \mathbf{F}^T \tag{5.10}$$

An alternative formulation can be done using the spatial description of the elasticity tensor instead of the material representation

$$\boldsymbol{\sigma} = \underline{\mathbf{C}} : \mathbf{d} \tag{5.11}$$

Where  $\mathbf{d}$  is the rate of deformation tensor

$$\mathbf{d} = \frac{1}{2\rho_0} \left( \nabla \mathbf{p} + (\nabla \mathbf{p})^T \right) \tag{5.12}$$

And  $\underline{\mathbf{D}}$  can be obtained with (5.13). The the resultant viscous Piola–Kirchhoff is (5.14) where  $\mathbf{C}$  is the Cauchy–Green deformation tensor (5.15).

$$\underline{\mathbf{D}} = \frac{1}{\rho_0 J} \mathbf{C} \mathbf{F}^{-T} \quad (5.13)$$

$$\mathbf{P}^{VIS} = \underline{\mathbf{D}} : \nabla \mathbf{p} = \frac{\lambda_v}{\rho_0 J} (\mathbf{F}^{-T} : \nabla \mathbf{p}) \mathbf{F}^{-T} + \frac{\mu_v}{\rho_0 J} (\nabla \mathbf{p} \mathbf{C}^{-1} + \mathbf{F}^{-T} (\nabla \mathbf{p})^T \mathbf{F}^{-T}) \quad (5.14)$$

$$\mathbf{C} = \mathbf{F}^T \mathbf{F} \quad (5.15)$$

This is the result that we used to model the viscous effect.

### 5.3. Conservation–Law with viscous terms

After adding the viscous term, the governing equations are (5.16) and (5.17), and the compact wave form is (5.18).

► Conservation of momentum.

$$\frac{\partial p_i}{\partial t} = \frac{\partial}{\partial X_j} (P_{ij}^{INV} + P_{ij}^{VIS}) \quad (5.16)$$

► Deformation–linear momentum relation (remains without changes).

$$\frac{\partial F_{ij}}{\partial t} = \frac{\partial}{\partial X_k} \left( \frac{1}{\rho_0} p_i \delta_{jk} \right) \quad (5.17)$$

$$\frac{\partial U_i}{\partial t} + \frac{\partial}{\partial X_j} (\mathfrak{S}_{ij}^{INV} - \mathfrak{S}_{ij}^{VIS}) = 0 \quad (5.18)$$

For 2-D the vectors of unknowns and fluxes are

$$[\mathbf{U}_i] = \begin{bmatrix} p_1 \\ p_2 \\ F_{11} \\ F_{12} \\ F_{21} \\ F_{22} \end{bmatrix} \quad [\mathfrak{S}_{ij}^{INV}] = - \begin{bmatrix} P_{11}^{INV} & P_{12}^{INV} \\ P_{21}^{INV} & P_{22}^{INV} \\ p_1/\rho_0 & 0 \\ 0 & p_1/\rho_0 \\ p_2/\rho_0 & 0 \\ 0 & p_2/\rho_0 \end{bmatrix} \quad [\mathfrak{S}_{ij}^{VIS}] = \begin{bmatrix} P_{11}^{VIS} & P_{12}^{VIS} \\ P_{21}^{VIS} & P_{22}^{VIS} \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \quad (5.19)$$

In general, the results obtained with this method for two-node truss elements and linear triangles were very good. Nevertheless, the extension of this approach to quadrilateral elements showed oscillations.

## Chapter 6

# Stability and Convergence

### 6.1. CFL condition

The Courant, Friedrichs and Lewy condition [2], [25] is a *necessary* condition that must be satisfied by any scheme if we expect it to be stable and converge to the solution.

It states that a numerical method can be convergent only if its numerical domain of dependence contains the true domain of dependence of the PDE, at least in the limit as  $\Delta t$  and  $\Delta x$  go to zero. In other words, the propagation of errors is controlled by the maximum eigenvalue of the normal flux Jacobian (2.38).

Nevertheless, CFL condition is not always *sufficient* to guarantee stability. Even when the CFL condition is satisfied, the method can be unstable.

### 6.2. Stability criterion

Like other explicit methods, the two-step Taylor-Galerkin algorithm is conditionally stable. The stability criterion used is obtained from the one-dimensional linear equation and then it is generalized to multidimensional problems

$$\frac{\partial U}{\partial t} + c^2 \frac{\partial U}{\partial X} - \frac{E_{VIS}}{\rho_0} \frac{\partial^2 U}{\partial X^2} = 0 \quad (6.1)$$

Where  $E$  and  $E_{VIS}$  are the elastic and viscous elastic modula respectively.

$$c = \sqrt{\frac{E}{\rho_0}} \quad (6.2)$$

For the algorithm with a consistent mass matrix and equally spaced mesh, the following step in each node  $i$  is

$$\begin{aligned} \frac{1}{6}(U_{i+1}^{n+1} + 4U_i^{n+1} + U_{i-1}^{n+1}) &= \frac{1}{6}(U_{i+1}^n + 4U_i^n + U_{i-1}^n) - \frac{C}{2}(U_{i+1}^n - U_{i-1}^n) + \\ &+ C\left(\frac{C}{2} + \frac{1}{Pe}\right)(U_{i+1}^n - 2U_i^n + U_{i-1}^n) \end{aligned} \quad (6.3)$$

And with a lumped mass the expression is (6.4). Where  $C$  is the Courant number and  $Pe$  is the Peclet number (6.4).

$$U_i^{n+1} = U_i^n - \frac{C}{2}(U_{i+1}^n - U_{i-1}^n) + C\left(\frac{C}{2} + \frac{1}{Pe}\right)(U_{i+1}^n - 2U_i^n + U_{i-1}^n) \quad (6.4)$$

$$C = \frac{c\Delta t}{\Delta x} \quad Pe = \frac{\rho_0 c \Delta x}{E_{VIS}} \quad (6.5)$$

It is possible to insert a Fourier mode  $m$  of the form (6.6) where  $j$  is the imaginary number.

$$U_i^n = \lambda_m^n \exp(mi\Delta x j) = \lambda_m^n \{\cos(mi\Delta x) + j \sin(mi\Delta x)\} \quad (6.6)$$

The amplitude  $\lambda$  should not increase in each new step. Introducing the Fourier mode in the consistent discretization we get (6.7) and with the lumped mass matrix the resultant expression is (6.8).

$$\frac{\lambda_m^{n+1}}{\lambda_m^n} = 1 + C\left(C + \frac{2}{Pe}\right)(\cos(m\Delta x) - 1) - jC \sin(m\Delta x) \quad (6.7)$$

$$\frac{\lambda_m^{n+1}}{\lambda_m^n} = 1 + C\left(C + \frac{2}{Pe}\right)(\cos(m\Delta x) - 2) + jC \sin(m\Delta x) \quad (6.8)$$

So, the stability condition is

$$C\left(C + \frac{2}{Pe}\right) \leq \alpha \quad (6.9)$$

Where  $\alpha = 1/3$  in the case of consistent matrix, this is (6.10), and  $\alpha = 1$  for the lumped mass, this is (6.11).

$$C \leq \sqrt{\frac{1}{Pe^2} + \frac{1}{3}} - \frac{1}{Pe} \quad (6.10)$$

$$C \leq \sqrt{\frac{1}{Pe^2} + 1} - \frac{1}{Pe} \quad (6.11)$$

### 6.3. Generalization of the stability condition

For the Navier–Stokes equations in two and three dimensions we can obtain a necessary stability condition by generalizing the adimensional numbers defined above.

$$C = \frac{c\Delta t}{h_E} \quad Pe = \frac{\rho_0 c h_E}{\lambda_{VIS} + 2\mu_{VIS}} \quad (6.12)$$

Where  $c$  is now the maximum celerity of the wave, this is the speed of the pressure waves (5.2) and  $h_E$  is the minimum length of the element.

### 6.4. Convergence

Convergence is very difficult to establish theoretically and in practice we use Lax’s equivalence theorem [12] that states that for linear problems a necessary and sufficient condition is both consistency and stability. However, in our case this theorem is of limited use since the governing equations can be non–linear. Stability is a necessary condition for convergence, but not sufficient.

In practice, boundedness, conservativeness, and transportiveness are more critical properties to achieve a realistic engineering solution with meshes sometimes quite coarse than the order of convergence [19].

# Chapter 7

## Examples

### 7.1. Bar axially loaded

A straight bar was axially loaded with an instant linear momentum.

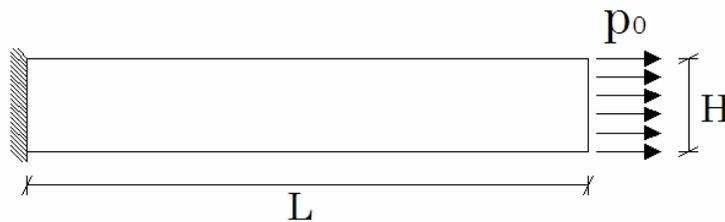


Figure 4: Sketch of the problem

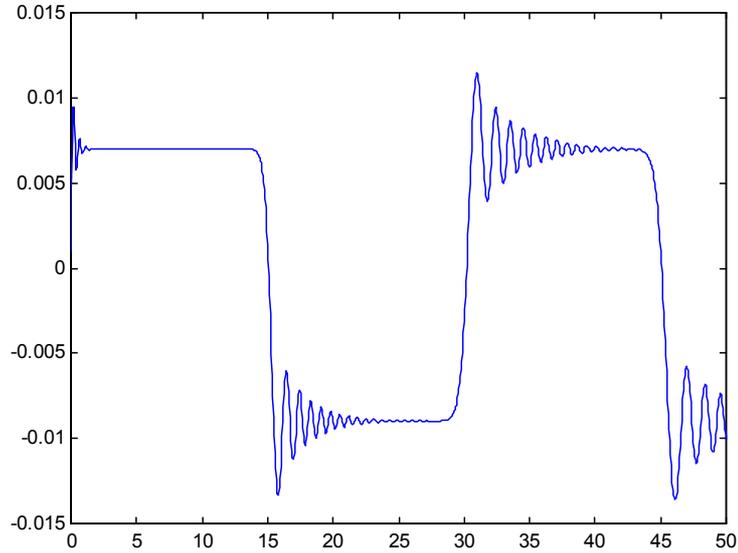
The data of the case are the following ones:

Instant pulse $p_0$	0.1
Length $L$	10
Height $H$	1
Young modulus $E$	1
Poisson modulus $\nu$	0.4
Density $\rho$	1

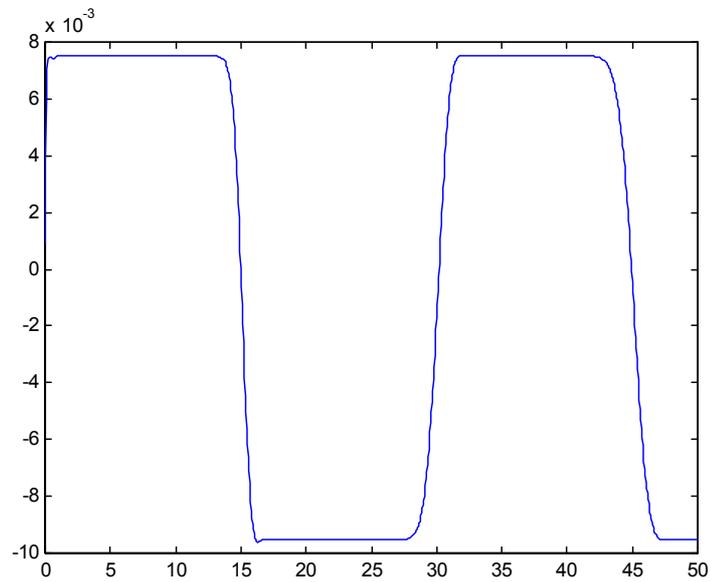
The purpose of this example is to illustrate a reasonable numerical damping to minimize the oscillations in 1-D and in 2-D. First, it was performed a one-dimensional simulation without viscosity with the following characteristics:

Element type	Linear truss 1D
Nodes	101
$\Delta x$	0.10
$\Delta t$	0.01
Courant Number	$0.15 < 1$
Peclet Number	$\infty$

The oscillations appear in the solution from the beginning but they become more visible after some time.



*Figure 5: Evolution of displacement in the head of the bar. No viscosity*



*Figure 6: Evolution of displacement in the head of the bar with viscosity*

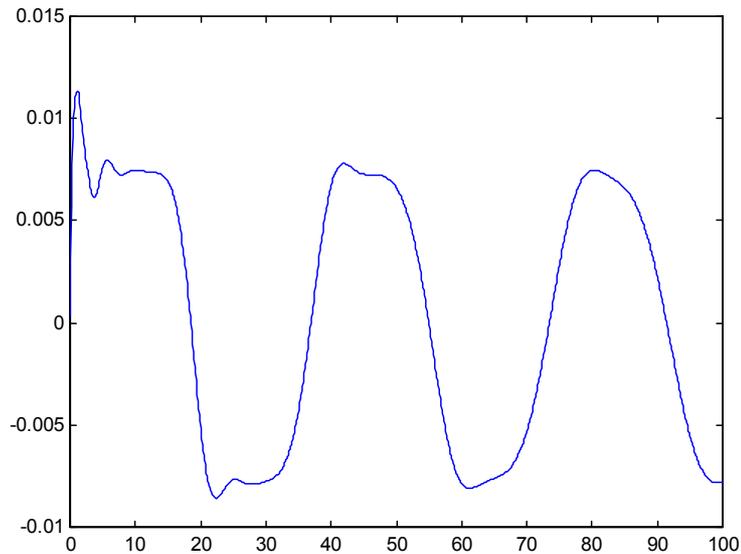
The same simulation was performed with artificial damping and the results were much better. The key variables are:

Element type	Linear truss 1D
Nodes	101
$\Delta x$	0.10
$\Delta t$	0.01
Courant Number	$0.15 < 0.82$
Peclet Number	4.88

If we compare the Figures 5 and 6, we can conclude that the viscous model represents better the amplitudes of the high frequency components.

For a quite coarse mesh in 2-D with the following characteristics the results were also acceptable,

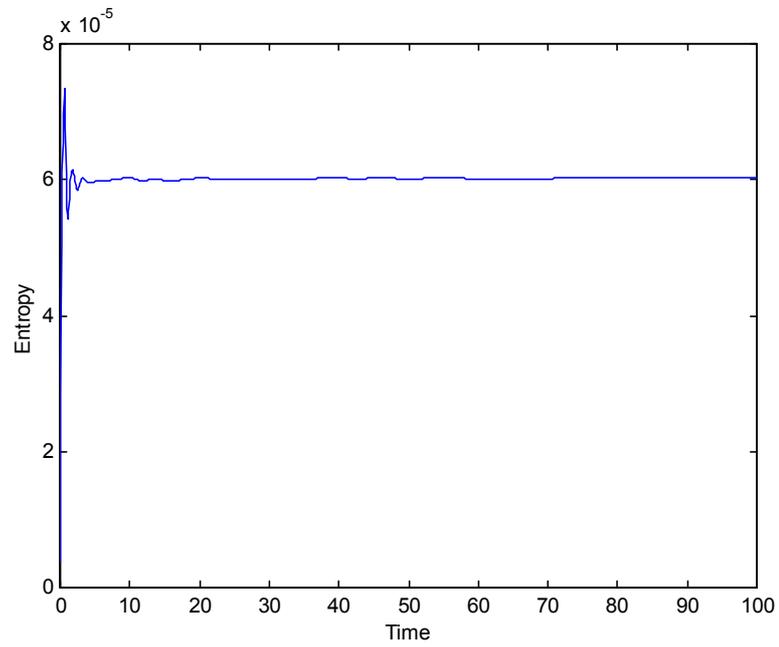
Element type	Linear triang 2D
Nodes	63
$\Delta x$	0.10
$\Delta t$	0.05
Courant Number	$0.10 < 0.87$
Peclet Number	6.88



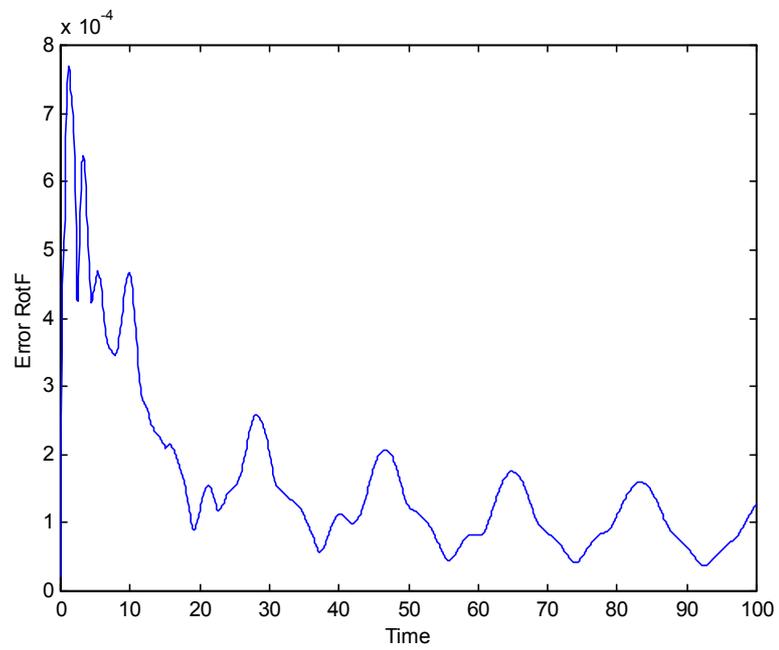
*Figure 7: Evolution of displacement in the head of the bar with viscosity*

The internal entropy-like energy variable (Fig. 8) is constant after the first steps as it was expected. The average rotational error of  $\mathbf{F}$  (3.22) gets reduced with the viscosity (Fig. 9).

The pressure waves evolve quickly concentrating the traction due to the impact close to the wall. In the Figure n° 10 it is possible to see distribution of pressure during the peak of traction. The violet zone is the part of the bar which has zero stress.



*Figure 8: Evolution of entropy-like energy variable*



*Figure 9: Evolution of the rotational error of F*

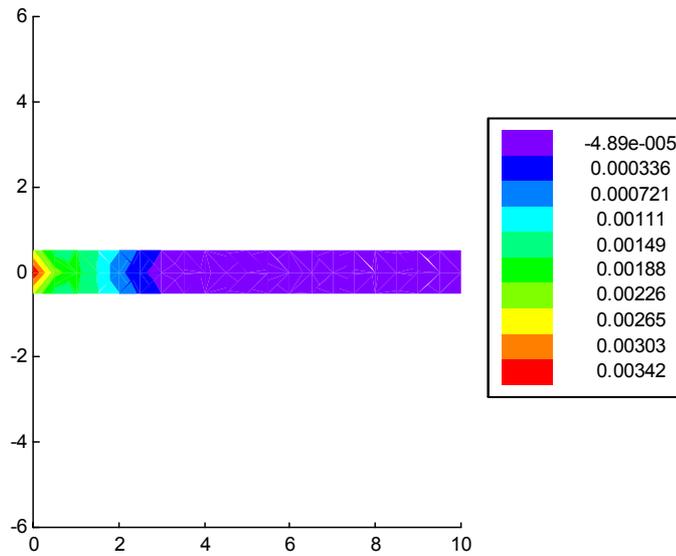


Figure 10: Distribution of pressure in the bar at Time=10

## 7.2. Rotating plates

An initial angular velocity  $\Omega$  (cycles per unit of time) is applied on a square plate without constraints. The initial conditions are such that there is no steady state.

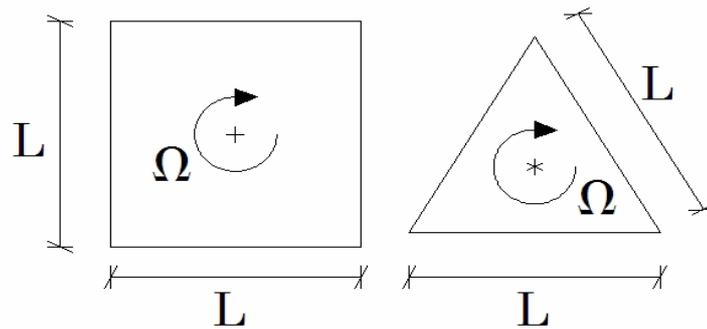


Figure 11: Sketches of the problem

### 7.2.1. Square plate

The Figure n° 12 shows a discretization of 8x8 square elements.

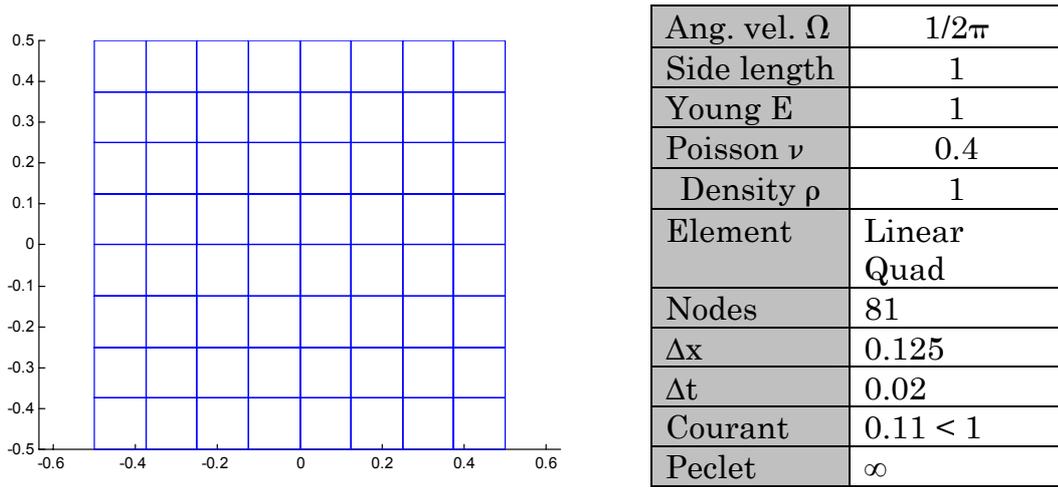


Figure n° 12: Original geometry

As it is possible to see from the figure below, the horizontal and the vertical displacements oscillate like a sinusoidal wave close to the rigid solution between  $\delta_{MAX} = 0.2071$  and  $\delta_{MIN} = -1.2071$ .

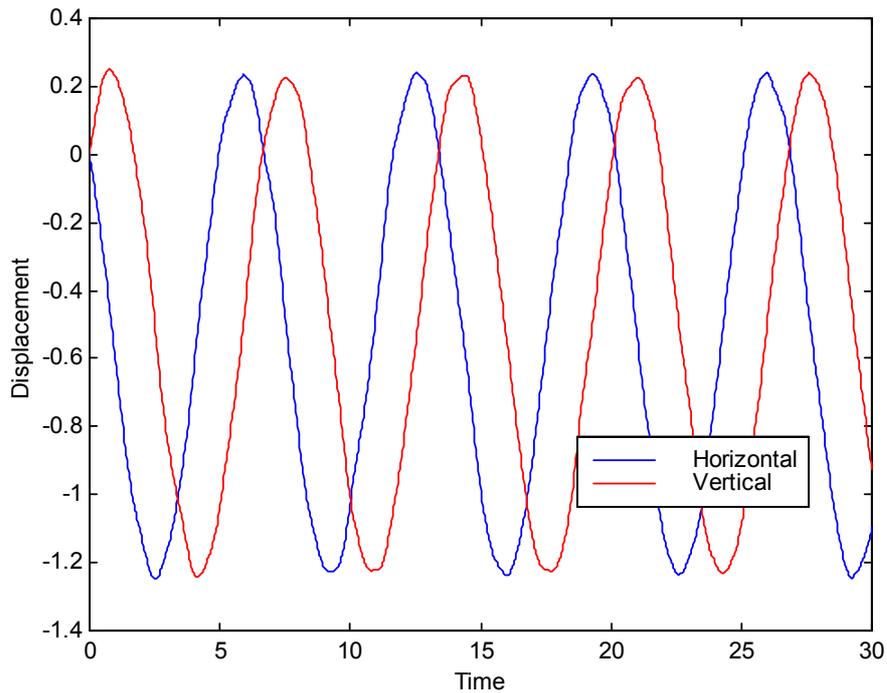


Figure n° 13: Evolution of displacement of the upper left corner

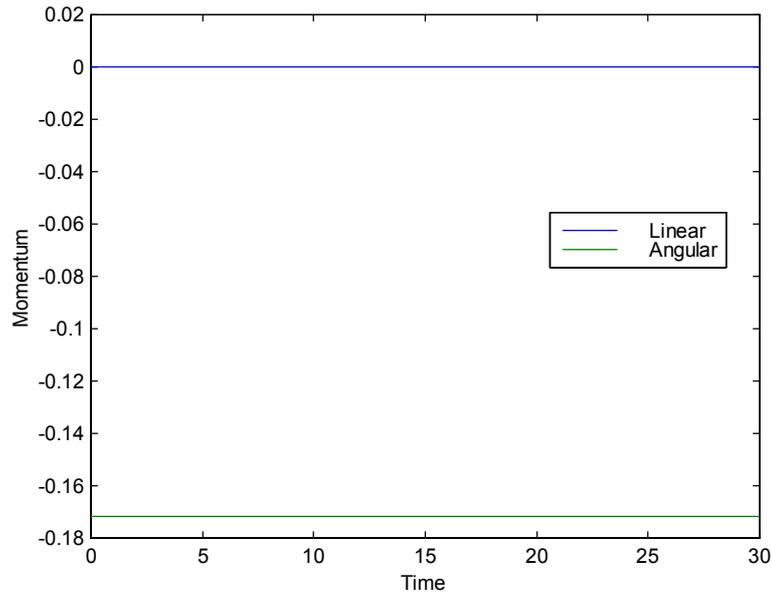


Figure n° 14: Evolution of displacement of the upper left corner

In the Figure n° 14 the modula of the linear momentum  $\mathbf{L}$  and the angular momentum  $\mathbf{H}$  (7.1) are plotted. No movement of the central node was appreciated (the linear momentum is null) and the variation of the angular momentum is close to the machine error.

$$\mathbf{L} = \int_{\Omega} \mathbf{p} d\Omega \qquad \mathbf{H} = \int_{\Omega} \mathbf{p} \times \mathbf{x} d\Omega \qquad (7.1)$$

Although it is not directly imposed, the energy is approximately kept and it oscillates around the exact value (Fig. 15).

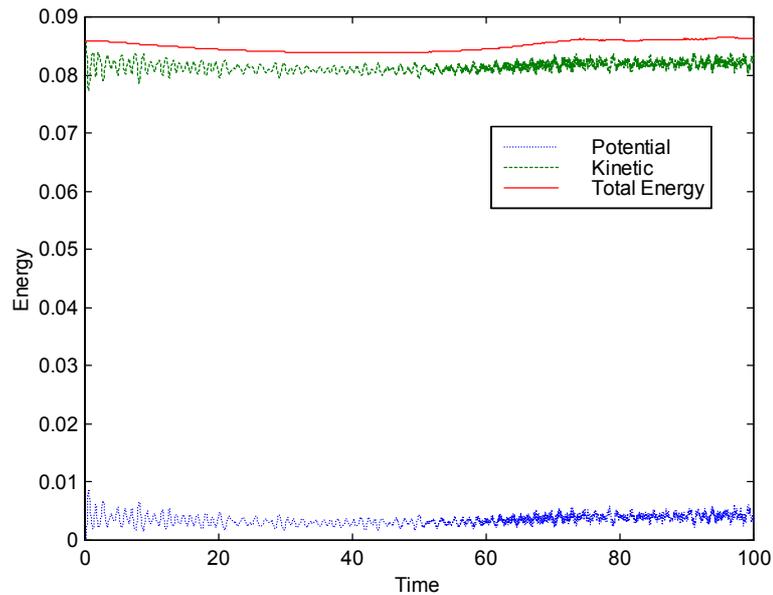


Figure 15: Evolution of energy for long time iterations

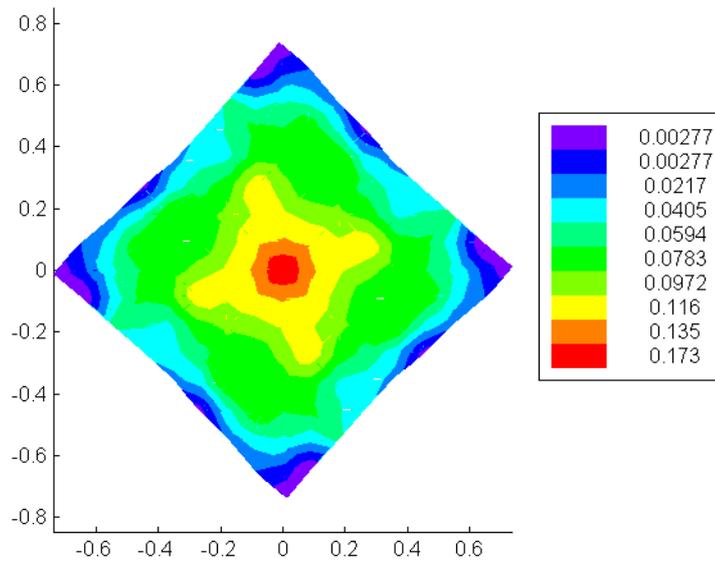


Figure n° 16: Pressure in the plate after 4.5 cycles. Deformed geometry

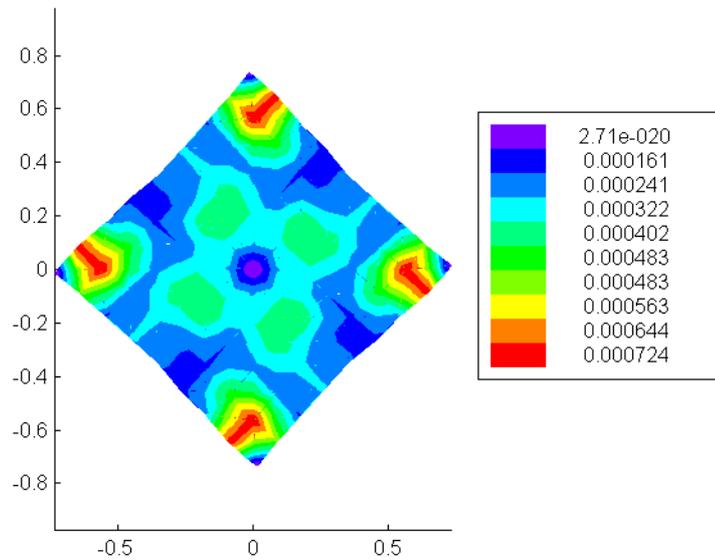
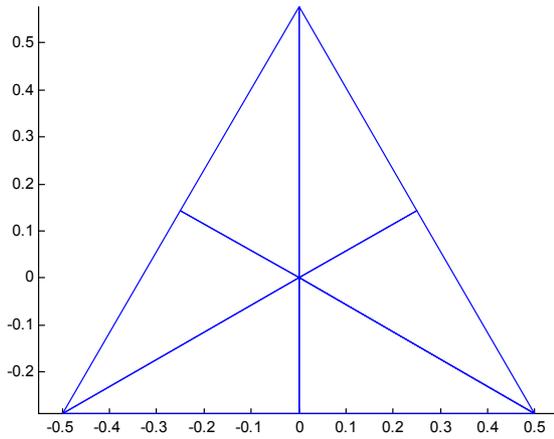


Figure 17: Absolute error in pressure after 4.5 cycles. Deformed geometry

The maximum jump of the pressure defined in the section 3.12 is approximately  $1/4$  on the boundary. The jump of the pressure in the centre is close to machine error.

### 7.2.2. Triangular plate

For an equilateral triangular plate with the same material properties of the previous plate under the same angular velocity, the results obtained are quite accurate even with a very coarse mesh.



Ang. vel. $\Omega$	$1/2\pi$
Side length	1
Young E	1
Poisson $\nu$	0.4
Density $\rho$	1
Element	Linear Triang
Nodes	25
$\Delta x$	0.25
$\Delta t$	0.03
Courant	$0.18 < 1$
Peclet	$\infty$

Figure 18: Original geometry

In this example, it becomes clear that the rotational error of the deformation gradient tensor  $\mathbf{F}$  is relevant though it remains limited due to the poor spatial discretization.

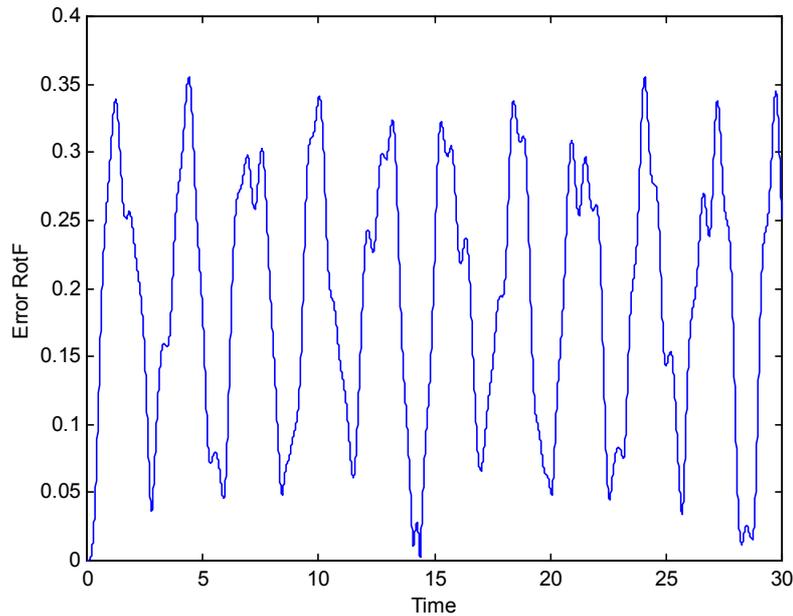
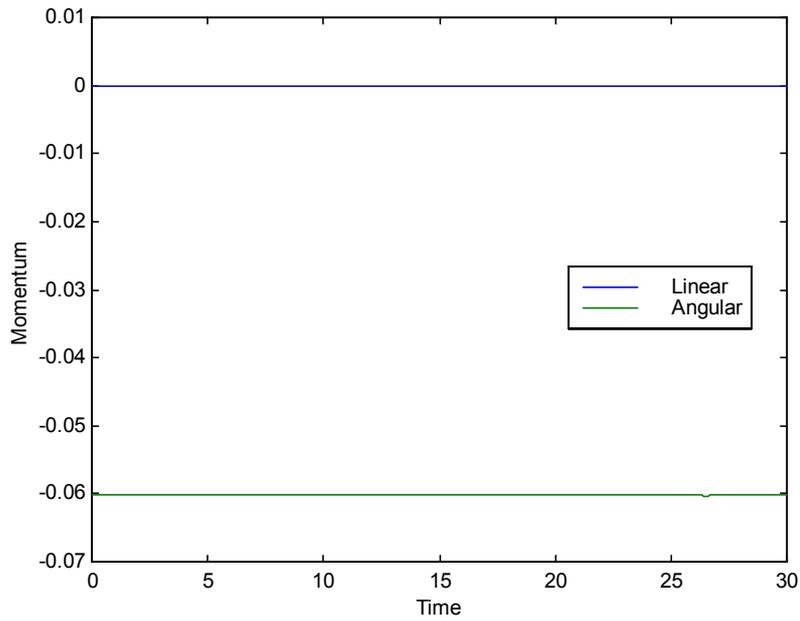
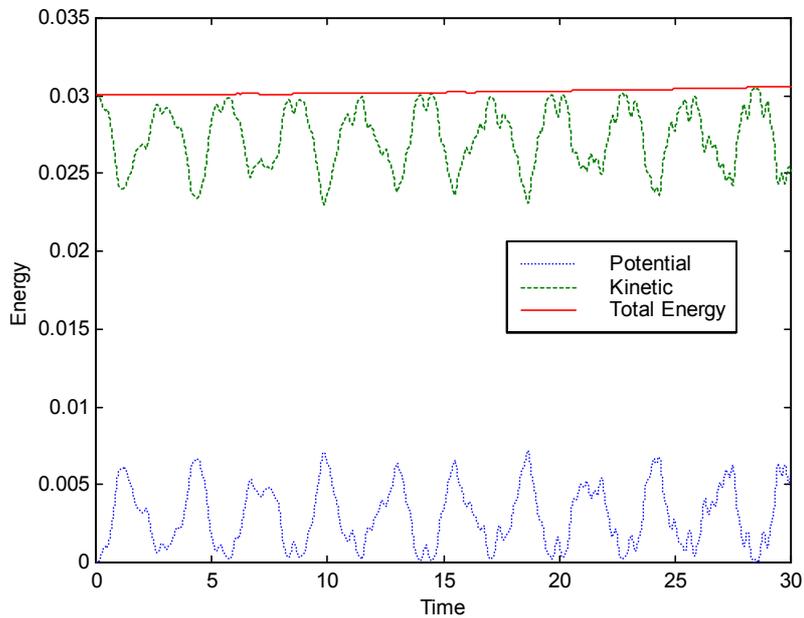


Figure n° 19: Evolution of rotational error of  $F$  with the coarse mesh

As expected, the momentum remains conserved exactly throughout the simulation as shown in Figure n° 20. The momentum calculated in each step was based on the previous expressions (7.1).



*Figure n° 20: Evolution of momentum*



*Figure n° 21: Evolution of energy*

The time history of energy is still quite good in the beginning but for long time iterations the solution is corrupted by the noise of the high frequencies. In the next section, 7.3, we propose to add an artificial viscosity to overcome this problem in very non-linear problems with coarse meshes.

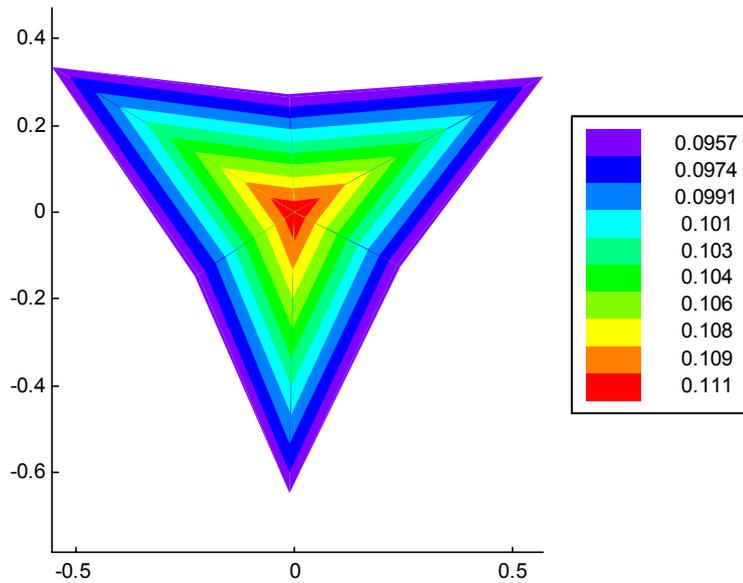


Figure n° 21: Pressure after 3.5 cycles

### 7.3. Deformation of a tube

As it was pointed in the previous section, the addition of a little amount of artificial viscosity can help to obtain smoother solutions because of the reduction of the noise due to the high frequencies. To show this effect, a long cylindrical tube has an initial deformation given by the deformation gradient tensor:

$$F = \begin{bmatrix} 1.20 & 0 \\ 0 & 0.80 \end{bmatrix}$$

Then, the tube oscillates freely.

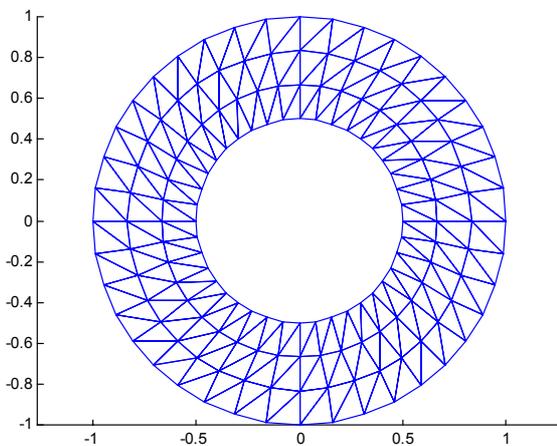
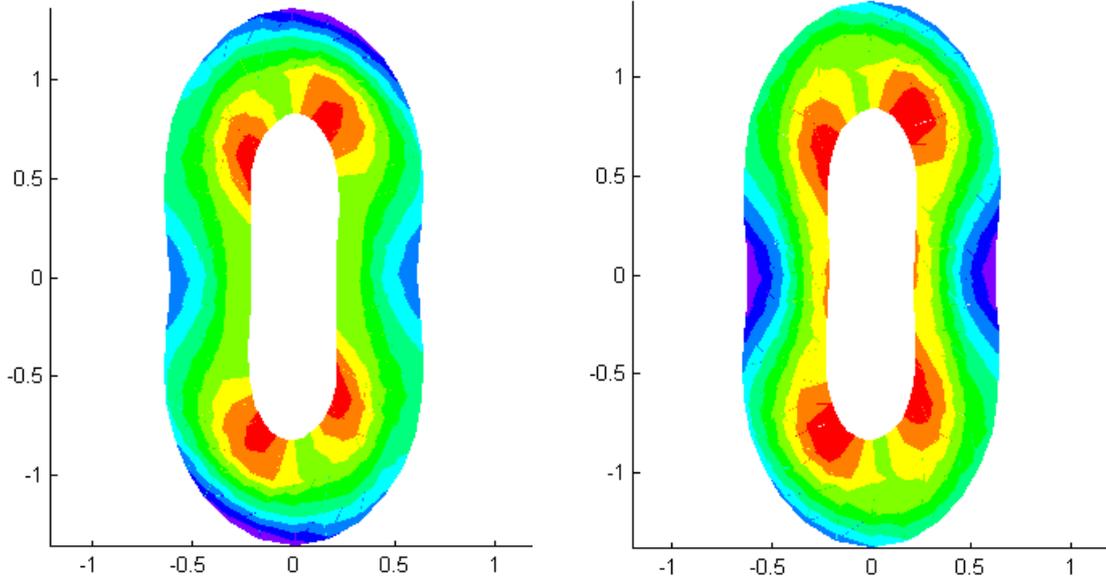


Figure n° 22: Original geometry

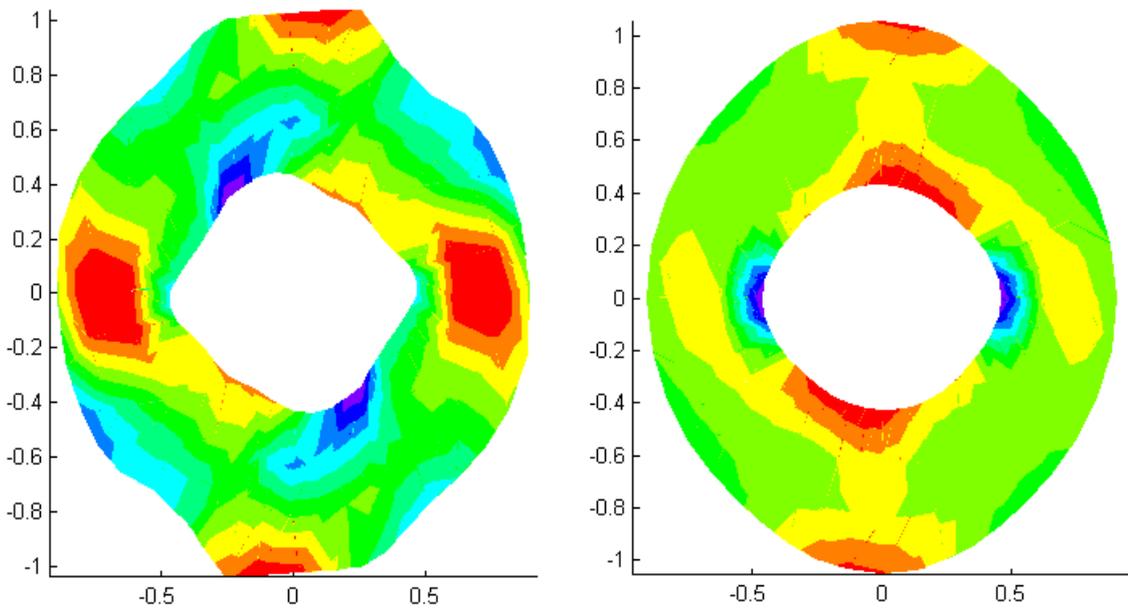
External Radius	1
Internal Radius	0.5
Young E	1
Poisson $\nu$	0.4
Density $\rho$	1
Element	Linear Quad 2D
Nodes	160
$\Delta x$	$\sim 0.1$
$\Delta t$	0.01
Courant	$0.15 < 0.98$
Peclet	$\sim 50$

The chosen mesh is skew symmetric (Fig. 22).

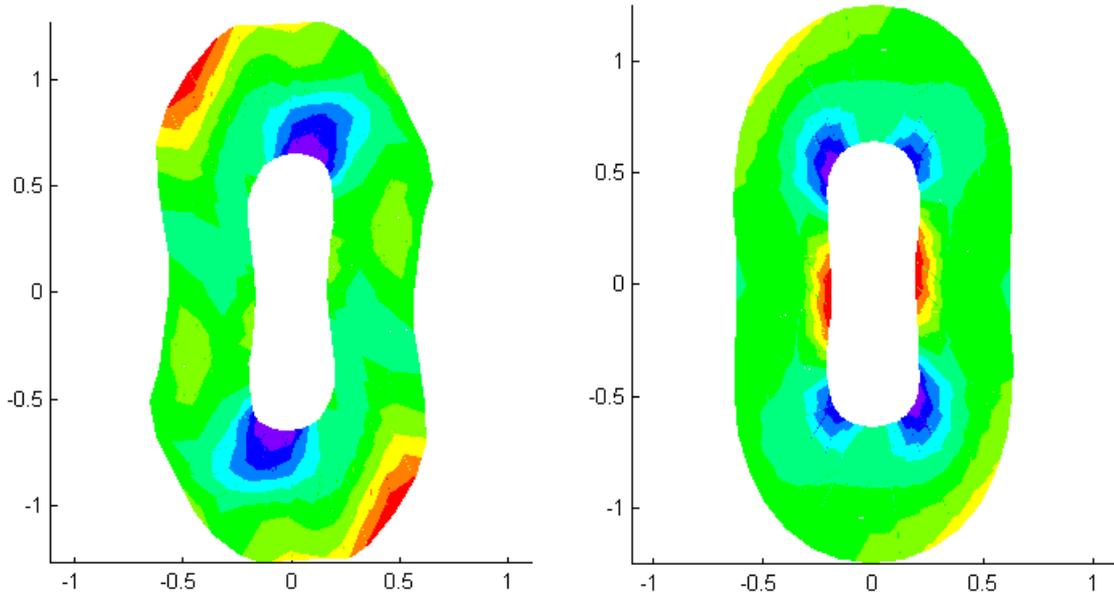
*Figures n° 23 Deformed geometry at different times.  
 Pressure Red = High traction, Blue = High compression  
 Without numerical damping (left)      With some artificial viscosity (right)*



*Time=5 (1/4 of cycle).*

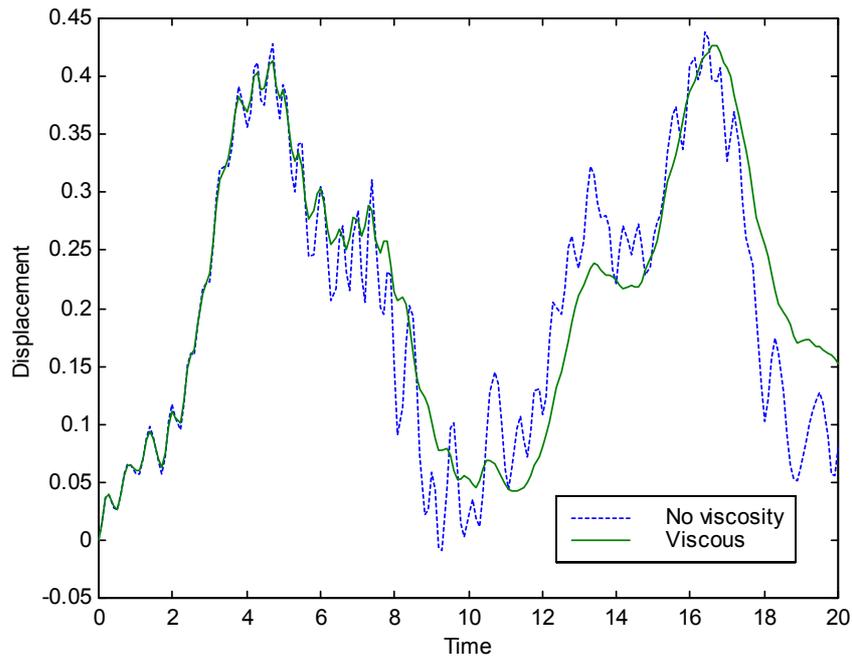


*Time=10 (1/2 of cycle).*



*Time=15 (3/4 of the cycle).*

With the inviscid model, the solution is affected by different modes of vibration. Adding a little amount of viscosity the cylindre vibrates mainly according to the first mode. Shock waves present high frequency components which can be artificially smoothed by the numerical scheme (Fig. 24). If such is the case, the stress intensity will be underpredicted. It is therefore important to use numerical algorithms with optimal damping and dispersion properties. Also to be outlined is that the tube remains centred at the same point and without rotations. This fact demonstrates the conservation of linear and angular momentum in the inviscid and viscous cases.



*Figure n° 24: Evolution of displacement of the apex of the disk*

#### 7.4. Deformed plate with analytical solution

We consider a square flat plate of unit side length under plane strain. The left and bottom boundaries were restricted to move only tangentially, whereas the top and right boundaries are restricted to move normally, as shown in the Figure n° 25.

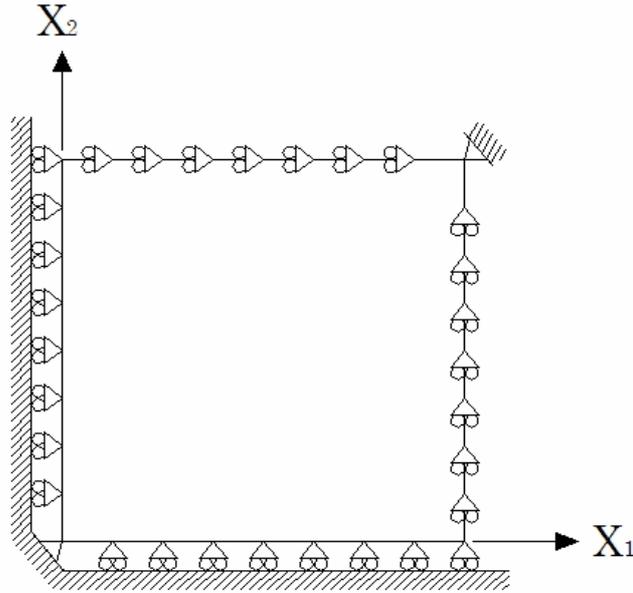


Figure n° 25: Test case

Under the assumption of small displacements, this is

$$\mathbf{u} = \mathbf{x} - \mathbf{X},$$

the equation that describes the movement is the classical Euler–Lagrange equation for plane strain.

$$(\lambda + \mu)\text{div}(\mathbf{u}) + \mu\Delta\mathbf{u} = 0 \tag{7.2}$$

The solution for the given boundary conditions is (7.3).

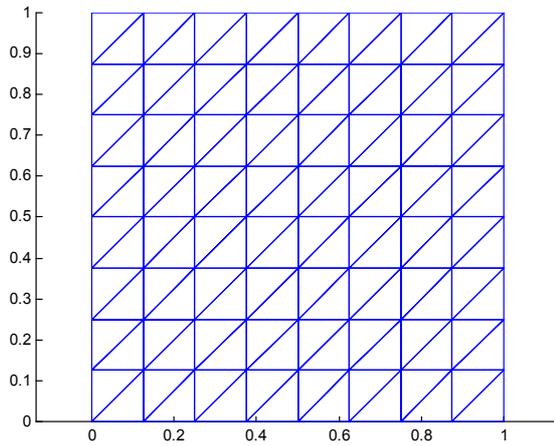
$$\mathbf{u} = u_0 \cos\left(\frac{c_d \pi t}{2}\right) \begin{bmatrix} \sin\left(\frac{\pi X_1}{2}\right) \cos\left(\frac{\pi X_2}{2}\right) \\ -\cos\left(\frac{\pi X_1}{2}\right) \sin\left(\frac{\pi X_2}{2}\right) \end{bmatrix} \tag{7.3}$$

The initial conditions are (7.4) and (7.5).

$$\mathbf{p}_0 = \mathbf{0} \tag{7.4}$$

$$\mathbf{F} = \mathbf{I} + \nabla_x \mathbf{u} \tag{7.5}$$

For values of the maximum initial displacement  $u_0$  below  $1e-3$ , there is no appreciable difference between the analytical solution and the numerical one. The code was run for the model which is summarized in the chart.



External Radius	1
Internal Radius	0.5
Young E	1
Poisson $\nu$	0.4
Density $\rho$	1
Element	Linear Quad 2D
Nodes	81
$\Delta x$	$1/8\sqrt{2}$
$\Delta t$	0.02
Courant	$0.15 < 1$
Peclet	$\infty$

Figure n° 26: Original geometry

The average element rotational of  $\mathbf{F}$  is small and bounded (Fig. 27).

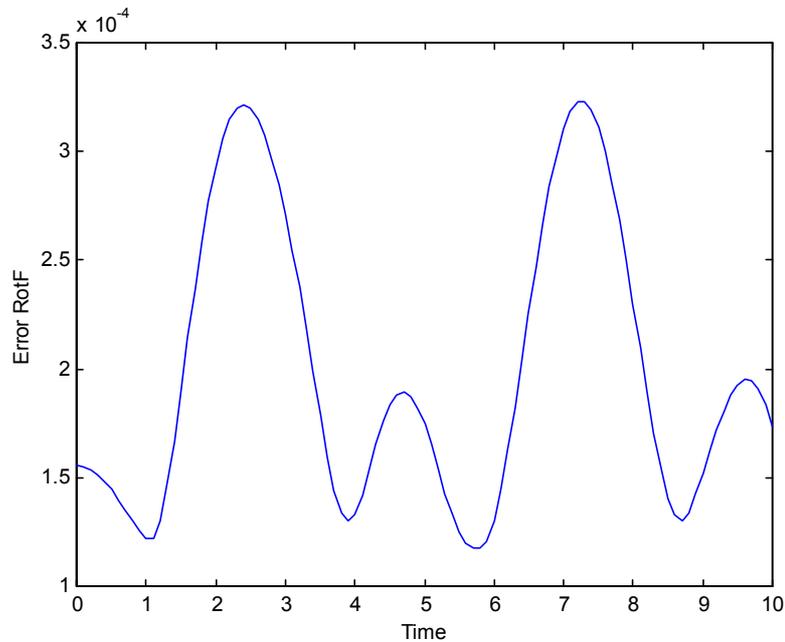


Figure n° 27: Evolution of rotational error

The total internal energy is perfectly maintained (Fig. 28) provided that no external forces are applied. The method possesses good energy conservation which makes it adequate for long time simulations.

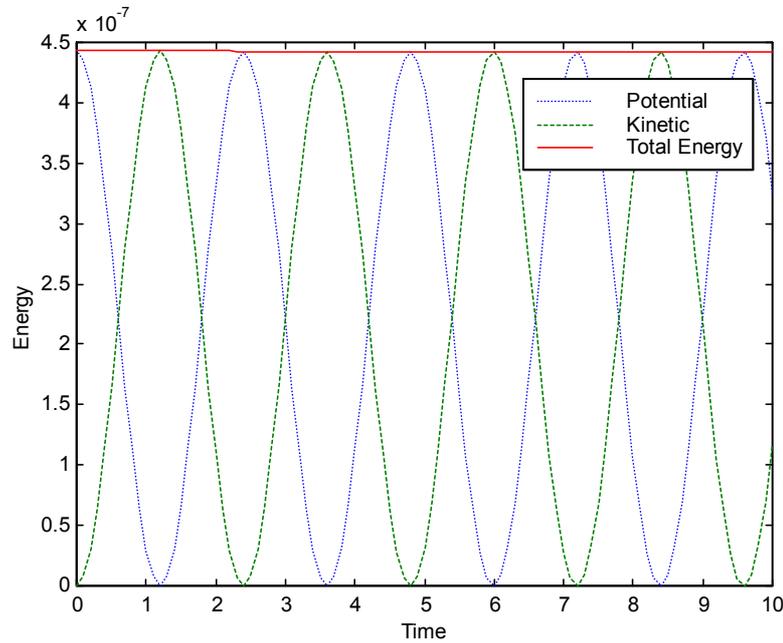


Figure n° 28: Evolution of the energy

The momentum is not constant but the average level of linear momentum and angular momentum of the plate respect to the origin in each period ( $T = 4.7328$ ) does not decay or increase.

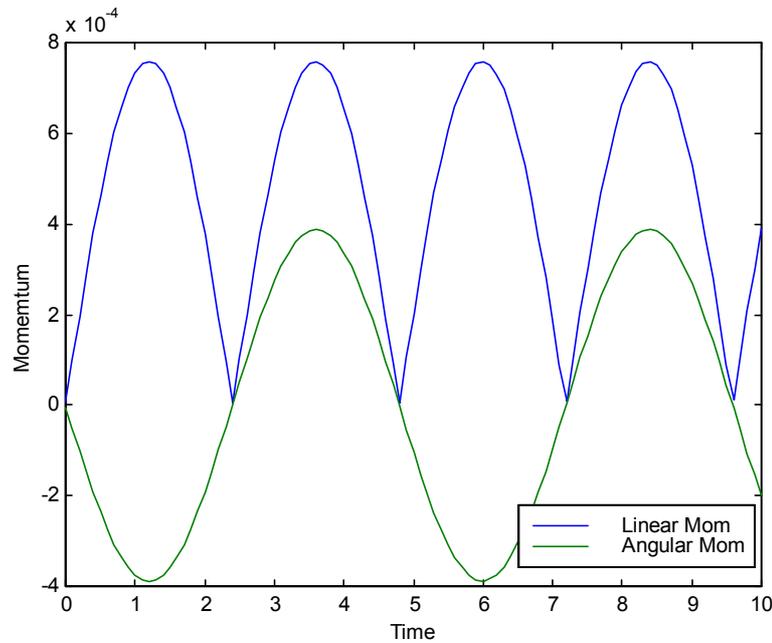


Figure n° 29: Evolution of the momentum

Even for a not very fine mesh, the result is quite accurate. The error of the peak displacement error is approximately +6% and the period elongation error was also around +3% (Fig. 30). The distribution of displacements was the expected one (Fig. 31).

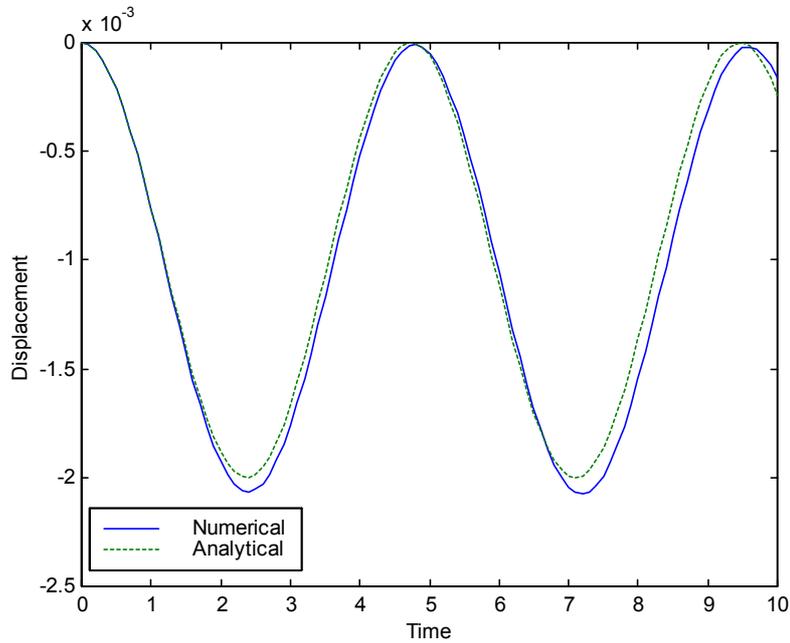
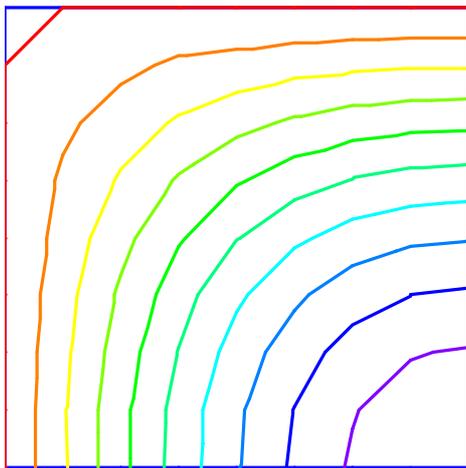
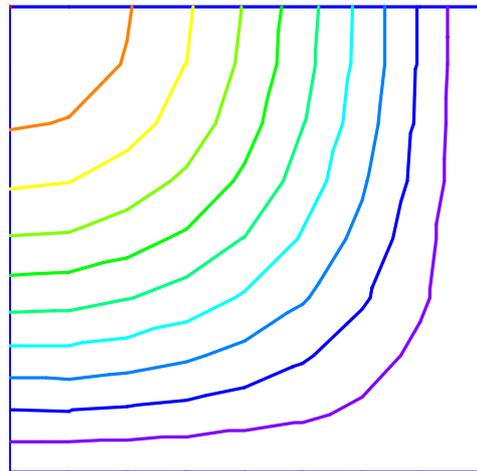


Figure n° 30: Evolution of the horizontal displacement of the bottom right corner



Blue: minimum horizontal displ =  $-2e-3$   
 Red: maximum horizontal displ = 0



Blue: minimum vertical displacement = 0  
 Red: maximum vertical displacement =  $2e-3$

Figure n° 31 : Distribution of displacement at time = 2.3664 ( $\frac{1}{2}$  cycle)

### 7.5. Buckling of a beam

This example shows the effect of the Poisson modulus. A bar is axially loaded only on half of the head with a uniform horizontal pressure  $F_0 = 0.01$ .

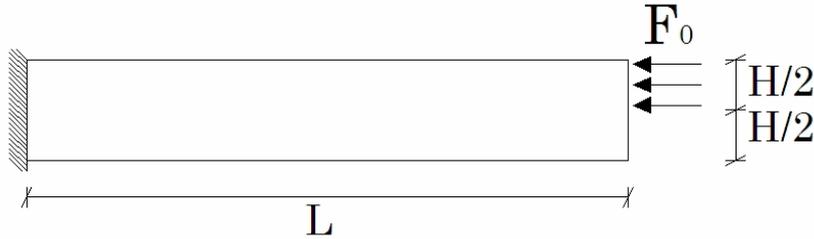
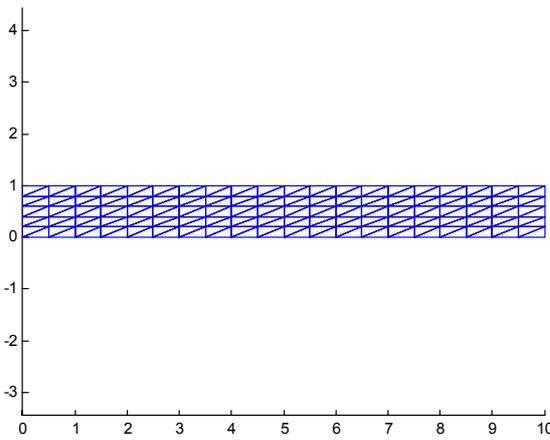


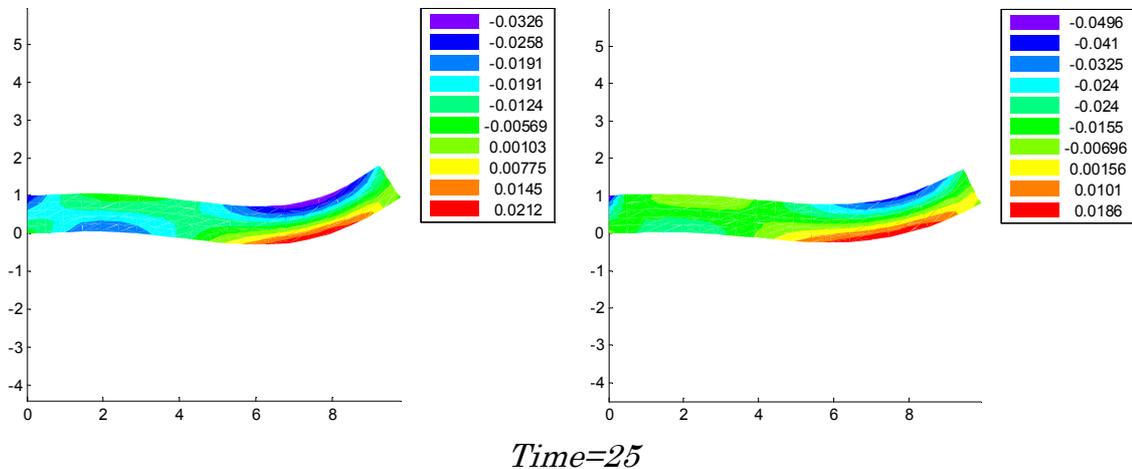
Figure n° 32: Sketch of the problem

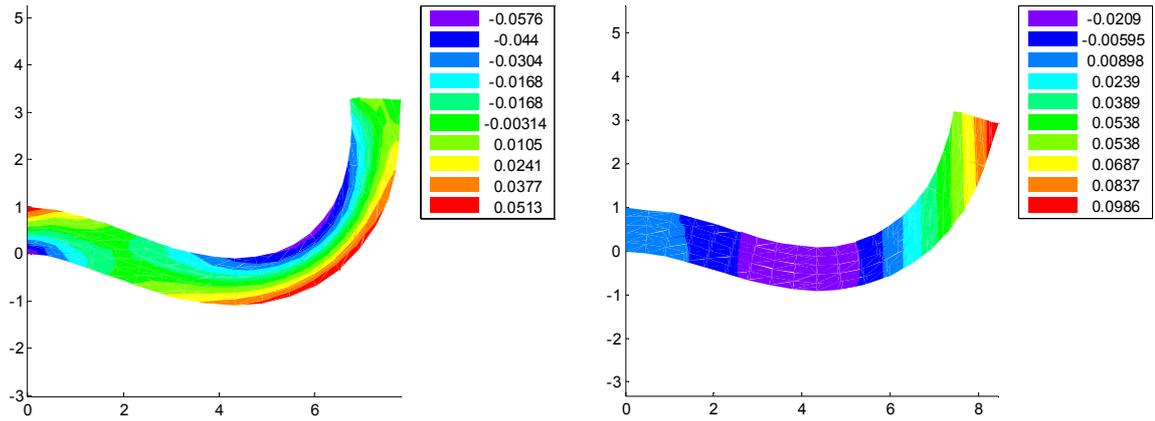


Length L	10.0
Height h	1
Young E	1
Density $\rho$	1
Element	Linear Triang 2D
Nodes	126
$\Delta x$	0.186
$\Delta t$	0.01
Courant	0.11 < 1
Peclet	$\infty$

Figure n° 33: Original geometry

Figure n° 34: Pressure on a buckling beam at different moments  
 Compressible material  $\nu = 0.2$       Nearly incompressible material  $\nu = 0.45$





Time=50

The Figure n° 34 shows that there is no volumetric locking when using nearly incompressible materials. It is well known that simple triangles with the classic displacement formulation exhibit a locking effect and the structures are much stiffer.

### 7.6. Optimum viscosity

#### 7.6.1. 1-D bar problem

For an axially loaded bar, the peaks of the kinetic energy were represented for different viscosities in order to determine an optimum Peclet number. When the artificial damping is increased, the kinetic energy is dissipated in a few steps but the allowed time step becomes smaller.

The data of the problem are  $L=10m$ ,  $E=1$ ,  $Poisson=0.4$ , load  $F_0=0.1$ . A discretization of 30 two–node elements was chosen to plot the Figure n° 36.

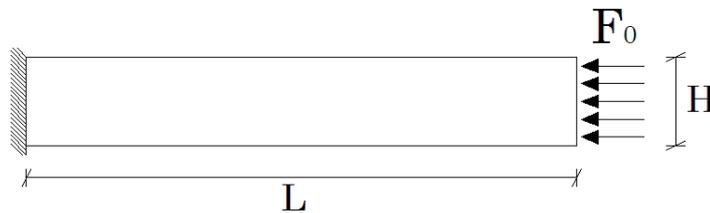


Figure n° 35: Sketch of the problem

If we use a time step close to the limit of stability a good estimation of the viscous parameters is by using the non-dimensional parameter  $\alpha$  in (7.5) that was fully described in the Chapter n° 5.

$$\lambda_{VIS} = \alpha \frac{\lambda h_E}{c} \quad \mu_{VIS} = \alpha \frac{\mu h_E}{c} \quad (7.5)$$

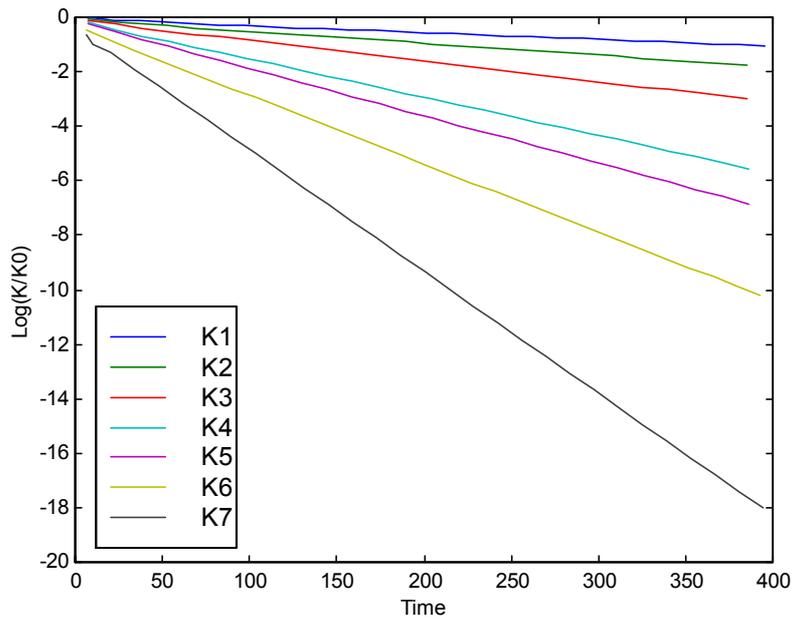


Figure n° 36: Dissipation of kinetic energy for different viscosities in 1-D

The variation of the kinetic energy fits well a typical exponential decrease in time.

Case	Pecllet	Time to 10% K0	$\Delta t$ max
1	7.2	-	1.2e-1
2	3.5	351	1.1e-1
3	2.3	246	9.4e-2
4	1.2	142	6.7e-2
5	1.0	114	5.9e-2
6	0.7	77.3	4.5e-2
7	0.4	49.2	2.8e-2

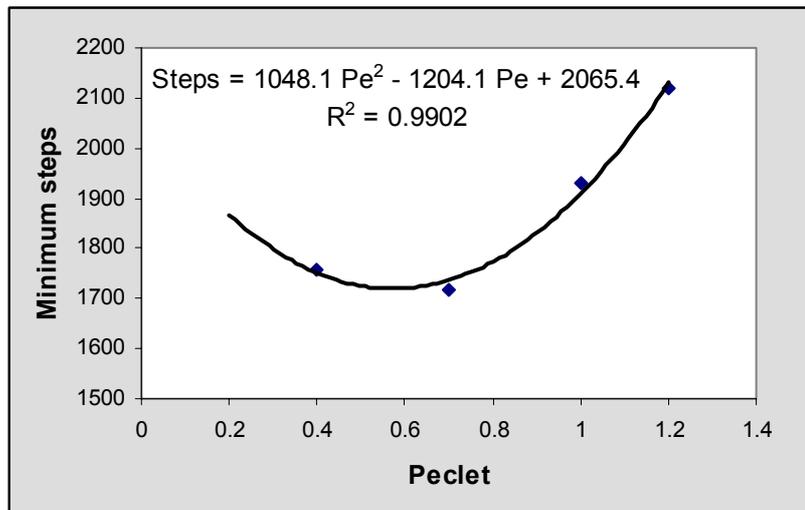


Figure n° 37: Optimum Pecllet number for the 1-D problem

The optimum Peclet number that minimizes the number of required steps is around  $Pe=0.57$  (Fig. 37). The Figure n° 38 represents the effect of the number of elements in the dissipation for  $\alpha=1.75$ , which is close to the optimum parameter. For the same Peclet number the dissipation is more effective in coarse meshes. In very fine meshes the viscosity is less necessary and can be neglected. It has to be noted that for all the chosen Peclet numbers, the formulation remains variationally consistent.

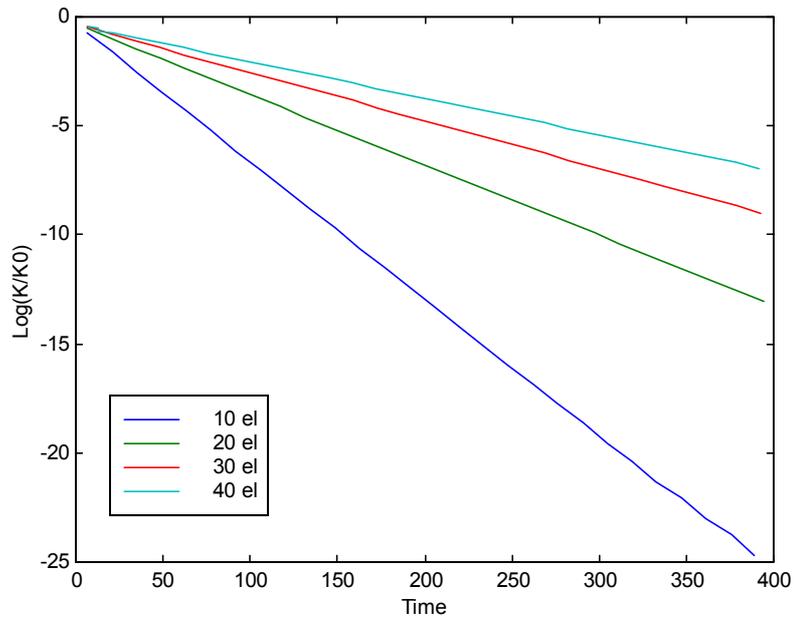


Figure n° 38: Influence of mesh size in the dissipation

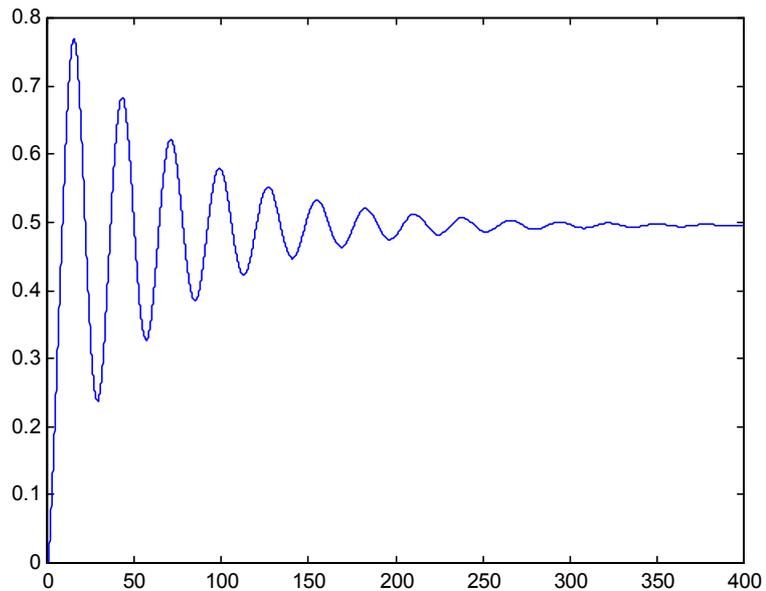


Figure n° 39: Evolution of displacement in the head of the bar. 10 elements and  $Pe=0.57$ .

**7.6.2. 2-D bar problem**

The same problem previously described was modelled with constant strain linear triangles.

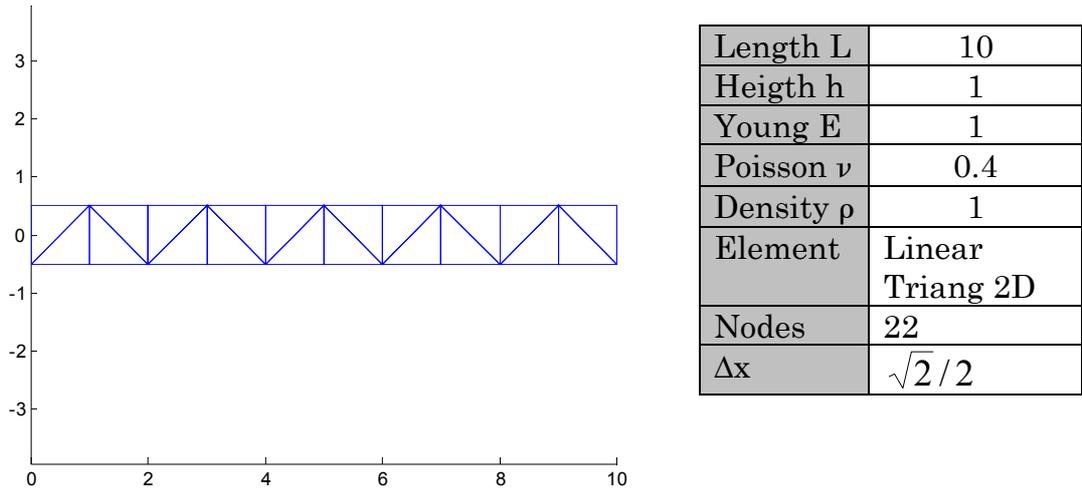


Figure n° 40: Original geometry

The model was run using several artificial viscosities (Fig. 41).

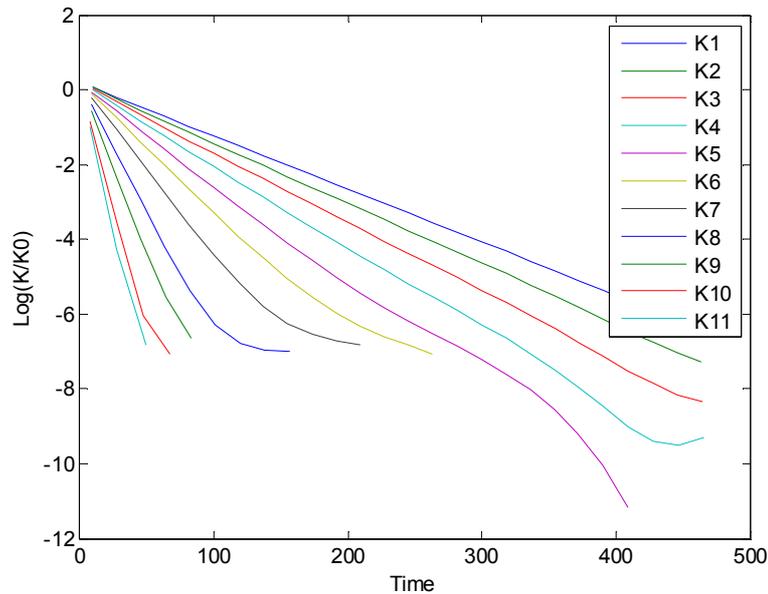


Figure n° 41: Dissipation of kinetic energy for different viscosities in 2-D

The oscillations at the end of the dissipation lines of the graph are probably due to the inaccuracy comparing the peaks of energy when they are very small.

Case	Peclet	Time to 1% K0	$\Delta t$ max
1	1	337.8	0.20
2	0.875	298.2	0.18
3	0.75	258.3	0.16
4	0.625	216.6	0.14
5	0.5	174.6	0.11
6	0.4	139.9	9.3e-2
7	0.3	105.0	7.1e-2
8	0.2	69.98	4.8e-2
9	0.15	53.00	3.6e-2
10	0.1	35.50	2.4e-2
11	0.085	30.30	2.0e-2

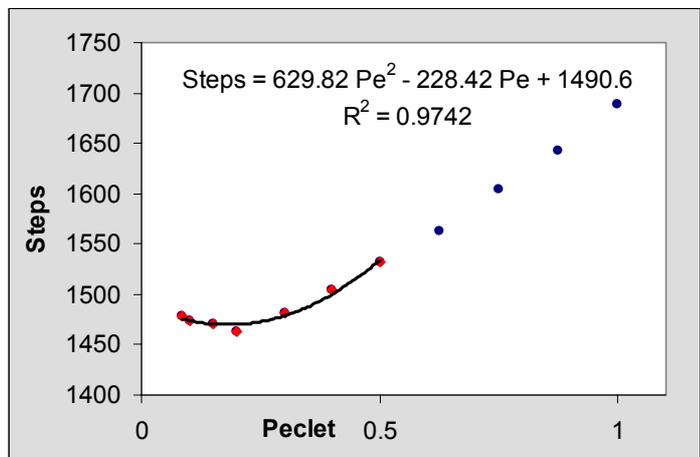


Figure n° 42: Optimum Peclet number for the 2-D problem

So, the Peclet number that optimizes the number of required steps is approximately **Pe=0.18**. Comparing to 1-D, the ideal viscosity in two-dimensional problems seems to be higher.

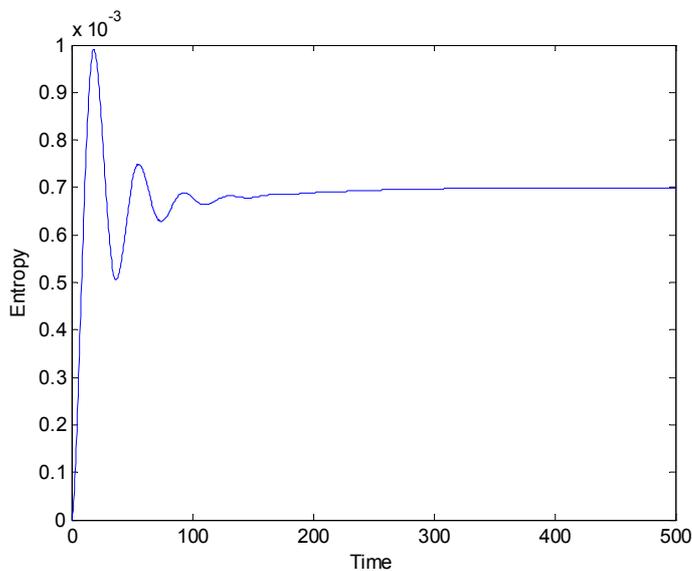


Figure n° 43: Evolution of the entropy-like energy variable

For  $\alpha = 5.5$ , close to the optimum parameter, the results are shown in the Figures 43, 44 and 45. The entropy-like energy variable and the internal energy (Fig. 43 and 44) stay constant after the main dissipation of the kinetic part.

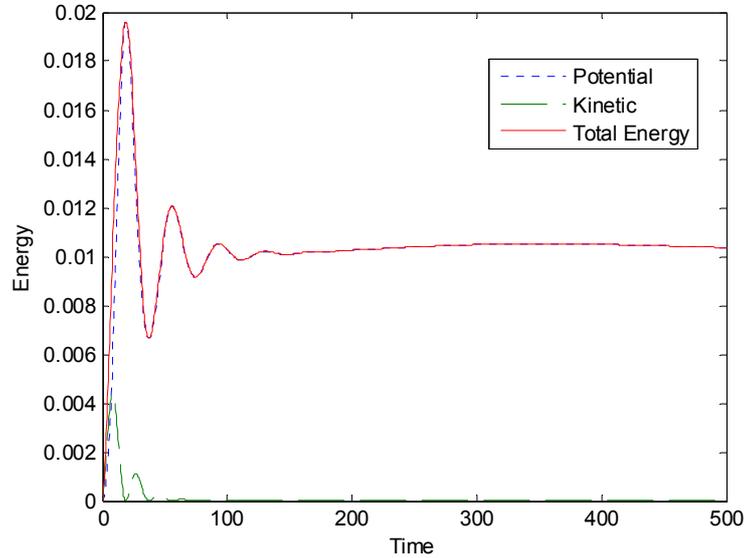


Figure n° 44: Evolution of the energy

After a couple of oscillation cycles the kinetic component of the energy has been reduced by the viscous dissipation and the static displacement is obtained. Assuming small deformations the analytical static displacement can be calculated like this

$$\delta_{STAT} = \frac{F_0 L}{\left(\frac{4}{3}\mu + \kappa\right)A} = \frac{F_0 L}{(\lambda + 2\mu)A} = 0.4666 \dots$$

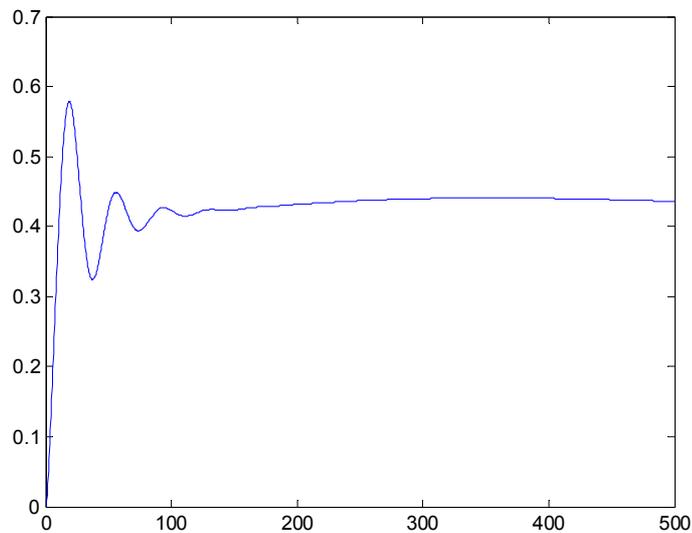


Figure n° 45: Evolution of the displacement in the head of the bar

The numerical results, Fig. 39 and Fig. 45, basically coincide with the analytical estimation.

### 7.7. Bending of an infinite rectangular plate

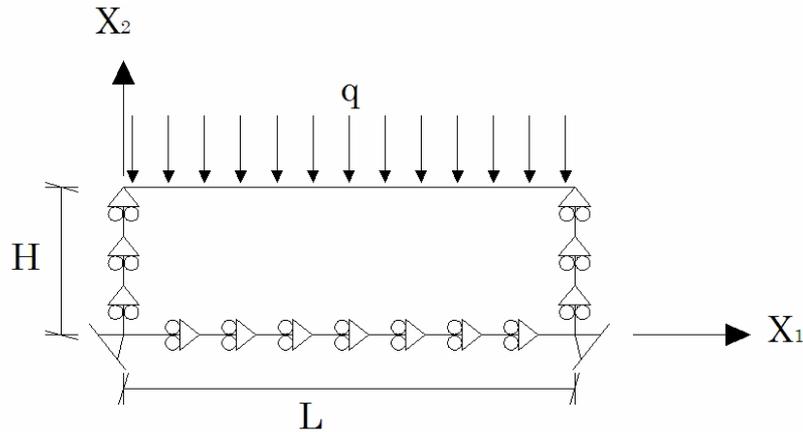


Figure n° 46: Skech of the problem

An infinite simply supported plate is uniformly loaded under plane strain conditions. The boundary conditions are:

- Lateral faces: only horizontal displacements are permitted.
- Bottom face: only vertical displacements are permitted.
- Top face: uniform load  $q$  applied.

The objective of this example is to prove that the method can be valid in both dynamic and stationary bending dominated situations.

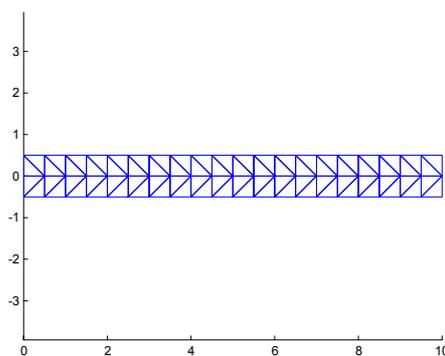


Figure n° 47: Original geometry

Load $q$	$5e-6$
Length $L$	10
Height $h$	1
Young $E$	1
Poisson $\nu$	0.4
Density $\rho$	1
Element	Linear Triang 2D
Nodes	63
$\Delta x$	0.35
$\Delta t$	$5e-3$
Courant	$0.021 < 0.086$
Peclet	0.17

The energy is stabilized after four or five cycles (Fig. 48 and 49) and the velocity goes to zero (Fig. 50). The solution obtained with the viscous two-step TG seems to be more flexible and a bit more accurate than the classical FEM formulation for the same mesh (Fig. 51).

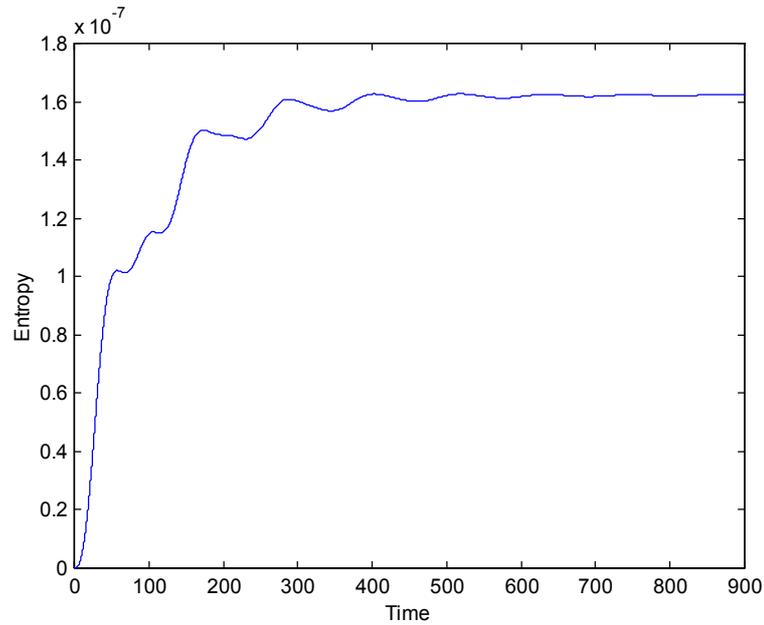


Figure n° 48: Evolution of entropy-like energy variable

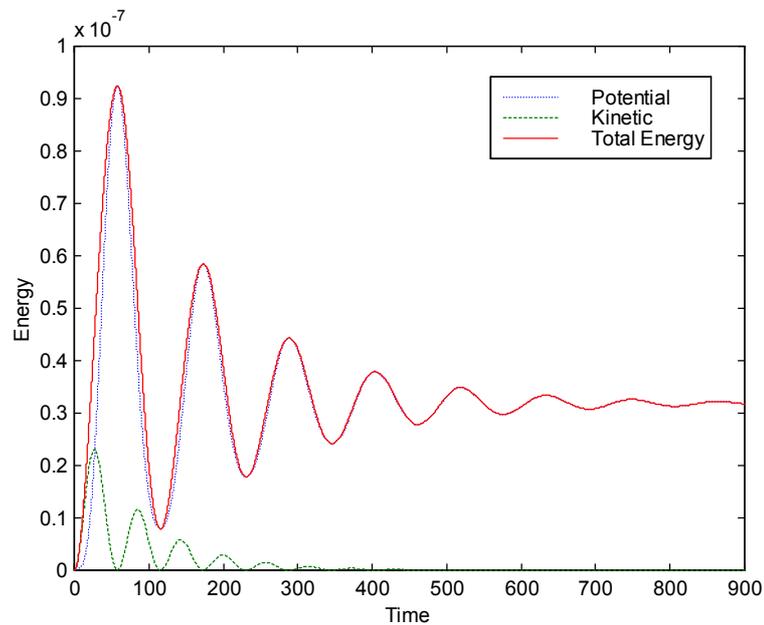


Figure n° 49: Evolution of energy

Furthermore, it is important that the deflection is not underestimated. As a consequence, the natural period of vibration is lower.

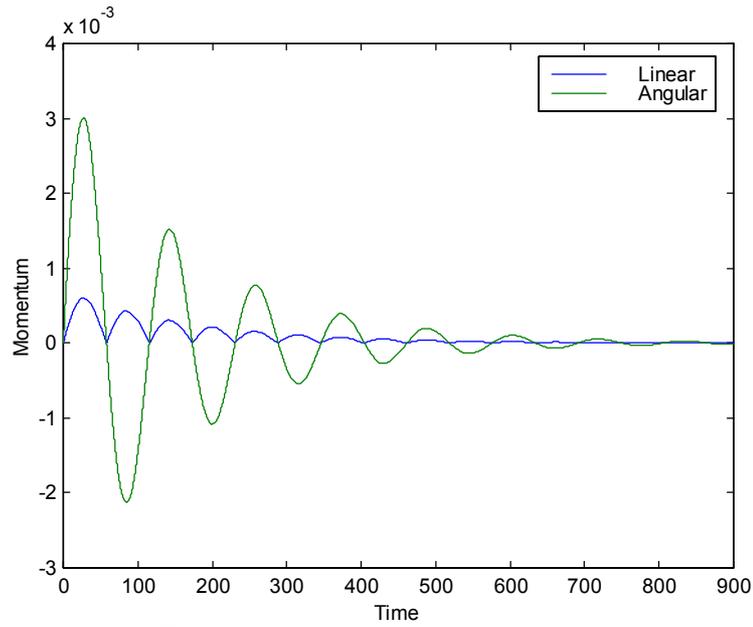


Figure n° 50: Evolution of momentum

The following chart summarizes the static results of the Figure n° 51.

Method	Deflection
Viscous TG 80 elem	2.101e-3
FEM 80 elements	1.397e-3
FEM 10000 elem	1.818e-3

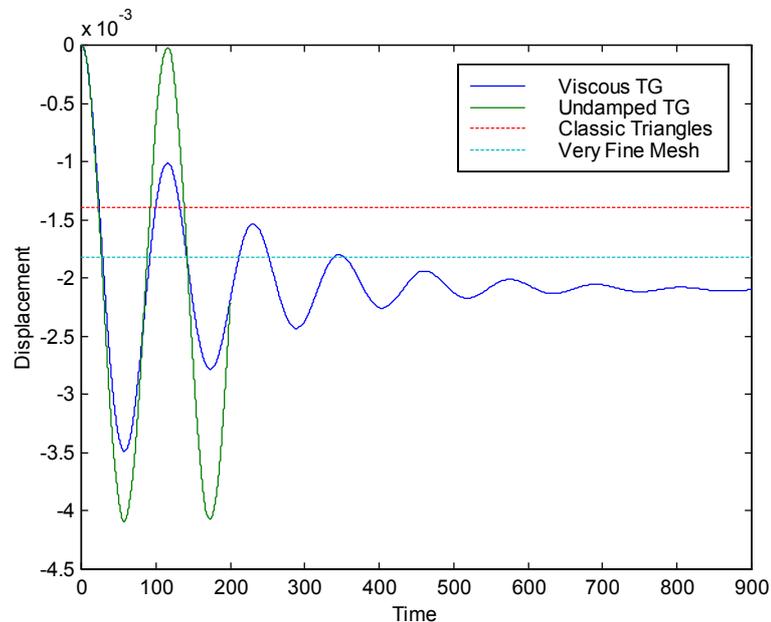


Figure n° 51: Evolution of displacement

The theoretical dynamic amplification factor can be obtained using the Duhamel integral (7.6). For linear to constant loads the maximum DAF is given by (7.7).

$$DAF = \frac{\delta}{\delta_{STAT}} = \int_0^t \sin \omega(t - \tau) d\tau \quad (7.6)$$

$$DAF_{MAX} = 1 + \frac{|\sin(\omega t_0 / 2)|}{\omega t_0 / 2} \quad (7.7)$$

If the application time  $t_0$  is small enough, then  $DAF_{MAX} = 2$ . In our problem we have chosen  $t_0 = 0.01$ . The numerical maximum DAF was only 2.65% lower than the theoretic value.

$$DAF_{MAX} = \frac{4.090e-3}{2.101e-3} = 1.947 \approx 2 \quad (7.8)$$

### 7.8. Flexible foundation

A uniform load  $q$  is applied linearly in  $t_0 = 0.01$  units of time between  $X_1=0.5$  and  $X_1=1.0$ ; the medium is symmetric around the edge  $X_1=1$  and is supported by the sides  $X_1=0$  and  $X_2=0$ ;

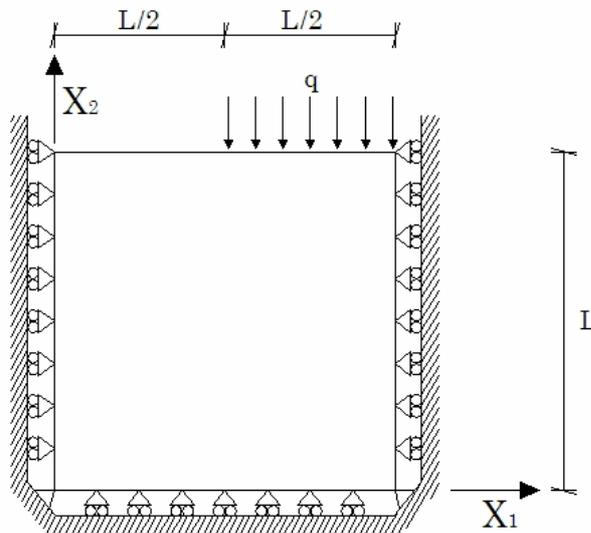
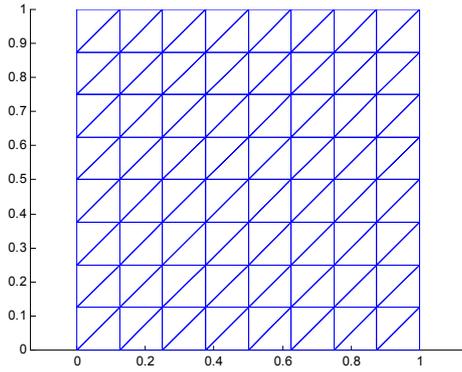


Figure n° 52: Sketch of the problem

The main aim of this case is to demonstrate that the proposed scheme can give safe dynamic and static stresses for the design of foundations and structures. Entropy-like energy and internal energy are stable after two cycles of oscillation (Fig. 54 and 55), the momentum tends to zero because of the dissipation of the kinetic term (Fig. 56) and the steady state is achieved quickly.



Load $q$	$5e-3$
Length $L$	1
Young $E$	1
Poisson $\nu$	0.4
Density $\rho$	1
Element	Linear Triang 2D
Nodes	81
$\Delta x$	0.0884
$\Delta t$	$1e-3$
Courant	$0.017 < 0.086$
Peclet	0.1725

Figure n° 53: Original geometry

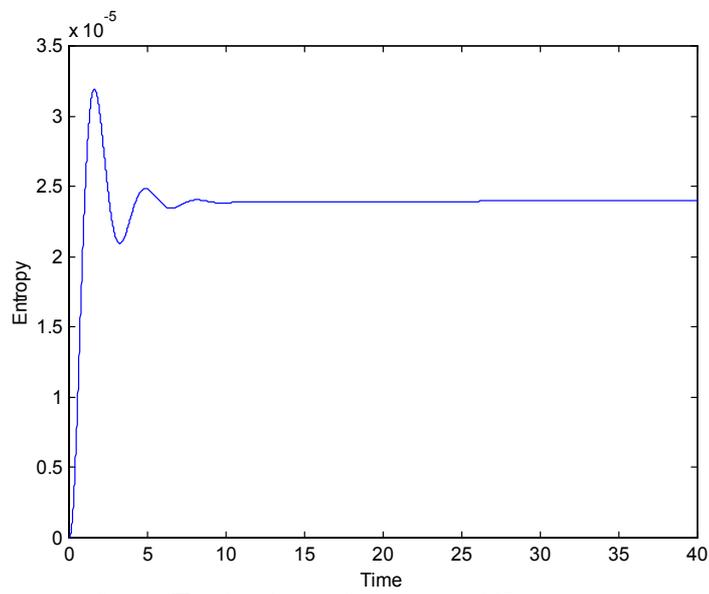


Figure n° 54: Evolution of entropy like energy variable

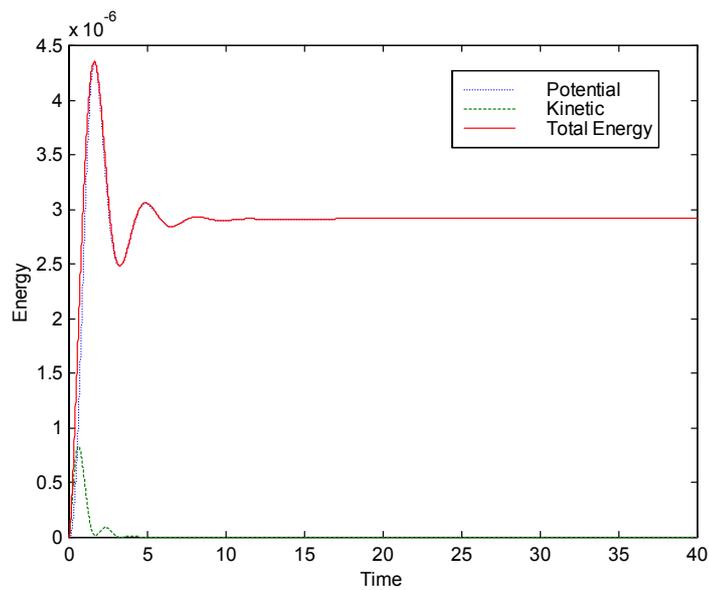


Figure n° 55: Evolution of energy

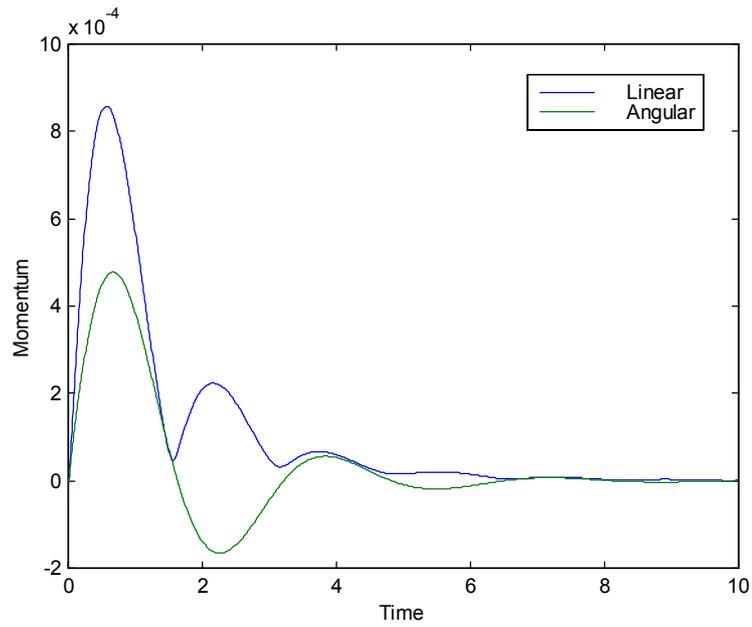


Figure n° 56: Evolution of momentum

The static results of the Figure n° 57 are summed up in the following table.

Method	Deflection
Viscous TG 128 elem	2.628e-3
FEM 128 elements	2.586e-3
FEM 6800 elements	2.608e-3

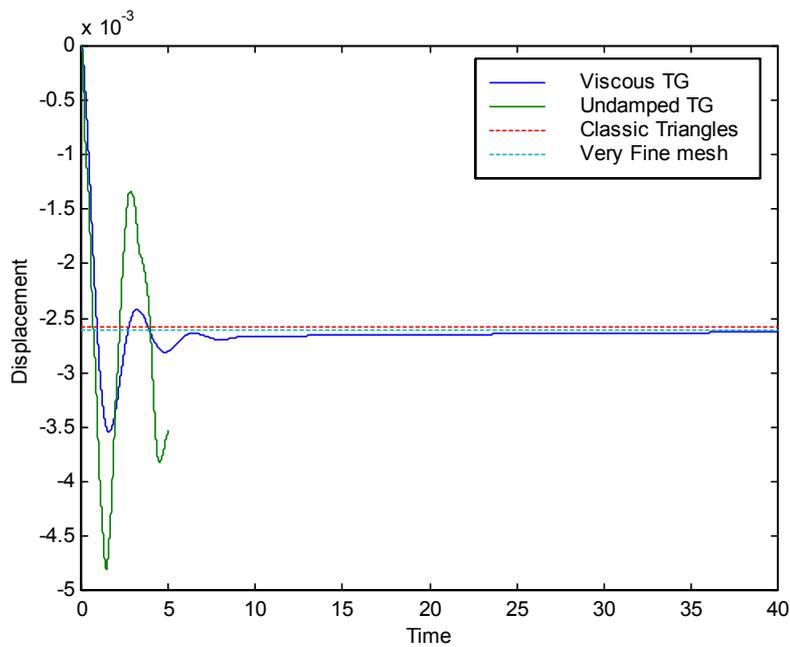


Figure n° 57: Evolution of displacement

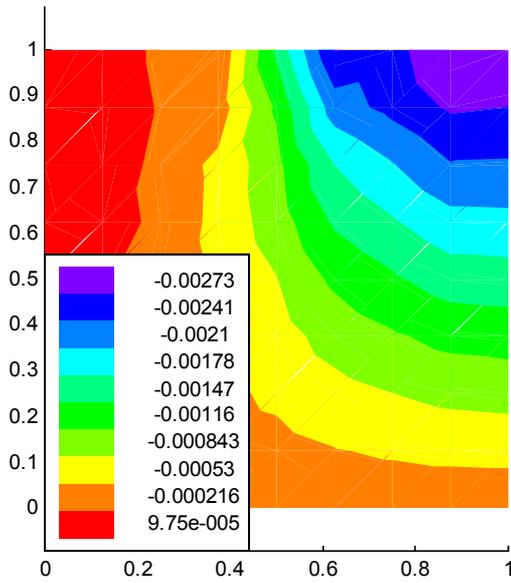


Figure n° 58.1: Vertical displacement

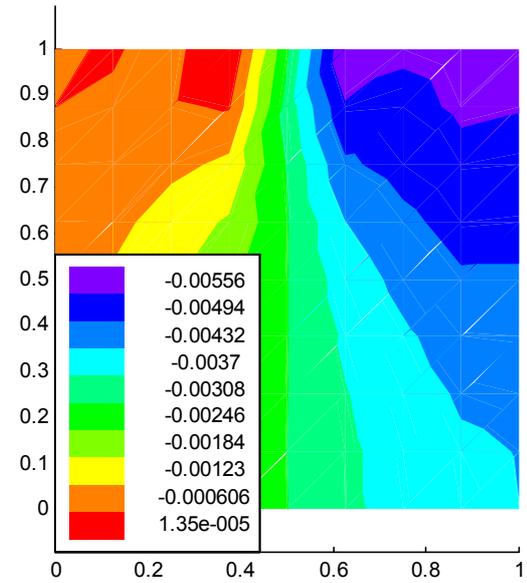


Figure n° 58.2: Vertical Cauchy Stress Y

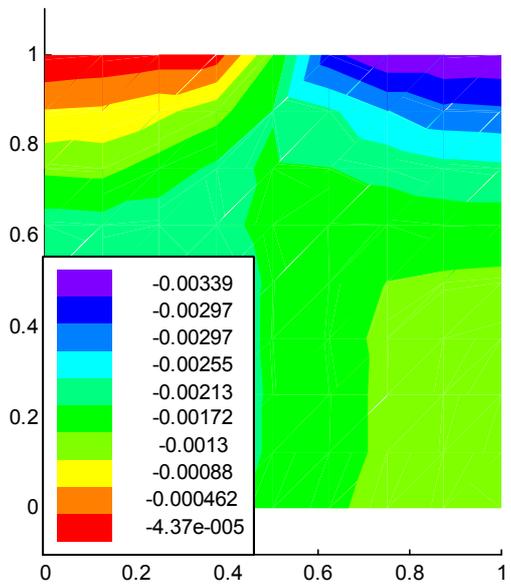


Figure n° 58.3: Horizontal Cauchy Stress

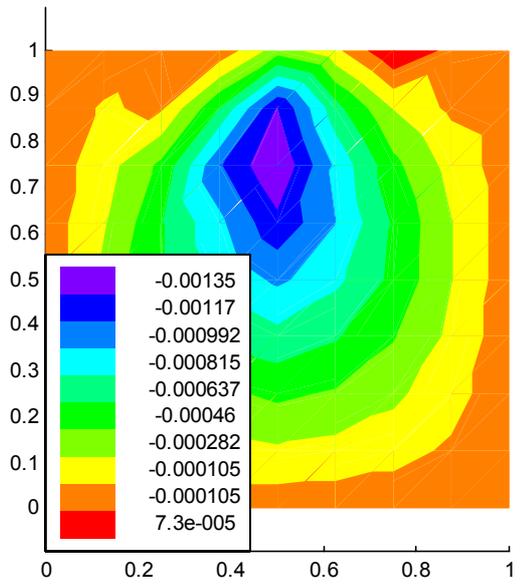


Figure n° 58.4: Tangent Cauchy Stress XY

The distribution of static displacements and stresses shown in Figures n° 58 coincides with the classical FEM solution. The normal stresses showed little errors (Fig. 60 and 61). The highest errors were obtained in the shear under the border of the foundation (Fig. 59). The border of the foundation is a sensitive region and mesh adaptation would be required to obtain more accurate result.

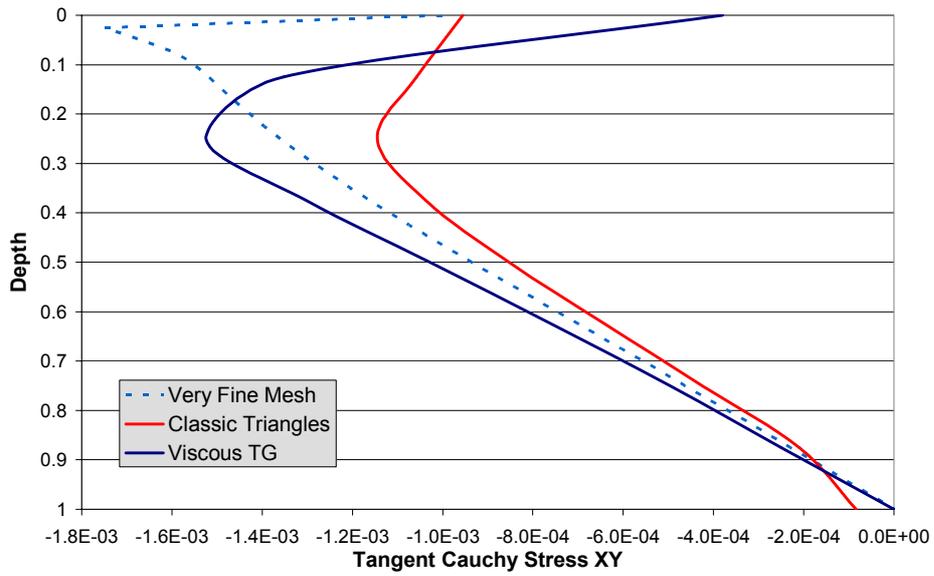


Figure n° 59: Distribution of shear stress under the border of the foundation

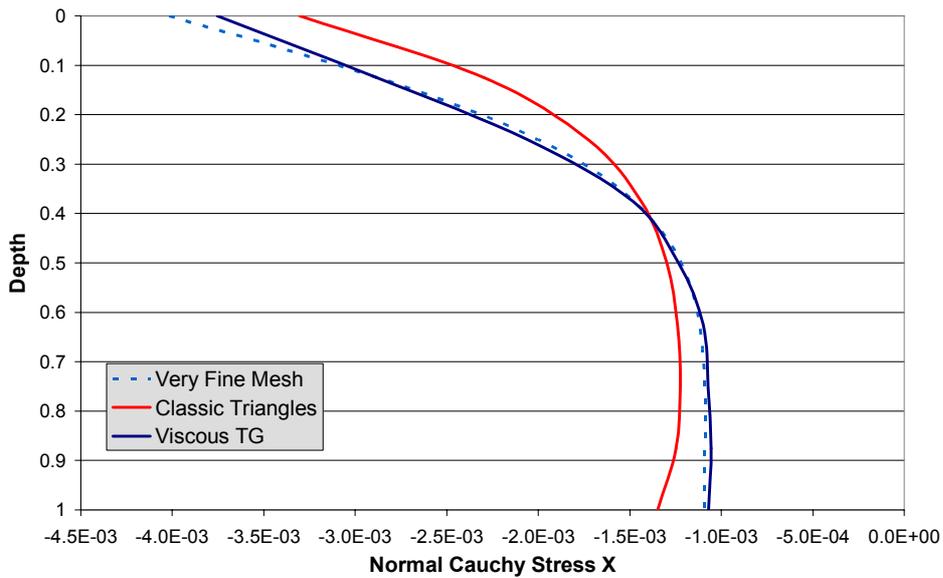


Figure n° 60: Distribution of normal horizontal stress under the edge of symmetry the foundation

The static stresses calculated with the two-step Taylor Galerkin (Fig.59, 60 and 61) are better and more flexible (this is safer) than those obtained with the classic static formulation for the same mesh.

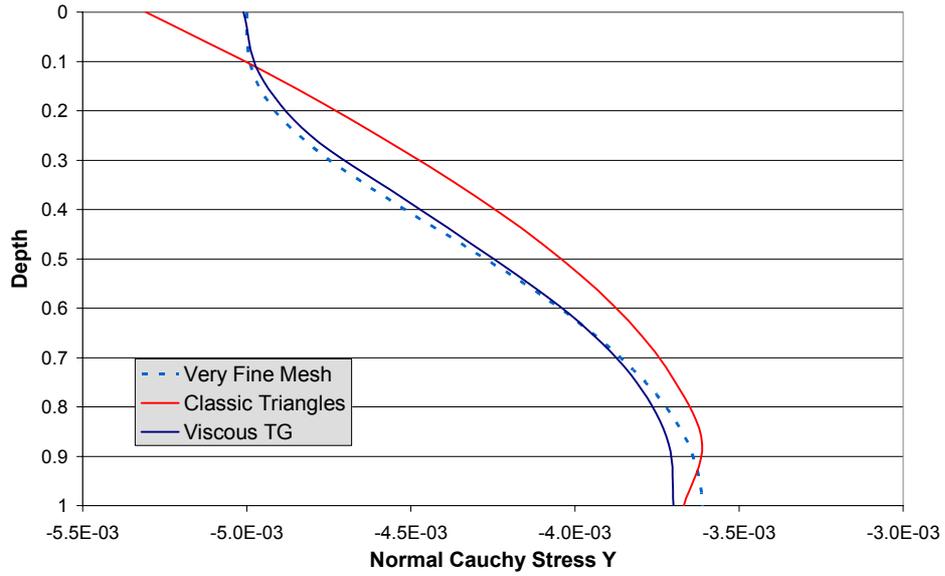


Figure n° 61: Distribution of normal vertical stress under the edge of symmetry the foundation

The dynamic amplification factor is just 8% smaller than the theoretic one.

$$DAF_{MAX} = \frac{4.817e-3}{2.628e-3} = 1.833 \quad (7.9)$$

# Chapter 8

## Conclusions

This work described the basis of a fractional step algorithm (the two-step Taylor–Galerkin) applied to a formulation in terms of the linear momentum and the first Piola–Kirchhoff stresses. It pretends to be an alternative to the classical formulation for special elastodynamic problems that require a specific treatment. The method conserves exactly linear and angular momentum. Moreover, the method exhibits an excellent energy conservation that makes it appropriate for long time integrations.

Good results were obtained in terms of accuracy and computational performance for all the examples included in this work. It is faster and simpler to implement than other one-step schemes. Here we have shown with simple cases that in addition to these advantages, the two-step Taylor–Galerkin was efficient in bending dominated situations, where it performs better than common displacement formulations. Linear triangles can be used without locking effect in nearly incompressible materials.

The use of an adequate amount of artificial viscosity avoids the corruption of the solution due to the high frequencies. Steady state can be achieved successfully with the viscous formulation.

Clearly, further work is required. Future research directions should include the extension of the numerical method to other type of elements. The algorithm must be tested in more complicated problems, where severe mesh distortions are encountered. A mesh adaptation criteria based on an error–estimator would be required to make the scheme more effective. Finally, plastic and viscoplastic materials with large deformations need to be thoroughly studied.

## References

- [1] T. R. Chandrupatla, A. D. Belegundu. Introduction to finite elements in engineering. 3<sup>rd</sup> edition. Prentice Hall, 2002.
- [2] R. Courant, O. Friedrichs, H. Lewy. Über die partiellen Differenzengleichungen der mathematischen, Annual Review of Mathematics, 100, pages 32–74, 1928.
- [3] J. Donea, A Taylor–Galerkin method for convection transport phenomena, Int. J. Numer. Methods Engrg. 20, pages 101–119, 1984.
- [4] J. M. Goicolea. Lecture notes for the PhD in Computational Mechanics. Estructuras sometidas a impacto. E.T.S. Ingenieros de Caminos, Universidad Politécnica Madrid, Spain. [www.mecanica.upm.es](http://www.mecanica.upm.es)
- [5] G. L. Goudreau. Evaluation of discrete methods for the linear dynamic response of elastic and viscoelastic solids. US SESM Report 69–15, University of California Berkley, 1970.
- [6] M. E. Gurtin, An introduction to continuum mechanics. Mathematics in science and engineering, vol. 158. Academic Press, 2003.
- [7] Hao Lin, A. J. Szeri. Shock formation in the presence of entropy gradients. Int. J. Fluid Mech. 431, pages 161–188, 2001.
- [8] O. Hassan. PhD theses: Finite element computations of high speed viscous compressible flows. University College of Swansea, 1990.
- [9] T. R. J. Hughes. The finite element method. Linear static and dynamic finite element analysis. Dover, 2000.
- [10] S. D. K. Lahiri, J. Bonet et al. A variationally consistent fractional–step integration method for incompressible Lagrangian dynamics. Int, J, Numer. Meth. Eng., 63 : pages 1371–1395, 2005.
- [11] S.D. K. Lahiri, Variationally Consistent Methods for Lagrangian Dynamics in Continuum Mechanics, PhD Thesis. Department of Aeronautics and Astronautics, Massachusetts Institute of Technology, 2006.

- [12] P.D. Lax, Hyperbolic systems of conservation laws and the mathematical structure of shock waves, SIAM Regional Conference Series in Applied Mathematics, no. 11, 1972.
- [13] R. J. Leveque. Finite volume methods for hyperbolic problems. Cambridge texts in applied mathematics, 2002.
- [14] F. Lohmeyer, O. Vornberger. Notes on Numerical Fluid Mechanics. Department of Mathematics and Computer Science. University of Osnabruck, Germany, 1994.
- [15] O. C. Zienkiewicz, R. Löhner, K. Morgan, , S. Nakazawa. Finite elements in fluid mechanics – a decade of progress, in Finite Elements in Fluids (eds R.H. Gallagher et al.) Vol. 5, chap. 1, pp. 1-26, Willey, Chichester, 1984.
- [16] R. Löhner, K. Morgan, J. Peraire, O. C. Zienkiewicz. Convection dominated problems, in Numerical Methods for Compressible flows – Finite difference, element and volume techniques, AMD 78, pp. 129–47, ASME, 1987.
- [17] M. Mabssout, M. Pastor. A two–step Taylor–Galerkin algorithm applied to shock wave propagation in soils. Int, J, Numer. Meth Geomech., 27 : pages 685–706, 2003.
- [18] M. Mabssout, M. Pastor. A Taylor–Galerkin algorithm for shock wave and strain localization failure of viscoplastic continua. Int, J, Numer. Meth Eng., 192 : pages 955–971, 2003.
- [19] W. Malalasekera, H.K. Versteeg. Computational fluid Dynamics. Chapter 5: The finite volume method for convection–diffusion problems. Prentice Hall, 1995.
- [20] K. Morgan, O. Hassan, J. Peraire. A time domain unstructured grid approach to simulation of electromagnetic scattering in piecewise homogeneous media. Comp. Mech. Appl. Meth. Eng., 24, pages 101 – 115, 1987.
- [21] K. Morgan, P.J. Brookes, O. Hassan and N.P. Weatherill. Parallel processing for the simulation of problems involving scattering of electromagnetic waves. Comp. Meth. Appl. Mech. Eng., 152, pages 157 – 174, 1998.
- [22] N. M. Newmark. A method of computation for structural dynamics. ASCE Journal of Engineering Mechanics Division, 85: pages 67–94, 1959.

- [23] J. Peraire, J. Bonet, Y. Vidal. A conservation-law formulation for Lagrangian dynamic analysis of hyperelastic materials. April, 2006. Not published yet.
- [24] J.Peraire. A finite method for convection dominated flows. PhD Thesis. University College of Swansea, 1986.
- [25] R.D. Richtmyer, K.W. Morton. Difference methods of Initial-Value Problems. 2<sup>nd</sup> edition, Interscience, New York, 1967.
- [26] [http://en.wikipedia.org/wiki/Lagrangian\\_mechanics](http://en.wikipedia.org/wiki/Lagrangian_mechanics) (Hamilton's principle). The open encyclopedia.
- [27] O.C. Zienkiewicz, R.L. Taylor, The Finite Element Method, vol. 3, Fluid dynamics, fifth ed., Heinemann, 2000.
- [28] O. C. Zienkiewicz and J. Z. Zhu. The superconvergent patch recovery and a-posteriori error estimates, part 1: The recovery technique. Int. Journal. for Numerical Methods in Engrg., 33:1331–1364, 1992.
- [29] J. A. Zukas, editor. High velocity impact dynamics. Chapter 1: Stress waves in solids. John Wiley, 1990.

# Appendix: Design and Implementation of the Code

## I. Algorithm

The design of a program to compute the two-step Taylor – Galerkin algorithm presents the following structure:

1<sup>st</sup>) Read input file.

2<sup>nd</sup>) Initialize the variables and state the Initial Conditions.

3<sup>rd</sup>) Create the consistent mass matrix  $\mathbf{M}_E$  for every element

$$M_{ca} = \int_{\Omega} N_c N_a d\Omega \quad (\text{I})$$

- Assemble the global matrix.
- Renumber the nodes if necessary to reduce the bandwidth (only for non-lumped matrices)
- Store the matrix as a banded matrix (as an array if a lumped matrix is used).

4<sup>th</sup>) **FOR** each **TIME STEP**.

4.1) Prediction step

**FOR** every **ELEMENT** predict the unknowns,

$$\begin{bmatrix} U_i^{n+\frac{1}{2}} \end{bmatrix}_E = \begin{bmatrix} U_i^n \end{bmatrix}_E - \frac{\Delta t}{2} \begin{bmatrix} \frac{\partial \mathfrak{S}_{ij}^n}{\partial X_j} \end{bmatrix}_E = \begin{bmatrix} U_i^n \end{bmatrix}_E - \frac{\Delta t}{2} \begin{bmatrix} \mathfrak{S}_{ij}^n \end{bmatrix}_c \begin{bmatrix} \frac{\partial N_c}{\partial X_j} \end{bmatrix}_E \quad (\text{II})$$

Where

c = 1 : Number of nodes per element

i = 1 : Number of unknowns

j = 1 : Number of dimensions

and calculate the fluxes of the predicted unknowns,

$$\left[ \mathfrak{S}_{ij}^{n+\frac{1}{2}} \right]_E = \mathfrak{S} \left( \left[ U_i^{n+\frac{1}{2}} \right]_E \right) \quad (\text{III})$$

The displacement could be also calculated here provided that  $x_{n+1} = x_n + v_{n+1/2} \Delta t$ .

4.2) Create the right hand side term.

For every local node of the element calculate the internal flux contribution.

$$\left[ f_i^{INT} \right]_c = \Delta t \int_{\Omega} \left[ \mathfrak{S}_{ij}^{n+\frac{1}{2}} \right]_a N_a \frac{\partial N_c}{\partial X_j} d\Omega \quad (\text{IV})$$

Assemble in the global right hand side.

**END of FOR each ELEMENT.**

**FOR each SIDE on the boundary**

- Modify the normal flux depending on the Boundary Type. This is, impose weak Boundary Conditions.
- Add the external contribution.

$$\left[ f_i \right]_c = \left[ f_i^{INT} \right]_c - \Delta t \int_{\Gamma} \left[ \mathfrak{S}_{ij}^{n+\frac{1}{2}} \right]_a N_a N_c d\Gamma \quad (\text{V})$$

a, c = 1 : Number of nodes per element

- Assemble in the global right hand side vector.

**END of FOR each SIDE.**

- Impose strong Boundary Conditions on the global right hand side array.

4.3) Solve the lineal system with a Jacobi iterative method if a non-lumped matrix was used.

$$M\delta U = f \quad (\text{VI})$$

- Create lumped diagonal mass matrix and iterate (about 3 to 6 iterations).

$$\delta U^0 = M_L^{-1} f \quad (\text{VII})$$

$$\delta U^{(r+1)} = \delta U^{(r)} + M_L^{-1}(f - M\delta U^{(r)}) \quad \text{(VIII)}$$

Where  $[M_L]_{ii} = \sum_k M_{ik}$

In the proposed code the lumped matrix is used therefore the system is fully explicit.

4.4) Correct the unknowns and get the fluxes.

$$[U_i^{n+1}]_a = [U_i^n]_a + [\delta U_i]_a \quad \text{(IX)}$$

a = 1 : Number of total nodes of the model

$$[\mathfrak{S}_{ij}^{n+1}]_E = \mathfrak{S}([U_i^{n+1}]_E) \quad \text{(X)}$$

**END of FOR each TIME STEP**

5<sup>th</sup>) Write output file and plot results.

As it was pointed previously, although the element consistent mass matrix offers more accuracy than the lumped one, they have the same asymptotic order and the use of the lumped matrix means less computational cost.

The most important results are often limited to small regions of the calculation domain. It is an efficient idea to use unstructured grids to reduce the number of elements and grid points. As such calculations are very time consuming and inherently parallel the use of multiprocessor systems for this task seems to be a very natural idea [14].

## II. Diagram of procedures and functions

The Figure n° 62 shows the functional structure of the code described in the previous section. Input and output procedures were adapted from the codes of Chandruplata & Belegundu [1].

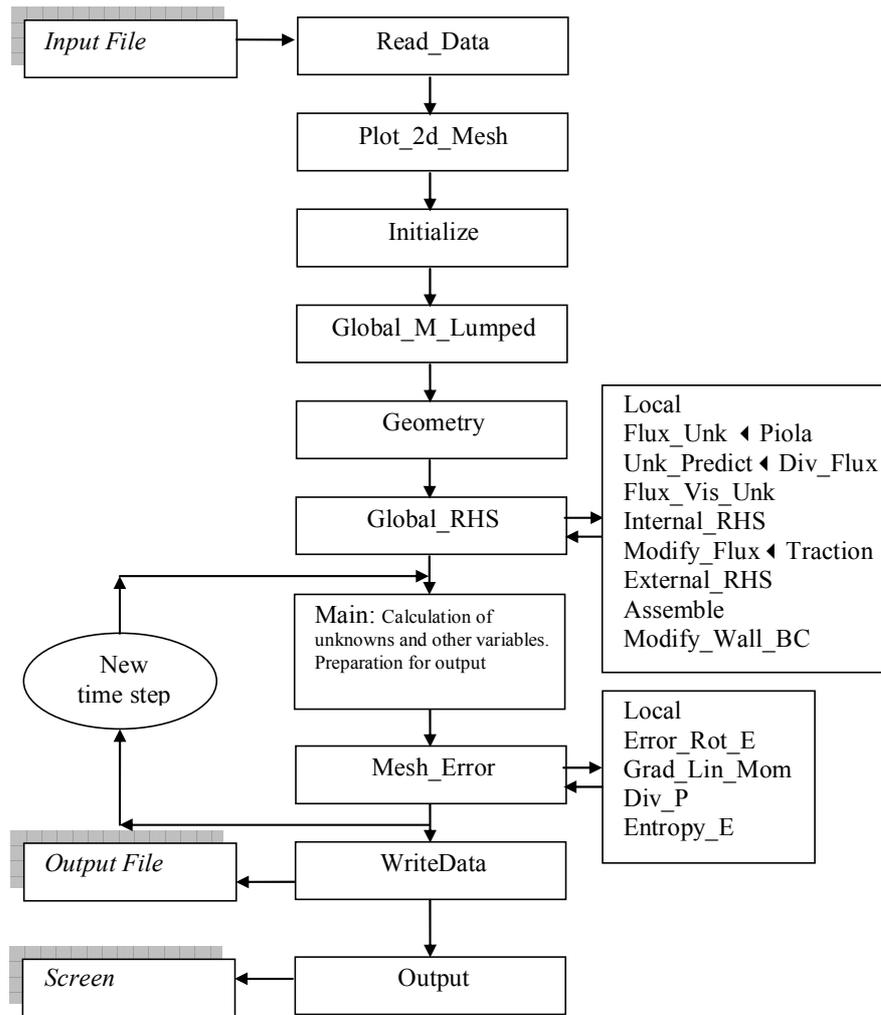


Figure n° 62: Diagram of procedures and functions

### III. Truss 1-D code

#### Main.m

```

clear all;
clc;

%----- Head of the program

disp('*****');
disp('* PROGRAM for LAGRANGIAN DYNAMICS *');
disp('* using the 2-STEP TAYLOR-GALERKIN *');
disp('* and 1-D 2-Nodes Truss elements *');
disp('* X.M.Carreira: xmcarreira@yahoo.com *');
disp('*****');

%----- Reading data of input/output files from keyboard

disp(blanks(1));

FILE1 = input('Input Data File Name ','s');
LINP = fopen(FILE1,'r');
  
```

A TWO-STEP TAYLOR-GALERKIN ALGORITHM APPLIED TO LAGRANGIAN DYNAMICS

```

FILE2 = input('Output Data File Name ','s');
LOUT = fopen(FILE2,'w');

[TITLE,NN,NE,NQ,NBC,NBE,NB,BT,NIC,NV,IC,...
 Mat_Prop,dx,dt,NT,PM,DP,Int,WF]=Read_Data(LINP);

%----- Initializations
[UG]=Initialize(NN,NIC,NV,IC);
Dens=Mat_Prop(3);
Time=0.0;
Counter=0;
Integral_UG=zeros(NQ,1);
Delta_UG=zeros(NQ,1);

%-----Reading Wall BC
[NU]=Wall_BC(NBC,NB,BT);

%----- Creating Mumped Mass Matrix
[M_Lumped] =Global_M_Lumped(NQ,dx);

%----- General loop
for TStep=1:NT

    [RHS]=Global_RHS(NQ,NE,NBE,NB,BT,NU,...
                    Mat_Prop,UG,Time,dx,dt);

    for dof=1:NQ

        Delta_UG(dof,1)=RHS(dof,1)/M_Lumped(dof,1);

    end;

    UG=UG+Delta_UG;

    Integral_UG=Integral_UG+dt*UG;

    Time=TStep*dt;

%----- Plot and output

if or((mod(TStep,PM)==0),TStep==1)
    %----- Display message of evolution of running
    Completed=100*TStep/NT;
    disp(sprintf('Percentage completed = %d', Completed));

    %----- Writing output file
    if (WF==1) % If WF=1 then write in file
        [Done]=Write_Data(LOUT,NN,Time,UG);
    end; % End of writing output file

    Counter=Counter+1;
    T(Counter)=Time;
    if (Int==1) % Integral of the result
        for dof=1:NQ
            Value(Counter,dof)=Integral_UG(dof);
        end;
    elseif (Int==0) % No integral, direct result
        for dof=1:NQ
            Value(Counter,dof)=UG(dof);
        end;
    end; % End of internal if, selection of integral or not

    K=0;
    for Node=1:NN
        dof=2*(Node-1)+1;
        K=K+0.5*M_Lumped(dof)*(UG(dof)^2)/Dens;
    end;
end;

```

```

end;
Kinetic(Counter)=K;

end; % End of external if ... write or plot every PM steps

end; % End of for general loop TStep=1:NT

fclose(LOUT);

disp('Running has finished');

% Plot the stored results

plot(T,Kinetic);

Key=1;
while (Key~=0)
Key=input('1 to plot another node, 0 to exit ');
if (Key==1)
Node=input('enter another node to plot ');
Dof=input('enter dof to be plotted (1 to 2) ');
DP=(Node-1)*2+Dof;
plot(T,Value(:,DP));
end; % End of if
end; % End of while

% End of Main program

```

## Read\_Data.m

```

%----- function Read_Data -----
function [TITLE,NN,NE,NQ,NBC,NBE,NB,BT,NIC,NV,IC,...
Mat_Prop,dx,dt,NT,PM,DP,Int,Wf]=Read_Data(LINP);
%
% SPECIFICATION: This procedure serves to read the input data from the
% input file.
%
% STRUCTURE OF THE INPUT FILE:
%
%-----Beggining of generic input file
% TITLE OF PROBLEM
% Write here your title
% Length NE
% x x
% NBC NIC
% x x
% Node# BT
% x x
% x x
% Node# DOF# Value_IC
% 4 1 0.01
% Node# DOF# Value_IC
% x x x ---- NIC lines, 3 entries
% Young Poisson Dens ---- 1 lines, 3 entries
% x x x ---- 1 lines, 3 entries
% dt NT ---- 1 line, 2 entries%
% x x ---- 1 line, 2 entries%
% Node# DOF# PlotMod Int WFile
% x x x x x ---- 1 line, 4 entries
%-----End of generic input file
%
% DESCRIPTION OF MAIN VARIABLES:
%
% NN: Number of nodes
% NE: Number of elements

```

A TWO-STEP TAYLOR-GALERKIN ALGORITHM APPLIED TO LAGRANGIAN DYNAMICS

```

% NQ:      Total number of degrees of freedom = 6 dof x NN
% NBC:     Number of BC (boundary conditions)
% NB:      Nodes on the boundary
% NB1,NB2: Nodes on the Boundary
% BT:      Boundary type
%          BT=  0 : Hinge          p=0
%          BT=  1 : Free boundary  No traction
%          BT= 10 : Load
% NIC:     Number of IC (this is initial linear momentum in our case)
% NV(I):   Number of the node of the initial condition n° I
% IC(I):   Value of the initial condition n° I
% Young, Poisson, Dens: Material properties
% dt:      Time step
% NT:      Number of time steps
% PM:      Plot results every PM number of steps
% DP:      Degree of freedom to be plotted
% Int:     Integration of plot variable (0 -> No, 1 -> Yes)
% WF:      Write file (0 -> No, 1 -> Yes)

% Reading data from input file

%----- General data -----

DUMMY = fgets(LINP);
TITLE = fgets(LINP);
DUMMY = fgets(LINP);
TMP = str2num(fgets(LINP));
[L, NE] = deal(TMP(1),TMP(2));

NN = NE+1;
NQ = 2*NN;
dx= L/NE;

DUMMY = fgets(LINP);
TMP = str2num(fgets(LINP));
[NBC, NIC]= deal(TMP(1),TMP(2));

%----- Boundaries -----
DUMMY = fgets(LINP);
for I=1:NBC
    TMP = str2num(fgets(LINP));
    [NBE(I),NB(I), BT(I)] = deal(TMP(1), TMP(2), TMP(3));
end

%----- Component Dofs for IC -----
DUMMY = fgets(LINP);
for I=1:NIC
    TMP = str2num(fgets(LINP));
    [Node(I),Dof(I), IC(I)]=deal(TMP(1),TMP(2), TMP(3));
    NV(I)=2*(Node(I)-1)+Dof(I);
end

%----- Material Properties -----
DUMMY = fgets(LINP);
TMP = str2num(fgets(LINP));
[Young,Poisson,Dens,Lambda,Mu] = deal(TMP(1),TMP(2),TMP(3),TMP(4),TMP(5));

Mat_Prop(1)=Young;
Mat_Prop(2)=Poisson;
Mat_Prop(3)=Dens;
Mat_Prop(4)=Lambda;
Mat_Prop(5)=Mu;

%----- Time Parameters -----
DUMMY = fgets(LINP);
TMP = str2num(fgets(LINP));
[dt,NT] = deal(TMP(1),TMP(2));

```

```
%----- Plot Parameters -----
DUMMY = fgets(LINP);
TMP = str2num(fgets(LINP));
[NodePlot,DofPlot,PM,Int,WF] = deal(TMP(1),TMP(2),TMP(3),TMP(4),TMP(5));
DP=(NodePlot-1)*2+DofPlot;

% End of reading input file
fclose(LINP);

end % End of function Read_Data
```

## Initialize.m

```
%----- function Initialize -----

function [UG]=Initialize(NN,NIC,NV,IC);

% SPECIFICATION: This function serves to create a initial UG with
corresponding IC

% NIC:   Number of IC (this is initial linear momentum in our case)
% NN:    Number of nodes
% NV(I): Number of the node of the initial condition n° I
% IC(I): Value of the initial condition n° I

%----- Start with F=I (no deformation),
%                p=0 (no velocities applied)

NQ=2*NN;

UG=zeros(NQ,1);

for Node=1:NN

    Inc=2*(Node-1);

    %   UG(1+Inc,1)=0.0; % p1=0
        UG(2+Inc,1)=1.0; % F11=1
end;

%----- Modify for Initial Conditions -----

for I = 1:NIC
    Eq = NV(I);
    UG(Eq,1) = IC(I);
end;

end; % End of function Initialize
```

## Wall\_BC.m

```
%----- function Wall_BC -----

function [NU]=Wall_BC(NBC,NB,BT);

% SPECIFICATION: This functions serves for putting a penalty where the BC are
located
%
% NBC:   Number of BC (boundary conditions)
% NU(I): Number of the dof of the boundary condition n° I
% BC(I): Value of the boundary condition n° I
% BT:    Boundary type (0 -> Wall, 1 -> FreeX, 10 -> Load)
```

```

% NB:   First node of the side of the element on the boundary
%----- Modify for Boundary Conditions -----
End=0;
for I = 1:NBC
    if (BT(I)==0) % if wall condition then ...
        % Node of the side
        Dof_Start=(NB(I)-1)*2+1;
        Start = End+1;
        End    = Start;
        NU(Start:End) = [ Dof_Start ];

    end;
end;
end % End of function Wall_BC

```

### Global\_M\_Lumped.m

```

%----- function Global_M_Lumped -----
-
function [M_Lumped]=Global_M_Lumped(NQ,dx);
%
% SPECIFICATION: This function serves to create the lumped mass matrix
%
% DESCRIPTION OF MAIN VARIABLES:
%
% NQ:           Total number of degrees of freedom
% NOC:          Number of Components
% dx:           Length of the element
% M_Lumped:     Global lumped mass matrix

%----- Global Stiffness Matrix

M_Lumped = ones(NQ,1);

M_Lumped(1,1) = 0.5;
M_Lumped(2,1) = 0.5;

M_Lumped(NQ-1,1) = 0.5;
M_Lumped(NQ ,1) = 0.5;

M_Lumped = dx*M_Lumped;

end; % End of function Global_M_Lumped

```

### Global\_RHS.m

```

%----- function Global_RHS -----
function [RHS]=Global_RHS(NQ,NE,NBE,NB,BT,NU,...
    Mat_Prop,UG,Time,dx,dt);
%
% SPECIFICATION: This procedure serves to create the system
%
% DESCRIPTION OF MAIN VARIABLES:
%

```

A TWO-STEP TAYLOR-GALERKIN ALGORITHM APPLIED TO LAGRANGIAN DYNAMICS

```

% NN:    Number of nodes
% NE:    Number of elements
% RHS:   Right hand side vector in global system
% RHSE:  Contribution to the right hand side vector of one element
% UE:    Unknowns=transpose[ p1 F1 | p2 F2 ] 4x1
% UG:    Unknowns in global coordinates
% dt:    Step of time
% B:     Bij is the partial of Ni respect to Xj (Voigt matrix)
% BT:    Boundary type of the element
%        BT=    0 : Hinge,                p=0
%        BT=    1 : Free boundary         No traction
%        BT=   10 : Load on the boundary  Traction
% DJ:    Determinant of the Jacobian of the transformation

% Initializing global RHS
RHS = zeros(NQ,1);
dUG = zeros(NQ,1);

% ---- Geometric properties of the element
B(1,1)=-1.0/dx;
B(2,1)= 1.0/dx;

% ---- Material properties
Young      = Mat_Prop(1);
Poisson    = Mat_Prop(2);
Dens       = Mat_Prop(3);
Lambda_Vis = Mat_Prop(4);
Mu_Vis     = Mat_Prop(5);

for N = 1:NE

    Node1=N;
    Node2=N+1;

    % Getting local element unknowns from global unknowns...
    [UE] = Local(UG,Node1,Node2);

    % Calculation of flux at time n
    [FluxE] = Flux_Unk(Young,Poisson,Dens,UE);

    % Prediction of new local element unknowns at time n+1/2 ...
    [dUE] = Unk_Predict(FluxE,B,dt);
    UE=UE+dUE;

    % Prediction of new element flux at time n+1/2
    [FluxE] = Flux_Unk(Young,Poisson,Dens,UE);

    % Viscous flux
    [FluxE_Vis]=Flux_Vis_Unk(Lambda_Vis,Mu_Vis,Dens,UE,B);

    % Total flux without viscosity
    FluxE=FluxE-FluxE_Vis;

    % Getting internal contribution of the element to RHS
    [RHSE]=Internal_RHS(FluxE,B,dx,dt);

    % Adding external flux if necessary
    for side = 1:length(NBE)
        if (N==NBE(side))
            if (Node1==NB(side))
                Normal= -1.0;
                FluxN = FluxE(1:2,1)*Normal;
                FluxN = Modify_Flux(FluxN,BT(side),Time);
                External=-0.5*dt*FluxN;
            % Another possible way to model free boundary is
            %         External(1,1) = dUE(1,1)*dx;
            %         External(2,1) = 0;
        end
    end
end

```

A TWO-STEP TAYLOR-GALERKIN ALGORITHM APPLIED TO LAGRANGIAN DYNAMICS

```

        RHSE(1:2,1)    = RHSE(1:2,1)+External;
    end;
    if (Node2==NB(side))
        Normal= 1.0;
        FluxN = FluxE(3:4,1)*Normal;
        FluxN = Modify_Flux(FluxN,BT(side),Time);
        External=-0.5*dt*FluxN;
%    Another possible way to model free boundary is the very clever way of
Obey Hassan
%        External(1,1) = 0;
%        External(2,1) = dUE(4,1)*dx;
        RHSE(3:4,1)    = RHSE(3:4,1)+External;
    end;
    end;
    end;

% Assembling
[RHS]=Assemble(RHS,Node1,Node2,RHSE);

end; % End of for N = 1:NE (2nd loop)

% Modify RHS for Wall conditions (Strong form)
[RHS]=Modify_Wall_BC(RHS,NU);

end; % End of function Global_RHS

%----- function Local -----
%
function [UE] = Local(UG,Node1,Node2);
%
% SPECIFICATION: This function serves to obtain
%                 the local unknowns from global coordinates
%
% UG: Global vector
% UE: Local vector for the element

    UE=zeros(4,1);

    % Node 1
    Start1 = 2*(Node1-1)+1;
    End1    = Start1+1;
    UE(1:2,1)    = UG(Start1:End1,1);

    % Node 2
    Start2 = 2*(Node2-1)+1;
    End2    = Start2+1;
    UE(3:4,1)   = UG(Start2:End2,1);

end; % End of function Local

%----- function Modify_Flux -----
%
function [Flux] = Modify_Flux(Flux,BT,Time)
%
% SPECIFICATION: This function modifies the Flux of the Element
%                 FluxE depending on the Boundary Type BTE
%
if (BT==0)
    % px=0, py=0
    Flux(2,1)=0.0;
elseif (BT==1)
    % Free condition, traction=P*N=0

```

A TWO-STEP TAYLOR-GALERKIN ALGORITHM APPLIED TO LAGRANGIAN DYNAMICS

```

    Flux(1,1)=0.0;
elseif (BT==10)
    % Free condition, traction
    Flux(1,1)=-1.0*Traction(Time);
end;

end; % End of function Modify_Flux

%----- function Internal_RHS -----
function [RHSE] = Internal_RHS(FluxE,B,dx,dt);
%
% SPECIFICATION: This function serves to calculate the contribution
%                 to RHS of each element considering the internal
%                 flux and external loss through the boundary.
%
Flux1 = FluxE(1:2,1);
Flux2 = FluxE(3:4,1);

Average_Flux = zeros(2,1);
Average_Flux = (Flux1+Flux2)/2;

Int_Rhs=zeros(4,1);
Int_Rhs(1:2,1) = dt*Average_Flux(1:2,1)*B(1,1)*dx; % Node 1
Int_Rhs(3:4,1) = dt*Average_Flux(1:2,1)*B(2,1)*dx; % Node 2

RHSE=Int_Rhs;

end % End of function Global_RHSE

%----- function Assemble -----
function [VG] = Assemble(VG,Node1,Node2,VE)
%
% SPECIFICATION: This function adds the contribution
%                 of elemental VE to the global VG
%
% Node 1
Start1 = 2*(Node1-1)+1;
End1   = Start1+1;
VG(Start1:End1,1) = VG(Start1:End1,1)+VE(1:2,1);

% Node 2
Start2 = 2*(Node2-1)+1;
End2   = Start2+1;
VG(Start2:End2,1) = VG(Start2:End2,1)+VE(3:4,1);

end; % End of function Assemble

%----- function Modify_RHS_Wall_BC -----
function [VG] = Modify_Wall_BC(VG,NU);

% SPECIFICATION: This function serves to modify global VG vector
%                 for the Wall Boundary Conditions in the nodes
%                 given by NU

for I = 1:length(NU)
    N = NU(I);
    VG(N) = 0.0;
end;

```

```
end
end % End of function Modify_RHS_Wall_BC
```

### Unk\_Predict.m

```
%----- function Unk_Predict -----
function [Inc_Unk_Pred]=Unk_Predict(Flux,B,dt);
%
% SPECIFICATION: This function serves to predict the unknowns per element at
the time step n+1/2
%               from the Flux given at time n (at element level), this is,
%               the Taylor prediction of the half-step
%
%
% DESCRIPTION OF MAIN VARIABLES:
%
% Flux:         -1*transpose[ P1 v1 | P2 v2 ] 4x1
% Unknowns:     transpose[ p1 F1 | p2 F2 ] 4x1
% B:            Bij is the partial of Ni respect to Xj (Voigt matrix)
% dt:          Step of time

Inc_Node_Pred = zeros(2,1);
Inc_Unk_Pred  = zeros(4,1);

% PREDICTOR STEP
Inc_Node_Pred = -0.5*dt*Div_Flux(Flux,B);

Inc_Unk_Pred(1:2,1) = Inc_Node_Pred;
Inc_Unk_Pred(3:4,1) = Inc_Node_Pred;

end % End of Unk_Predict
```

```
%-----function Div_Flux -----
%
function [DFlux]=Div_Flux(FluxE,B);
%
% SPECIFICATION: This function serves to calculate the divergence
%               of the Flux of the element
%
% DESCRIPTION OF MAIN VARIABLES:
%
% INPUT
%
% Flux:         The elemental flux is -1*transpose[ P1 v1 | P2 v2 ] 4x1,
%               but it can be any matrix with size 18x2
% dt:          Step of time
% B:            Bij is the partial of Ni respect to Xj (Voigt matrix)
%
% OUTPUT
%
% DivFlux:     Divergence of Flux

DFlux=zeros(2,1);

DFlux=(FluxE(1:2,1)*B(1,1)+FluxE(3:4,1)*B(2,1));

end; % End of Div_Flux function
```

### Flux\_Unk.m

```

%----- function Flux_Unk -----
function [FluxE]=Flux_Unk(Young, Poisson, Dens,UE);
%
% SPECIFICATION: This function serves to calculate the Flux from the
%                 Element Unknowns at the same time step for one element
%
% DESCRIPTION OF MAIN VARIABLES:
%
% INPUT
%
% Young:         Young elastic parameter
% Poisson:       Poisson elastic parameter
% Dens:          Density
% UE:           transpose[ p1 F1 | p2 F2 ] 4x1
%
% OUTPUT
% FluxE:        -1*transpose[ P1 v1 | P2 v2 ] 4x1
%
% Other variables:
% F:           Deformation gradient tensor

    FluxE=zeros(4,1);

    UN=UE(1:2,1); % Node 1
    FluxE(1:2,1)=Nodal_Flux(UN,Young,Poisson,Dens);

    UN=UE(3:4,1); % Node 2
    FluxE(3:4,1)=Nodal_Flux(UN,Young,Poisson,Dens);

end % End of function Flux_Unk;

%----- function Nodal_Flux -----
function [FluxN]=Nodal_Flux(UN,Young,Poisson,Dens);
% SPECIFICATION: This function serves to obtain the nodal Flux
%                 from the vector of Nodal Unknowns UN

    F=UN(2,1);
    P=Piola(Young,Poisson,F);

    FluxN=zeros(2,1);
    FluxN(1,1)=-P;
    FluxN(2,1)=-UN(1,1)/Dens;

end; % End of function Nodal_Flux

```

## Flux\_Vis\_Unk.m

```

%----- function Flux_Vis__Unk -----
function [FluxE_Vis]=Flux_Vis_Unk(Lambda_Vis,Mu_Vis,Dens,UE,B);
%
% SPECIFICATION: This function serves to calculate the viscous Flux from the
%                 Element Unknowns at the same time step for one element
%
% DESCRIPTION OF MAIN VARIABLES:
%
% INPUT
%
% Dens:         Density
% Mu_Vis, Lambda_Vis: Viscous parameters
% UE:          transpose[ p1 F1 | p2 F2 ] 4x1
% B:           matrix with the derivative of the shape functions
%
% OUTPUT

```

A TWO-STEP TAYLOR-GALERKIN ALGORITHM APPLIED TO LAGRANGIAN DYNAMICS

```

% FluxE_Vis:   -1*transpose[ P1vis 0 | P2vis 0 ] 4x1
%
% Other variables:
% F:           Deformation gradient tensor

% Linear momentum

p1=UE(1,1);
p2=UE(3,1);

% Gradient of linear momentum

[Grad_p]=Grad_Lin_Mom(p1,p2,B);

FluxE_Vis=zeros(4,1);

for Node=1:2
    Inc=2*(Node-1);
    U_Nodal=UE((1+Inc):(2+Inc));
    Flux_Nodal=zeros(2,1);
    Flux_Nodal=Nodal_Flux(U_Nodal,Lambda_Vis,Mu_Vis,Dens,Grad_p);
    FluxE_Vis((1+Inc):(2+Inc),1)=Flux_Nodal;
end; % End of for

end % End of function Flux_Unk;

%----- function Nodal_Flux -----

function [FluxN]=Nodal_Flux(UN,Lambda_Vis,Mu_Vis,Dens,Grad_p);
% SPECIFICATION: This function serves to obtain the nodal Flux
%                from the vector of Nodal Unknowns UN

    F=UN(2);
    P_Vis=Piola_Vis(Lambda_Vis,Mu_Vis,Dens,F,Grad_p);

    FluxN=zeros(2,1);
    FluxN(1,1)=+1.0*P_Vis;

end; % End of procedure Put_FluxE_Vis

%----- function Grad_Lin_Mom -----

function [Grad_p]=Grad_Lin_Mom(p1,p2,B);
%
% SPECIFICATION: This function serves to calculate the gradient
%                of the vector of Unknowns
%
% DESCRIPTION OF MAIN VARIABLES:
%
% INPUT
%
% pi=   linear momentum of node i
%       p=[ px ]
% dt:   Step of time
% B:    Bij is the partial of Ni respect to X (Voigt matrix)
%
% OUTPUT
%
% Grad_p: Gradient of Unknowns
%         | px, x |

Grad_p=p1*B(1,1)+p2*B(2,1);

end; % End of Grad_Lin_Mom function

```

## Piola.m

```

%----- function Piola -----
function [P]=Piola(Young,Poisson,F);
%
% SPECIFICATION: This function serves to calculate the first Piola
%                 stress tensor in 1D problems
%
% DESCRIPTION OF MAIN VARIABLES:
%
% INPUT:
%
% F :           Deformation gradient tensor
% Young:        Young elastic parameter
% Poisson:      Poisson elastic parameter
%
% OUTPUT:
%
% P:           First Piola-Kirchhoff stress tensor
%
% Model=1: Large deformation
% Model=2: Small deformation

Model=2;

% Calculation of main elastic parameters
Mu=0.5*Young/(1+Poisson);
Kappa=Young/(3-6*Poisson);

% Calculation of Piola Kirchhoff stress tensor

% Error P=Mu*(F^(-2/3)-(1/3)*(2*F^(-5/3)+F^(1/3)))+Kappa*(F-1);

if (Model==1)

P=(2/3)*Mu*(F^(-1/3)-F^(-5/3))+Kappa*(F-1);

elseif (Model==2)

P = ((4/3)*Mu+Kappa)*(F-1);

end;

% End of Piola function

```

## Piola\_Vis.m

```

%----- function Piola_Vis -----
function [P_Vis]=Piola_Vis(Lambda_Vis,Mu_Vis,Dens,F,Grad_p);
%
% SPECIFICATION: This function serves to calculate the viscous Piola
%                 stress tensor in 1D problems
%
% DESCRIPTION OF MAIN VARIABLES:
%
% INPUT:
% Lambda_Vis, Mu_Vis: Viscous parameter
%
% Grad_p:           Gradient of linear momentum
%
% OUTPUT:
%
% P_V:             Viscous Piola-Kirchhoff stress tensor

```

```
%
% Viscous constitutive model (for numerical purposes only)
Constant = ((Lambda_Vis+2*Mu_Vis)/Dens);
P_Vis = Constant*Grad_p*F^(-3);
end; % End of Piola_Vis function
```

## Traction.m

```
%----- function Traction -----
function [Stress]=Traction(Time);
%
% SPECIFICATION: This function serves to specify the applied
%                 Stress on the boundary sides.
%
Stress_Max = 0.1; % Positive means in the direction on the Normal vector
T0          = 0.5;

% Linear function
if (Time==0)
    Stress=0;
elseif and((Time < T0),(Time > 0))
    Stress=Stress_Max*(T0-(T0-Time))/T0;
else
    Stress=Stress_Max;
end;

end; % End of Load function
```

## Output.m

```
%----- function Write_Data -----
function [Done]=Write_Data(LOUT,NN,Time,UG);

for I=1:NN

    LinMomX(I)=UG(2*(I-1)+1);
    EV(I)     =UG(2*(I-1)+2)-1;

end

fprintf(LOUT,' %15.4E\n',Time);
fprintf(LOUT,' Node#    LinMomX          VolDef\n');
for I = 1:NN
    fprintf(LOUT,' %4d %15.3E %15.3E\n',I,LinMomX(I),EV(I));
end;
Done=1;
end;
```

## IV. Triang 2-D code

### Main.m

```

clear all;
clc;

%----- Head of the program

disp('*****');
disp('* PROGRAM for LAGRANGIAN DYNAMICS *');
disp('* using the 2-STEP TAYLOR-GALERKIN *');
disp('* and PLANE STRAIN 3-Nodes TRIANGLES *');
disp('* X.M.Carreira: xmcarreira@yahoo.com *');
disp('*****');

%----- Reading data of input/output files from keyboard

disp(blanks(1));

FILE1 = input('Input Data File Name ','s');
LINP = fopen(FILE1,'r');

FILE2 = input('Output Data File Name ','s');
LOUT = fopen(FILE2,'w');

[TITLE,NN,NE,NQ,X,NOC,NBC,NBE,NB1,NB2,BT,NIC,NV,IC,...
 Mat_Prop,dt,NT,PM,WF]=Read_Data(LINP);

%----- Material properties
Young = Mat_Prop(1);
Poisson = Mat_Prop(2);
Kappa=Young/(3-6*Poisson);
G=0.5*Young/(1.0+Poisson); % Shear modulus (also called Mu)
Dens = Mat_Prop(3);

%----- Plot mesh
[Done] = Plot_2d_Mesh(NN,NE,3,X,NOC);

%----- Initializations
[UG]=Initialize(NN,NIC,NV,IC);
Time=0.0;
Counter=0;
E=0;
Velocity=zeros(NN,2);
Displacement=zeros(NN,2);
Delta_UG=zeros(NQ,1);

%-----Creating mass matrix
[M_Lumped]=Global_M_Lumped(NQ,X,NOC,NE);

%-----Reading Wall BC
[NU]=Wall_BC(NBC,NB1,NB2,BT);

%-----Geometric properties
[DJ,B,Normal,Length,N1,N2] = Geometry(NE,X,NOC,NBC,NBE,NB1,NB2);

%----- General loop
for TStep = 1:NT

    [RHS,EP]=Global_RHS(DJ,B,Normal,Length,N1,N2,NE,NQ,NOC,NBE,BT,NU,...
        Mat_Prop,UG,Time,dt);

    for dof=1:NQ
        Delta_UG(dof,1)=RHS(dof,1)/M_Lumped(dof,1);
    end;
end;

```

```

UG=UG+Delta_UG;

% Velocities
for Node = 1:NN
    Start = 6*(Node-1);
    Velocity(Node,1)=UG(Start+1)/Dens;
    Velocity(Node,2)=UG(Start+2)/Dens;
end;

% Displacements
Displacement(1:NN,1:2)=Displacement(1:NN,1:2)+dt*Velocity(1:NN,1:2);
XD(1:NN,1:2)=X(1:NN,1:2)+Displacement(1:NN,1:2); % Deformed geometry

Time=TStep*dt;

%---- Preparing for output
if or(mod(TStep,PM)==0),TStep==1)

    % Initialization of some variables

        K=0;           % Kinetic energy
        Pot=0;        % Potential energy
        Lin=zeros(2,1); % Linear momentum
        Ang=0;        % Angular momentum

    % Creating output results
    Counter=Counter+1;
    T(Counter)=Time;
    Displ(1:NN,1:2,Counter) = Displacement;
    Vel(1:NN,1:2,Counter)   = Velocity;

    for Node=1:NN

        Start = 6*(Node-1);

        K=K+0.5*(M_Lumped(Start+1)*UG(Start+1)^2+...
            M_Lumped(Start+2)*UG(Start+2)^2)/Dens;

        Lin(1,1)=Lin(1,1)+M_Lumped(Start+1)*UG(Start+1);
        Lin(2,1)=Lin(2,1)+M_Lumped(Start+2)*UG(Start+2);

        Ang=Ang+(M_Lumped(Start+1)*UG(Start+1)*XD(Node,2)-...
            M_Lumped(Start+2)*UG(Start+2)*XD(Node,1));

        F(1,1) = UG(Start+3);
        F(1,2) = UG(Start+4);
        F(2,1) = UG(Start+5);
        F(2,2) = UG(Start+6);

        J=det(F);

        Pressure(Node,1,Counter)=Kappa*(J-1);

        Almansi_Strain=0.5*(ones(2,2)-inv(F*transpose(F)));

        Strain(Node,1,Counter) = Almansi_Strain(1,1);
        Strain(Node,2,Counter) = Almansi_Strain(2,2);
        Strain(Node,3,Counter) = Almansi_Strain(1,2);

        Cauchy_Stress= Piola(Young,Poisson,F)*transpose(F)/det(F);

        Stress(Node,1,Counter) = Cauchy_Stress(1,1);
        Stress(Node,2,Counter) = Cauchy_Stress(2,2);
        Stress(Node,3,Counter) = Cauchy_Stress(1,2);

        C=transpose(F)*F; % Cauchy-Green deformation tensor

        Nodal_Area=M_Lumped(Start+1);
    end;
end;

```

A TWO-STEP TAYLOR-GALERKIN ALGORITHM APPLIED TO LAGRANGIAN DYNAMICS

```

    % Nodal Potential energy
    Pot_Vol = 0.5*Nodal_Area*Kappa*(J-1)^2;
    Pot_Dev = 0.5*Nodal_Area*G*(J^(-2/3)*(1+trace(C))-3);
    Pot=Pot+Pot_Vol+Pot_Dev;

end; % End of for Node=1:NN

Error_Press(1:NN,1,Counter)=EP(1:NN);

%----- Control of mesh errors: Rotational of F, Entropy and Energy
[ErrorRF(Counter),DE] = Mesh_Error(NOC,NE,UG,B,Young,Poisson);
E=E+DE*dt;
Entropy(Counter)=E;
Kinetic(Counter)=K;
Potential(Counter)=Pot;
Linear_Mom(Counter)=sqrt(Lin(1,1)^2+Lin(2,1)^2);
Angular_Mom(Counter)=Ang;

%----- Display message of evolution of running
Completed=100*TStep/NT;
disp(sprintf('Percentage completed = %d', Completed));

end; % End of external if ... write or plot every PM steps

end; % End of for general loop TStep = 1:NT

% END OF RUNNING.....

if (WF==1) % If WF=1 then write in file
    [Done] = Write_Data(LOUT,NN,T,Displ,Vel,...
        Strain,Stress,Pressure,Error_Press);
end; % End of writing output file

clc;

% Plot the average error vs time: evolution of average error
plot(T,ErrorRF);
xlabel('Time');
ylabel('Error RotF');

pause=input('press any key to plot another graphic');

% Plot the average error vs time: evolution of average error
plot(T,Entropy);
xlabel('Time');
ylabel('Entropy');

pause=input('press any key to plot another graphic');

Energy=Potential+Kinetic;

% Plot energy
plot(T,Potential,':',T,Kinetic,'--',T,Energy);
legend('Potential','Kinetic','Total Energy');
xlabel('Time');
ylabel('Energy');

pause=input('press any key to plot another graphic');

% Plot linear momentum
plot(T,Linear_Mom);
xlabel('Time');
ylabel('Linear momentum');

pause=input('press any key to plot another graphic');

% Plot angular momentum
plot(T,Angular_Mom);

```

```
xlabel('Time');
ylabel('Angular momentum');

% Postprocessing data
[Done] = Output(NN,NE,X,NOC,T,Displ,Vel,Strain,...
               Stress,Pressure,Error_Press);

% End of Main program
```

## Read\_Data.m

```
%----- function Read_Data -----

function [TITLE,NN,NE,NQ,X,NOC,NBC,NBE,NB1,NB2,BT,NIC,NV,IC,...
        Mat_Prop,dt,NT,PM,Wf]=Read_Data(LINP);

%
% SPECIFICATION: This procedure serves to read the input data from the
%                 input file.
%
% STRUCTURE OF THE INPUT FILE:
%
%-----Beggining of generic input file
% TITLE OF PROBLEM
% Write here the description of the problem
% NN NE
% x x ----- 1 line, 2 entries
% NBC NIC
% x x ----- 1 line, 2 entries
% Node# X Y
% x x x ----- NN lines, 3 entries
% Elem# N1 N2 N3
% x x x x ----- NE lines, 4 entries
% Elem# Nodel Node2 BT
% x x x x ----- NBC lines, 4 entries
% Node# DOF# Value_IC
% x x x ----- NIC lines, 3 entries
% Young Poisson Dens Lambda Mu
% x x x x x ----- 1 lines, 5 entries
% dt NT
% x x ----- 1 line, 2 entries%
% PlotMod WFile
% x x ----- 1 line, 2 entries
%-----End of generic input file
%
% DESCRIPTION OF MAIN VARIABLES:
%
% NN:      Number of nodes
% NE:      Number of elements
% NQ:      Total number of degrees of freedom = 6 dof x NN
% NBC:     Number of BC (boundary conditions)
% NB:      Nodes on the boundary
% BT:      Boundary type
% BT=      0 : Clamped wall px=0, py=0
% BT=     -1 : Hinges parallel to X py=0
% BT=     -2 : Hinges parallel to Y px=0
% BT=    -10 : Movement parallel to X py=0, Pn=0
% BT=    -20 : Movement parallel to Y px=0, Pn=0
% BT=      1 : Free boundary Pn=0
% BT=     10 : Load Traction=Pn
% NIC:     Number of IC
% NV(I):   Number of the node of the initial condition n° I
% IC(I):   Value of the initial condition n° I
% NOC:     Number of Components
```

## A TWO-STEP TAYLOR-GALERKIN ALGORITHM APPLIED TO LAGRANGIAN DYNAMICS

```

% X:          Coordinates
% Young, Poisson, Dens: Material properties
% dt:         Time step
% NT:         Number of time steps
% PM:         Plot results every PM number of steps
% DP:         Degree of freedom to be plotted
% Int:        Integration of plot variable (0 -> No, 1 -> Yes)
% WF:         Write file (0 -> No, 1 -> Yes)
% PT:         Plot Type (1 -> Lines, 2 -> Filled)

% Reading data from input file

DUMMY = fgets(LINP);
TITLE = fgets(LINP);
DUMMY = fgets(LINP);
TMP = str2num(fgets(LINP));
[NN, NE] = deal(TMP(1),TMP(2));

NQ = 6 * NN;

DUMMY = fgets(LINP);
TMP = str2num(fgets(LINP));
[NBC, NIC]= deal(TMP(1),TMP(2));

%----- Coordinates -----
DUMMY = fgets(LINP);
for I=1:NN
    TMP = str2num(fgets(LINP));
    [N, X(N,1:2)]=deal(TMP(1),TMP(2:3));
end

%----- Connectivity -----
DUMMY = fgets(LINP);
for I=1:NE
    TMP = str2num(fgets(LINP));
    [N,NOC(N,1:3)]=deal(TMP(1),TMP(2:4));
end

%----- Boundaries -----
DUMMY = fgets(LINP);
for I=1:NBC
    TMP = str2num(fgets(LINP));
    [NBE(I), NB1(I), NB2(I), BT(I)] = deal(TMP(1),TMP(2),TMP(3),TMP(4));
end

%----- Component Dofs for IC -----
DUMMY = fgets(LINP);
for I=1:NIC
    TMP = str2num(fgets(LINP));
    [Node(I),Dof(I),IC(I)]=deal(TMP(1),TMP(2),TMP(3));
    NV(I)=6*(Node(I)-1)+Dof(I);
end

%----- Material Properties -----
DUMMY = fgets(LINP);
TMP = str2num(fgets(LINP));
[Young,Poisson,Dens,Lambda,Mu] = ...
    deal(TMP(1),TMP(2),TMP(3),TMP(4),TMP(5));

Mat_Prop(1)=Young;
Mat_Prop(2)=Poisson;
Mat_Prop(3)=Dens;
Mat_Prop(4)=Lambda;
Mat_Prop(5)=Mu;

%----- Time Parameters -----
DUMMY = fgets(LINP);
TMP = str2num(fgets(LINP));

```

```
[dt,NT] = deal(TMP(1),TMP(2));

%----- Plot Parameters -----
DUMMY = fgets(LINP);
TMP = str2num(fgets(LINP));
[PM,WF] = deal(TMP(1),TMP(2));

% End of reading input file
fclose(LINP);

% End of function Read_Data
```

## Plot\_2d\_Mesh.m

```
function [Done] = Plot_2d_Mesh(NN,NE,NEN,X,NOC)

% SPECIFICATION: This function serves to plot the mesh

% This function was based on the original
% program of Chandrupatla and Belegundu

% Calculate offset, epsilon, for printing node numbers
LX = abs(max(X(:,1))-min(X(:,1)));
LY = abs(max(X(:,2))-min(X(:,2)));
LMAX = max(LX,LY);
epsilon = 0.01*LMAX;

for N=1 : NE
    for j=1 : NEN
        xe(j)=X(NOC(N,j),1);
        ye(j)=X(NOC(N,j),2);
    end;
    xe(j+1)=xe(1);
    ye(j+1)=ye(1);
    line(xe,ye)
end;

disp(' Type 1 for on/off node labels,?n');
disp(' Type 2 for on/off element labels,?n');
disp(' Type 3 for on/off node & element labels,?n');
disp(' Type 0 to CONTINUE,?n');

status = 1; k=0; kold = -1;
while status ~= 0
    user_input=input(' Type <1,2,3 OR 0 > : ','s');
    if k == 1 %on-->off
        if kold ~= -1
            ktemp=str2num(user_input);
            if ktemp ~= kold
                user_input = num2str(kold);
            end;
        end;
        k=0;
    else % off-->on
        k=1;
    end;
    figure(1);
    status = str2num(user_input);
    switch status
    case 1
        if k==1
            for i=1:NN
                xc = X(i,1)-epsilon;yc=X(i,2)-epsilon;
                hhl(i,:) = text(xc,yc,num2str(i),'FontSize',8);
            end;
        end;
    end;
end;
```

```

        end;
        kold = str2num(user_input);
    else
        delete(hh1);
    end;
case 2
    if k==1
        for i=1:NE
            xc=0;yc=0;
            for ii=1 : NEN
                xc=xc + X(NOC(i,ii),1);
                yc=yc + X(NOC(i,ii),2);
            end;
            xc=xc/NEN;yc=yc/NEN;
            hh2(i,:) = text(xc,yc,num2str(i),'FontSize',8);
        end;
        kold = str2num(user_input);
    else
        delete (hh2);
    end;
case 3
    if k==1
        for i=1:NN
            xc = X(i,1);yc=X(i,2);
            hh3(i,:) = text(xc,yc,num2str(i),'FontSize',8);
        end;
        for i=1:NE
            xc=0;yc=0;
            for ii=1 : NEN
                xc=xc + X(NOC(i,ii),1);
                yc=yc + X(NOC(i,ii),2);
            end;
            xc=xc/NEN;yc=yc/NEN;
            hh4(i,:) = text(xc,yc,num2str(i),'FontSize',8);
        end;
        kold = str2num(user_input);
    else
        delete(hh3);delete(hh4);
    end;
end;
axis equal;
end;

Done=1;

% End of function Plot_2d_Mesh

```

## Contour1.m

```

function [Done] = Contour1(NN,NE,NEN,X,NOC,FF)

global hh

% Nodal variable is stored in FF(1:NN)

    FMAX = max(FF); FMIN = min(FF);

    NCL = 10; %no. of colours
    STP = (FMAX - FMIN) / NCL;
    % Red-Orange-Green-Blue-Magenta colors
    % for 10 contour lines [MM(1,:) to MM(10,:)]
    for i=1:10
        a=((10-i)/12); b=1; c=1;
        H=[a,b,c];MM(i,:)=hsv2rgb(H);
    end;

% Find boundary lines

```

A TWO-STEP TAYLOR-GALERKIN ALGORITHM APPLIED TO LAGRANGIAN DYNAMICS

```

% Edges defined by nodes in NOC to nodes in NCON
for IE = 1 : NE
    for I = 1 : NEN
        I1 = I + 1;
        if I1 > NEN
            I1 = 1;
        end;
        NCON(IE, I) = NOC(IE, I1);
    end;
end;
for IE = 1 : NE
    for I = 1 : NEN
        I1 = NCON(IE, I); I2 = NOC(IE, I);
        INDX = 0;
        for JE = IE + 1 : NE
            for J = 1 : NEN
                flow=2;
                if (NCON(JE, J) == 0)
                    flow=1;
                end;
                if ((I1 ~= NCON(JE, J)) & (I1 ~= NOC(JE, J)))
                    flow=1;
                end;
                if ((I2 ~= NCON(JE, J)) & (I2 ~= NOC(JE, J)))
                    flow=1;
                end;
                if (flow==2)
                    NCON(JE, J) = 0; INDX = INDX + 1;
                end
            end;
        end;
        if INDX > 0
            NCON(IE, I) = 0;
        end;
    end;
end;
axis off

%===== Draw Boundary =====
for IE = 1 : NE
    for I = 1 : NEN
        if NCON(IE, I) > 0
            I1 = NCON(IE, I); I2 = NOC(IE, I);
            xe(1)=X(I1,1);xe(2)=X(I2,1);
            ye(1)=X(I1,2);ye(2)=X(I2,2);
            line (xe,ye, 'LineWidth', 2, 'color', 'b')
        end;
    end;
end

%===== Contour Plotting =====
for IE = 1 : NE
    if (NEN == 3)
        for IEN = 1 : NEN
            IEE = NOC(IE, IEN);
            U(IEN) = FF(IEE);
            XX(IEN) = X(IEE, 1);
            YY(IEN) = X(IEE, 2);
        end;
        LPLOT(U,XX,YY,FMIN,STP,NCL, MM);
    elseif (NEN == 4)
        XB = 0; YB = 0; UB = 0;
        for IT = 1 : NEN
            NIT = NOC(IE, IT);
            XB = XB + .25 * X(NIT, 1);
            YB = YB + .25 * X(NIT, 2);
            UB = UB + .25 * FF(NIT);
        end;
    end;
end;

```

```

    for IT = 1 : NEN
        IT1 = IT + 1;
        if (IT1 > 4)
            IT1 = 1;
        end;
        XX(1) = XB; YY(1) = YB; U(1) = UB;
        NIE = NOC(IE, IT);
        XX(2) = X(NIE, 1); YY(2) = X(NIE, 2); U(2) = FF(NIE);
        NIE = NOC(IE, IT1);
        XX(3) = X(NIE, 1); YY(3) = X(NIE, 2); U(3) = FF(NIE);
        LPLOT(U,XX,YY,FMIN,STP,NCL, MM);
    end;
else
    disp( 'NUMBER OF ELEMENT NODES > 4 IS NOT SUPPORTED')
    stop
end;
end;

% Draw legend
for i=1 : NCL
    cc(i,1)=FMIN+STP*i;
end;
ccl=num2str(cc,3);
legend(hh,ccl,-1);
axis equal;
Done=1;

% End of function contour1

%----- function LPLOT
function [] = LPLOT(U,XX,YY,FMIN,STP,NCL, MM)
global hh
%THREE POINTS IN ASCENDING ORDER
    for I = 1 : 2
        C = U(I); II = I;
        for J = I + 1 : 3
            if C > U(J)
                C = U(J); II = J;
            end;
        end;
        U(II) = U(I); U(I) = C;
        C1 = XX(II); XX(II) = XX(I); XX(I) = C1;
        C1 = YY(II); YY(II) = YY(I); YY(I) = C1;
    end;

    SU = (U(1) - FMIN) / STP;
    II = fix(SU+1.e-10)+1;
    if II>NCL
        II=NCL;
    end;

    UT = FMIN + II * STP;
    while UT <= U(3)
        X1 = ((U(3)-UT)*XX(1) + (UT-U(1))* XX(3))/(U(3) - U(1));
        Y1 = ((U(3)-UT)*YY(1) + (UT-U(1))* YY(3))/(U(3) - U(1));
        L = 1;
        if UT > U(2)
            L = 3;
        end;
        X2 = ((U(L)-UT)* XX(2) + (UT-U(2))*XX(L))/(U(L) - U(2));
        Y2 = ((U(L)-UT)* YY(2) + (UT-U(2))*YY(L))/(U(L) - U(2));
        xe(1)=X1;xe(2)=X2;ye(1)=Y1;ye(2)=Y2;
        hh(II,1)=line(xe,ye,'LineWidth',2,'color',MM(II,:));
        UT = UT + STP;
        II = II + 1;
    end;

% End of function LPLOT

```

## Contour2.m

```
function [Done] = Contour2(NN,NE,NEN,X,NOC,FF)

global hh cc

% Nodal variable is stored in FF(1:NN)

FMAX = max(FF); FMIN = min(FF);

NCL = 10; %no. of colours
STP = (FMAX - FMIN) / NCL;

% Red-Orange-Green-Blue-Magenta colors
% for 10 contour lines [MM(1,:) to MM(10,:)]
for i=1:10
    a=((10-i)/12); b=1; c=1;
    H=[a,b,c];
    MM(i,:)=hsv2rgb(H);
end;

%===== Draw Contour =====

for N=1 : NE
    hold on;
    if (NEN == 3)
        for IEN = 1 : NEN
            IEE = NOC(N, IEN);
            UU(IEN) = FF(IEE);
            XX(IEN) = X(IEE, 1);
            YY(IEN) = X(IEE, 2);
        end;
        LPLOT(UU,XX,YY,FMIN,STP,NCL, MM);
    elseif (NEN == 4)
        XB = 0; YB = 0; UB = 0;
        for IT = 1 : NEN
            NIT = NOC(N, IT);
            XB = XB + .25 * X(NIT, 1);
            YB = YB + .25 * X(NIT, 2);
            UB = UB + .25 * FF(NIT);
        end;
        for IT = 1 : NEN
            IT1 = IT + 1;
            if (IT1 > 4)
                IT1 = 1;
            end;
            XX(1) = XB; YY(1) = YB; UU(1) = UB;
            NIE = NOC(N, IT);
            XX(2) = X(NIE, 1); YY(2) = X(NIE, 2); UU(2) = FF(NIE);
            NIE = NOC(N, IT1);
            XX(3) = X(NIE, 1); YY(3) = X(NIE, 2); UU(3) = FF(NIE);
            LPLOT(UU,XX,YY,FMIN,STP,NCL, MM);
        end;
    else
        disp('NUMBER OF ELEMENT NODES > 4 IS NOT SUPPORTED')
        stop
    end;
end;

% Draw legend;
dd = nonzeros(cc);
cc1=num2str(dd,3);
legend(hh,cc1,-1);
```

A TWO-STEP TAYLOR-GALERKIN ALGORITHM APPLIED TO LAGRANGIAN DYNAMICS

```

axis equal;
Done=1;

% End of function contour2

%----- function LPLOT

function [] = LPLOT(UU,XX,YY,FMIN,STP,NCL, MM)
global hh cc
[U, INDEX] = sort(UU);
SU = (U(2) - FMIN) / STP;
II = fix(SU+1.e-10)+1;
if II>NCL
    II=NCL;
end;
UT = FMIN + II*STP;IT=II;U2= U(2);
ULEVEL2 = UT;ILEVEL2=II;

%lower triangle
if U(3)~=U(1)
    while UT > U(1)
        psi = (U2-U(1))/(U(3)-U(1));
        xpsi=XX(INDEX(1))+psi*(XX(INDEX(3))-XX(INDEX(1)));
        ypsi=YY(INDEX(1))+psi*(YY(INDEX(3))-YY(INDEX(1)));
        if U(2)~=U(1)
            psi2=(U2-U(1))/(U(2)-U(1));
        else
            psi2=1;
        end;
        xc = XX(INDEX(1))+psi2*(XX(INDEX(2))-XX(INDEX(1)));
        yc=YY(INDEX(1))+psi2*(YY(INDEX(2))-YY(INDEX(1)));
        xe(1)=XX(INDEX(1));xe(2)=xc;xe(3)=xpsi;
        ye(1)=YY(INDEX(1));ye(2)=yc;ye(3)=ypsi;
        hh(IT,:)=...
        fill(xe,ye,MM(IT,:), 'LineWidth', .0000001, 'LineStyle', 'none');
        cc(IT,1)=UT;
        UT = UT-STP; U2=UT; IT = IT-1;
        if IT<1
            IT=1;
        end;
    end;
else
    xe(1)=XX(INDEX(1));xe(2)=XX(INDEX(2));xe(3)=XX(INDEX(3));
    ye(1)=YY(INDEX(1));ye(2)=YY(INDEX(2));ye(3)=YY(INDEX(3));
    hh(IT,:)=...
    fill(xe,ye,MM(IT,:), 'LineWidth', .0000001, 'LineStyle', 'none');
end;

%upper triangle
UT=ULEVEL2-STP; IT=ILEVEL2; U2=U(2);
if U(3)~=U(1)
    while UT < U(3)
        psi = (U2-U(1))/(U(3)-U(1));
        xpsi=XX(INDEX(1))+psi*(XX(INDEX(3))-XX(INDEX(1)));
        ypsi=YY(INDEX(1))+psi*(YY(INDEX(3))-YY(INDEX(1)));
        if U(3)~=U(2)
            psi2=(U2-U(2))/(U(3)-U(2));
        else
            psi2=0;
        end;
        xc = XX(INDEX(2))+psi2*(XX(INDEX(3))-XX(INDEX(2)));
        yc=YY(INDEX(2))+psi2*(YY(INDEX(3))-YY(INDEX(2)));
        xe(1)=XX(INDEX(3));xe(2)=xc;xe(3)=xpsi;
        ye(1)=YY(INDEX(3));ye(2)=yc;ye(3)=ypsi;
        hh(IT,:)=...
        fill(xe,ye,MM(IT,:), 'LineWidth', .0000001, 'LineStyle', 'none');
        cc(IT,1)=UT;
        UT = UT+STP; U2=UT; IT = IT+1;
    end;
end;

```

```

        if IT>NCL
            IT=NCL;
        end;
    end;
else
    xe(1)=XX(INDEX(1));xe(2)=XX(INDEX(2));xe(3)=XX(INDEX(3));
    ye(1)=YY(INDEX(1));ye(2)=YY(INDEX(2));ye(3)=YY(INDEX(3));
    hh(IT,:)=...
    fill(xe,ye,MM(IT,:), 'LineWidth',.0000001, 'LineStyle','none');
end;

% End of function LPLOT

```

## Initialize.m

```

%----- function Initialize -----
function [UG]=Initialize(NN,NIC,NV,IC);

% SPECIFICATION: This function serves to create a initial UG with
%                 corresponding IC
%
% NIC:   Number of IC (this is initial linear momentum in our case)
% NN:    Number of nodes
% NV(I): Number of the node of the initial condition n° I
% IC(I): Value of the initial condition n° I

%----- Start with F=I (no deformation),
%                 p=0 (no velocities applied)

NQ=6*NN;

UG=zeros(NQ,1);

for Node=1:NN

    Inc=6*(Node-1);

    % UG(1+Inc,1)=0; % p1=0
    % UG(2+Inc,1)=0; % p2=0
    % UG(3+Inc,1)=1; % F11=1
    % UG(4+Inc,1)=0; % F12=0
    % UG(5+Inc,1)=0; % F21=0
    % UG(6+Inc,1)=1; % F22=1
end;

%----- Modify for Initial Conditions -----

for I = 1:NIC
    Dof = NV(I);
    UG(Dof,1) = IC(I);
end;

% End of function Initialize

```

## Global\_M\_Lumped.m

```

%----- function Global_M_Lumped -----
function [M_Lumped]=Global_M_Lumped(NQ,X,NOC,NE);
%
% SPECIFICATION: This function serves to create the lumped mass matrix
%

```

## A TWO-STEP TAYLOR-GALERKIN ALGORITHM APPLIED TO LAGRANGIAN DYNAMICS

```

% DESCRIPTION OF MAIN VARIABLES:
%
% NQ:      Total number of degrees of freedom NQ=6*NN
% NOC:     Number of Components
% X:       Coordinates
% M_Lumped: Global lumped mass matrix

%----- Global Stiffness Matrix

M_Lumped = zeros(NQ,1);

for N = 1:NE

%--- Getting global numeration
    Node1 = NOC(N,1);
    Node2 = NOC(N,2);
    Node3 = NOC(N,3);

%--- Getting coordinates
    X1 = X(Node1,1);
    Y1 = X(Node1,2);
    X2 = X(Node2,1);
    Y2 = X(Node2,2);
    X3 = X(Node3,1);
    Y3 = X(Node3,2);

%--- Jacobian determinant and area of element
    Area = 0.5*((X1-X3)*(Y2-Y3)-(X2-X3)*(Y1-Y3));

% --- Assembling matrix

    M_Lumped=Assemble_M(M_Lumped,Node1,Area); % Contribution 1st node

    M_Lumped=Assemble_M(M_Lumped,Node2,Area); % Contribution 2nd node

    M_Lumped=Assemble_M(M_Lumped,Node3,Area); % Contribution 3rd node

end; % End of for loop N = 1:NE

% End of function Global_M_Lumped

%----- function Assemble_M -----

function [M_Lumped]=Assemble_M(M_Lumped,Node,Area)
%
% SPECIFICATION: This function adds the contribution
%                of elemental Area to a Node and
%                modifies the global M_Lumped

    Start=6*(Node-1)+1;
    End=Start+5;
    M_Lumped(Start:End,1)=M_Lumped(Start:End,1)+(Area/3)*ones(6,1);

% End of function Assemble_M


```

### Wall\_BC.m

```

%----- function Wall_BC -----

function [NU]=Wall_BC(NBC,NB1,NB2,BT);
%
% SPECIFICATION: This functions serves for putting a penalty where
%                the BC are located in a strong way on p (linear momentum)
%                Strictly it is not necessary to do this but results

```

A TWO-STEP TAYLOR-GALERKIN ALGORITHM APPLIED TO LAGRANGIAN DYNAMICS

```

%           are better in practice
%
% NBC:      Number of BC (boundary conditions)
% NU(side): Number of the dof of the boundary condition n° I
% BC(side): Value of the boundary condition n° I
% NBE:      Element on the boundary
% NB1,NB2:  Nodes on the boundary side
% BT:      Boundary type
% BT=      0 : Clamped wall                px=0, py=0
% BT=     -1 : Hinges parallel to X        py=0
% BT=     -2 : Hinges parallel to Y        px=0
% BT=    -10 : Movement parallel to X      py=0, Pn=0
% BT=    -20 : Movement parallel to Y     px=0, Pn=0
% BT=      1 : Free boundary               No strog condition
% BT=     10 : Load                        No strog condition

Point=0;

for side = 1:NBC

    Dof_Start1=6*(NB1(side)-1);
    Dof_Start2=6*(NB2(side)-1);

    if (BT(side)==0) % px=0, py=0 Fxy=Fyx=0

        Point=Point+1;

        NU(Point)    = (Dof_Start1+1);
        NU(Point+1)  = (Dof_Start1+2);
        NU(Point+2)  = (Dof_Start1+4);
        NU(Point+3)  = (Dof_Start1+5);

        NU(Point+4)  = (Dof_Start2+1);
        NU(Point+5)  = (Dof_Start2+2);
        NU(Point+6)  = (Dof_Start2+4);
        NU(Point+7)  = (Dof_Start2+5);

        Point=Point+7;

    elseif (BT(side)==-1) % py=0 Fxy=Fyx=0

        Point=Point+1;

        NU(Point)    = (Dof_Start1+2);
        NU(Point+1)  = (Dof_Start1+4);
        NU(Point+2)  = (Dof_Start1+5);

        NU(Point+3)  = (Dof_Start2+2);
        NU(Point+4)  = (Dof_Start2+4);
        NU(Point+5)  = (Dof_Start2+5);

        Point=Point+5;

    elseif (BT(side)==-2) % px=0 Fxy=Fyx=0

        Point=Point+1;

        NU(Point)    = (Dof_Start1+1);
        NU(Point+1)  = (Dof_Start1+4);
        NU(Point+2)  = (Dof_Start1+5);

        NU(Point+3)  = (Dof_Start2+1);
        NU(Point+4)  = (Dof_Start2+4);
        NU(Point+5)  = (Dof_Start2+5);

        Point=Point+5;

    elseif (BT(side)==-10) % py=0 Fxx=Fyy=1

```

```

    Point=Point+1;

    NU(Point)    = (Dof_Start1+2);
    NU(Point+1)  = (Dof_Start1+3);
    NU(Point+2)  = (Dof_Start1+6);

    NU(Point+3)  = (Dof_Start2+2);
    NU(Point+4)  = (Dof_Start2+3);
    NU(Point+5)  = (Dof_Start2+6);

    Point=Point+5;

elseif (BT(side)==-20) % px=0 Fyy=Fxx=1

    Point=Point+1;

    NU(Point)    = (Dof_Start1+1);
    NU(Point+1)  = (Dof_Start1+3);
    NU(Point+2)  = (Dof_Start1+6);

    NU(Point+3)  = (Dof_Start2+1);
    NU(Point+4)  = (Dof_Start2+3);
    NU(Point+5)  = (Dof_Start2+6);

    Point=Point+5;

end;

end;

% End of function Wall_BC

```

## Geometry.m

```

function [DJ,B,Normal,Length,N1,N2] = Geometry(NE,X,NOC,NBC,NBE,NB1,NB2)
%
% SPECIFICATION: This function serves to create the geometric
%                 properties of the element
%
% MAIN VARIABLES
%
% OUTPUT
% DJ:            Jacobian of the local-global transformation DJ=2A
%                if counter clock wise notation of nodes DJ>0
% X:             Coordinates
% NOC:          Number of components
% N:            Number of the element
% Normal:       Normal to the side on the boundary element
% Length:       Length of the side on the boundary of the element
% N1,N2:        Local coodinates of the nodes on teh boundary NB1,NB2
%
% INPUT
% NE:           Number of elements
% X:           Global coordinates of nodes
% NOC:         Local numeration of nodes
% NBC:         Number of Boundary Conditions
% NBE:         Number of Element on the boundary
% NB1,NB2:     Nodes on the Boundary side

DJ = zeros(1,NE);
B = zeros(3,2,NE);
Normal = zeros(2,1,NBC);
Length = zeros(1,NBC);
N1 = zeros(1,NBC);
N2 = zeros(1,NBC);

```

A TWO-STEP TAYLOR-GALERKIN ALGORITHM APPLIED TO LAGRANGIAN DYNAMICS

```

for N = 1:NE
    [DJ(1,N),B(1:3,1:2,N)] = Voigt(N,X,NOC);
end;

for side = 1:NBC
    [Normal(1:2,1,side),Length(1,side),N1(1,side),N2(1,side)]=...
        Get_Normal(X,NOC,NB1(side),NB2(side),NBE(side));
end;

% End of function Geometry

%----- function Voigt -----
function [DJ,B] = Voigt(N,X,NOC);

% SPECIFICATION: This procedure serves to create the Jacobian
%                 and Voigt B matrix

% MAIN VARIABLES

% DJ:             Jacobian of the local-global transformation DJ=2A
%                 if counter clock wise notation of nodes DJ>0
% B(i,j):         dNi/dXj
% X:              Coordinates
% NOC:            Number of components
% N:              Number of the element
% I1,I2,I3:       Global notation of node

%--- Get coordinates

    I1 = NOC(N, 1);
    I2 = NOC(N, 2);
    I3 = NOC(N, 3);
    X1 = X(I1, 1);
    Y1 = X(I1, 2);
    X2 = X(I2, 1);
    Y2 = X(I2, 2);
    X3 = X(I3, 1);
    Y3 = X(I3, 2);

%--- Definition of B() Matrix (Voigt matrix)

%   B(i,j)= partial of shape function Ni respect to Xj

    X21 = X2 - X1;
    X32 = X3 - X2;
    X13 = X1 - X3;
    Y12 = Y1 - Y2;
    Y23 = Y2 - Y3;
    Y31 = Y3 - Y1;

    DJ=X13*Y23-X32*Y31;

    B(1, 1) = Y23 / DJ;
    B(1, 2) = X32 / DJ;
    B(2, 1) = Y31 / DJ;
    B(2, 2) = X13 / DJ;
    B(3, 1) = Y12 / DJ;
    B(3, 2) = X21 / DJ;

% End of function Voigt

%----- function Get_Normal -----
%
function [Normal,Length,N1,N2]=Get_Normal(X,NOC,NB1,NB2,N);

```

```

%
% SPECIFICATION: This function serves to calculate:
%               - The normal between NB1 NB2 for the element N,
%               - The length of the side
%               - The local coordinates N1,N2 of NB1,NB2

    if      (NB1==NOC(N,1)); N1=1;
    elseif (NB1==NOC(N,2)); N1=2;
    elseif (NB1==NOC(N,3)); N1=3; end;
    if      (NB2==NOC(N,1)); N2=1;
    elseif (NB2==NOC(N,2)); N2=2;
    elseif (NB2==NOC(N,3)); N2=3; end;

    if (N1==N2)

        Normal=zeros(2,1);

        Length=0;

    else

        N3 = 6-N1-N2;
        NC3 = NOC(N,N3);

        X1 = X(NB1,1);
        Y1 = X(NB1,2);
        X2 = X(NB2,1);
        Y2 = X(NB2,2);
        X3 = X(NC3,1);
        Y3 = X(NC3,2);

        X21 = X2 - X1;
        X32 = X3 - X2;
        X13 = X1 - X3;
        Y12 = Y1 - Y2;
        Y23 = Y2 - Y3;
        Y31 = Y3 - Y1;

        DJ=X13*Y23-X32*Y31;

        Length=sqrt(X21*X21+Y12*Y12);

        Normal(1,1)=-sign(DJ)*Y12/Length;
        Normal(2,1)=-sign(DJ)*X21/Length;

    end; % End of if

% End of function Get_Normal

```

## Global\_RHS.m

```

%----- function Global_RHS -----
function [RHS,Error_P]=Global_RHS(DJ,B,Normal,Length,N1,N2,NE,NQ,NOC,...
                                NBE,BT,NU,Mat_Prop,UG,Time,dt);

%
% SPECIFICATION: This procedure serves to create the system
%
% DESCRIPTION OF MAIN VARIABLES:
%
% NN:    Number of nodes
% NE:    Number of elements
% NOC:   Number of Components
% RHS:   Right hand side vector in global system
% RHSE:  Contribution to the right hand side vector of one element
% UE:    Unknowns=transpose[ p1 F1 | p2 F2 | ... ] 6*NN rows, 1 column
%        (for triang2d, 18x1)

```

A TWO-STEP TAYLOR-GALERKIN ALGORITHM APPLIED TO LAGRANGIAN DYNAMICS

```

% UG:    Unknowns in global coordinates
% NDIM:  Number of coordinates per node (i.e. NDIM=2 in 2D)
% dt:    Step of time
% B:     Bij is the partial of Ni respect to Xj (Voigt matrix)
% BT:    Boundary type
% BT=    0 : Clamped wall                px=0, py=0
% BT=   -1 : Hinges parallel to X        py=0
% BT=   -2 : Hinges parallel to Y        px=0
% BT=  -10 : Movement parallel to X      py=0, Pn=0
% BT=  -20 : Movement parallel to Y     px=0, Pn=0
% BT=    1 : Free boundary                Pn=0
% BT=   10 : Load                          Traction=Pn
% DJ:    Determinant of the Jacobian of the transformation
%        DJ=2*Area of element

% ---- Initializing global RHS and Error variables
RHS = zeros(NQ,1);
NN=NQ/6;
P_Max = -1e300*ones(NN,1);
P_Min = 1e300*ones(NN,1);
Error_P= zeros(NN,1);

% ---- Material properties
Young = Mat_Prop(1);
Poisson = Mat_Prop(2);
Kappa=Young/(3-6*Poisson);
Dens = Mat_Prop(3);
Lambda_Vis = Mat_Prop(4);
Mu_Vis = Mat_Prop(5);

for N = 1:NE

    % ---- Geometric properties
    % Global nodes mumeration
    Node1 = NOC(N,1);
    Node2 = NOC(N,2);
    Node3 = NOC(N,3);

    % Other geometric properties
    DJE=DJ(1,N);
    BE=B(1:3,1:2,N);

    % ----- At time n ...
    % Getting local element unknowns from global unknowns...
    [UE] = Local(UG,Node1,Node2,Node3);

    % Calculation of old flux
    [FluxE] = Flux_Unk(Young,Poisson,Dens,UE);

    % ----- At time n+1/2 ...
    % Prediction of new local element unknowns
    [dUE] = Unk_Predict(FluxE,BE,dt);
    UE=UE+dUE;

    % ----- Start Error control structure over pressure
    F1(1,1) = UE(3);    F1(1,2) = UE(4);    % F Node1
    F1(2,1) = UE(5);    F1(2,2) = UE(6);
    F2(1,1) = UE(9);    F2(1,2) = UE(10); % F Node2
    F2(2,1) = UE(11);   F2(2,2) = UE(12);
    F3(1,1) = UE(15);   F3(1,2) = UE(16); % F Node3
    F3(2,1) = UE(17);   F3(2,2) = UE(18);
    % Volumetric Stress: Pressure
    P1=Kappa*(det(F1)-1);                % PV1
    P_Max(Node1) = max(P_Max(Node1),P1);
    P_Min(Node1) = min(P_Min(Node1),P1);
    P2=Kappa*(det(F2)-1);                % PV2
    P_Max(Node2) = max(P_Max(Node2),P2);

```

A TWO-STEP TAYLOR-GALERKIN ALGORITHM APPLIED TO LAGRANGIAN DYNAMICS

```

P_Min(Node2) = min(P_Min(Node2),P2);
P3=Kappa*(det(F3)-1); % PV3
P_Max(Node3) = max(P_Max(Node3),P3);
P_Min(Node3) = min(P_Min(Node3),P3);
% ----- End of error control structure

% Prediction of new local element flux
[FluxE] = Flux_Unk(Young,Poisson,Dens,UE);

% ----- Viscous flux
[FluxE_Vis]=Flux_Vis_Unk(Lambda_Vis,Mu_Vis,Dens,UE,BE);

% Total flux without viscosity
FluxE=FluxE-FluxE_Vis;

% ----- Getting internal contribution of the element
RHSE=Internal_RHS(FluxE,DJE,BE,dt);

% ----- Getting external contribution of the boundary

for side=1:length(NBE)

    if (N==NBE(side))

        NormalE=Normal(1:2,1,side);
        LengthE=Length(1,side);
        N1E=N1(1,side);
        N2E=N2(1,side);

        Start1 = 6*(N1E-1)+1;
        End1    = Start1+5;
        Start2 = 6*(N2E-1)+1;
        End2    = Start2+5;

        FluxN(1:6 ,1) = FluxE(Start1:End1,1:2)*NormalE; % Node1
        FluxN(1:6 ,1) = Modify_Flux(FluxN(1:6 ,1),BT(side),Time);

        FluxN(7:12,1) = FluxE(Start2:End2,1:2)*NormalE; % Node2
        FluxN(7:12,1) = Modify_Flux(FluxN(7:12,1),BT(side),Time);

        % Calculation of external contribution of element to RHS
        [External]=External_RHS(FluxN,N1E,N2E,LengthE,dt);

        RHSE=RHSE+External;

    end; % End of if (N==NBE(side))

end; % End of for side=1:length(NBE)

% ---- Assembling
[RHS] = Assemble(RHS,Node1,Node2,Node3,RHSE);

end; % End of general loop

% ---- Modify RHS for Wall conditions (Strong form)
[RHS] = Modify_Wall_BC(RHS,NU);

% ---- Calculate deviation error of volumetric pressure
Error_P(1:NN) = P_Max(1:NN) - P_Min(1:NN);

% End of function Global_RHS

%----- function Modify_Flux -----
%
function [Flux] = Modify_Flux(Flux,BT,Time)
%
% SPECIFICATION: This function modifies the Flux of the Element

```

A TWO-STEP TAYLOR-GALERKIN ALGORITHM APPLIED TO LAGRANGIAN DYNAMICS

```

%           FluxE depending on the Boundary Type BT
%
% BT=      0 : Clamped wall           px=0, py=0
% BT=     -1 : Hinges parallel to X   py=0
% BT=     -2 : Hinges parallel to Y   px=0
% BT=    -10 : Movement parallel to X   py=0, Pn=0
% BT=    -20 : Movement parallel to Y   px=0, Pn=0
% BT=      1 : Free boundary          Pn=0
% BT=     10 : Load                    Traction=Pn

if (BT==0)           % px=0, py=0

    Flux(3,1)=0;
    Flux(4,1)=0;
    Flux(5,1)=0;
    Flux(6,1)=0;

elseif (BT==-1)     % py=0

    Flux(5,1)=0;
    Flux(6,1)=0;

elseif (BT==-2)     % px=0

    Flux(3,1)=0;
    Flux(4,1)=0;

elseif (BT==-10)    % py=0 Pn=0

    Flux(1,1)=0;
    Flux(2,1)=0;
    Flux(5,1)=0;
    Flux(6,1)=0;

elseif (BT==-20)    % px=0 Pn=0

    Flux(1,1)=0;
    Flux(2,1)=0;
    Flux(3,1)=0;
    Flux(4,1)=0;

elseif (BT==1)      % Free condition Pn=0

    Flux(1,1)=0;
    Flux(2,1)=0;

elseif (BT==10)     % Load condition Pn=traction

    Flux(1,1)=-1*Traction(Time,1);
    Flux(2,1)=-1*Traction(Time,2);

end;

% End of function Modify_Flux

%----- function Internal_RHS -----
%
function [RHSE]=Internal_RHS(FluxE,DJ,B,dt);
%
% SPECIFICATION: This function serves to calculate the contribution
%                 to RHS of each element considering the internal
%                 flux and external loss through the boundary.
%
Flux1 = FluxE( 1:6, 1:2);
Flux2 = FluxE( 7:12, 1:2);
Flux3 = FluxE(13:18, 1:2);

```

A TWO-STEP TAYLOR-GALERKIN ALGORITHM APPLIED TO LAGRANGIAN DYNAMICS

```

Average_Flux = zeros(6,2);
Average_Flux = (Flux1+Flux2+Flux3)/3;

Area=0.5*DJ;

Int_Rhs=zeros(18,1);
Int_Rhs( 1:6 ,1) = dt*Area*(Average_Flux(1:6,1)*B(1,1)+...
                             Average_Flux(1:6,2)*B(1,2));    % Node 1
Int_Rhs( 7:12,1) = dt*Area*(Average_Flux(1:6,1)*B(2,1)+...
                             Average_Flux(1:6,2)*B(2,2));    % Node 2
Int_Rhs(13:18,1) = dt*Area*(Average_Flux(1:6,1)*B(3,1)+...
                             Average_Flux(1:6,2)*B(3,2));    % Node 3

RHSE=Int_Rhs;

% End of function Internal_RHS

%----- function External_RHS -----
%
function [External] = External_RHS(FluxN,N1,N2,Length,dt);
%
% SPECIFICATION: This function serves to calculate the contribution
%                 to RHS of each element considering the external
%                 loss through the boundary N1-N2
%
if (N1==N2)
    External=zeros(18,1);
else

    % Calculation of starting d.o.f. for N1 and N2

    External = zeros(18,1);

    Start1 = 6*(N1-1)+1;
    End1    = Start1+5;

    Start2 = 6*(N2-1)+1;
    End2    = Start2+5;

    Flux1 = FluxN(1:6 ,1);
    Flux2 = FluxN(7:12,1);

    External(Start1:End1,1) = -(0.5)*dt*((2*Flux1+1*Flux2)/3)*Length;
    External(Start2:End2,1) = -(0.5)*dt*((2*Flux2+1*Flux1)/3)*Length;

end;

% End of function External_RHS

%----- function Local -----
%
function [UE]=Local(UG,Node1,Node2,Node3);
%
% SPECIFICATION: This function serves to obtain
%                 the local unknowns from global coordinates
%
% UG: Global vector
% UE: Local vector for the element

UE=zeros(18,1);

% Node 1
Start1 = 6*(Node1-1)+1;
End1    = Start1+5;
UE(1:6,1) = UG(Start1:End1,1);

```

```

% Node 2
Start2 = 6*(Node2-1)+1;
End2   = Start2+5;
UE(7:12,1) = UG(Start2:End2,1);

% Node 3
Start3 = 6*(Node3-1)+1;
End3   = Start3+5;
UE(13:18,1) = UG(Start3:End3,1);

% End of function Local

%----- function Modify_Wall_BC -----
%
function [RHS]=Modify_Wall_BC(RHS,NU);
%
% SPECIFICATION: This function serves to modify global RHS vector
%                for the Wall Boundary Conditions in the nodes
%                given by NU

for I = 1:length(NU)
    N = NU(I);
    RHS(N) = 0;
end;

% End of function Modify_Wall_BC

%----- function Assemble -----
%
function [RHS]=Assemble(RHS,Node1,Node2,Node3,RHSE)
%
% SPECIFICATION: This function adds the contribution
%                of elemental RHS to the global RHS
%

% Node 1
Start1 = 6*(Node1-1)+1;
End1   = Start1+5;
RHS(Start1:End1,1) = RHS(Start1:End1,1)+RHSE( 1:6 ,1);

% Node 2
Start2 = 6*(Node2-1)+1;
End2   = Start2+5;
RHS(Start2:End2,1) = RHS(Start2:End2,1)+RHSE( 7:12,1);

% Node 3
Start3 = 6*(Node3-1)+1;
End3   = Start3+5;
RHS(Start3:End3,1) = RHS(Start3:End3,1)+RHSE(13:18,1);

% End of function Assemble

```

## Unk\_Predict.m

```

%----- function Unk_Predict -----
function [Inc_Unk_Pred]=Unk_Predict(Flux,B,dt);
%
% SPECIFICATION: This function serves to predict the unknowns per element
%                at the time step n+1/2 from the Flux given at time n
%                (at element level), this is, the Taylor prediction of
%                the half-step

```

## A TWO-STEP TAYLOR-GALERKIN ALGORITHM APPLIED TO LAGRANGIAN DYNAMICS

```

%
% DESCRIPTION OF MAIN VARIABLES:
%
% Flux:          -1*transpose[ P1 v1xI | P2 v2xI | P3 v3xI .... ] 6*NN rows,
%                2 columns (for triang2d, 18x2)
% Unknowns:      transpose[ p1 F1 | p2 F2 | ... ] 6*NN rows,
%                1 column (for triang2d, 18x1)
% B:             Bij is the partial of Ni respect to Xj (Voigt matrix)
% dt:           Step of time

Inc_Node_Pred = zeros(6,1);
Inc_Unk_Pred  = zeros(18,1);

% PREDICTOR STEP
Inc_Node_Pred = -0.5*dt*Div_Flux(Flux,B);

Inc_Unk_Pred(1:6,1) = Inc_Node_Pred;
Inc_Unk_Pred(7:12,1) = Inc_Node_Pred;
Inc_Unk_Pred(13:18,1) = Inc_Node_Pred;

% End of Unk_Predict

%-----function Div_Flux -----
%
function [DFlux]=Div_Flux(FluxE,B);
%
% SPECIFICATION: This Fluxction serves to calculate the divergence
%                of the Flux of the element
%
% DESCRIPTION OF MAIN VARIABLES:
%
% INPUT
%
% Flux:          Elemental flux
%                -1*transpose[ P1 v1xI | P2 v2xI | P3 v3xI .... ] 6*NN rows,
%                2 columns (for triang2d, 18x2)
% dt:           Step of time
% B:           Bij is the partial of Ni respect to Xj (Voigt matrix)
%
% OUTPUT
%
% DFlux:        Divergence of Flux

DFlux=zeros(6,1);

DFlux=(FluxE(1:6,1)*B(1,1)+FluxE(7:12,1)*B(2,1)+FluxE(13:18,1)*B(3,1)+...
        FluxE(1:6,2)*B(1,2)+FluxE(7:12,2)*B(2,2)+FluxE(13:18,2)*B(3,2));

% End of Div_Flux function

```

### Flux\_Unk.m

```

%----- function Flux_Unk -----
%
function [FluxE]=Flux_Unk(Young, Poisson, Dens,UE);
%
% SPECIFICATION: This function serves to calculate the Flux from the
%                Element Unknowns at the same time step for one element
%
% DESCRIPTION OF MAIN VARIABLES:
%
% INPUT
%
% Young:        Young elastic parameter

```

## A TWO-STEP TAYLOR-GALERKIN ALGORITHM APPLIED TO LAGRANGIAN DYNAMICS

```

% Poisson:      Poisson elastic parameter
% Dens:        Density
% UE:         transpose[ p1 F1 | p2 F2 | p3 F3 ]
%            6*Nodes rows, 1 column (for triang2d, 18x1)
%
% OUTPUT
% FluxE:      -1*transpose[ P1 v1xI | P2 v2xI | P3 v3xI ]
%            6*NN rows, NDIM columns (for triang2d, 18x2)
% F:         Deformation gradient tensor

    FluxE=zeros(18,2);

    UN=UE(1:6,1);   %Node1
    FluxE(1:6,1:2)=Nodal_Flux(UN,Young,Poisson,Dens);

    UN=UE(7:12,1); %Node2
    FluxE(7:12,1:2)=Nodal_Flux(UN,Young,Poisson,Dens);

    UN=UE(13:18,1); %Node3
    FluxE(13:18,1:2)=Nodal_Flux(UN,Young,Poisson,Dens);

% End of function Flux_Unk;

%----- function Nodal_Flux -----
function [FluxN]=Nodal_Flux(UN,Young,Poisson,Dens);
%
% SPECIFICATION: This function serves to obtain the nodal Flux
%                from the vector of Nodal Unknowns UN
%
% Deformation gradient tensor F

    F(1,1)=UN(3);   F(1,2)=UN(4);
    F(2,1)=UN(5);   F(2,2)=UN(6);

% Piola stress tensor
    P=Piola(Young,Poisson,F);

    FluxN=zeros(6,2);
    FluxN(1:2,1:2)=-P(1:2,1:2);
    FluxN(3,1)=-UN(1)/Dens;   FluxN(3,2)= 0;
    FluxN(4,1)= 0;           FluxN(4,2)=-UN(1)/Dens;
    FluxN(5,1)=-UN(2)/Dens;   FluxN(5,2)= 0;
    FluxN(6,1)= 0;           FluxN(6,2)=-UN(2)/Dens;

% End of function Nodal_Flux

```

### Flux\_Unk\_Vis.m

```

%----- function Flux_Unk_Vis -----
function [FluxE_Vis]=Flux_Vis_Unk(Lambda_Vis,Mu_Vis,Dens,UE,B);
%
% SPECIFICATION: This function serves to calculate the Flux from the
%                Element Unknowns at the same time step for one element
%
% DESCRIPTION OF MAIN VARIABLES:
%
% INPUT
%
% Lambda_vis: Volumetric viscous parameter
% Mu_Vis:     Shear viscous parameter
% Dens:       Density
% UE:        transpose[ p1 F1 | p2 F2 | ... ]
%            6*Nodes rows, 1 column (for triang2d, 18x1)

```

A TWO-STEP TAYLOR-GALERKIN ALGORITHM APPLIED TO LAGRANGIAN DYNAMICS

```

%
% OUTPUT
% FluxE_Vis:  -1*transpose[ P1 v1xI | P2 v2xI | P3 v3xI .... ]
%             6*NN rows, NDIM columns (for triang2d, 18x2)
% F:          Deformation gradient tensor

FluxE_Vis=zeros(18,2);

% Linear momentum

p1=UE(1:2,1);
p2=UE(7:8,1);
p3=UE(13:14,1);

% Gradient of linear momentum

[Grad_p]=Grad_Lin_Mom(p1,p2,p3,B);

for Node=1:3
    Inc=6*(Node-1);          % Inc=0,6,12,...
    U_Nodal=UE((1+Inc):(6+Inc));
    Flux_Nodal=zeros(6,2);
    Flux_Nodal=Nodal_Flux(U_Nodal,Lambda_Vis,Mu_Vis,Dens,Grad_p);
    FluxE_Vis((1+Inc):(6+Inc),1:2)=Flux_Nodal;
end; % End of for

% End of function Flux_Unk;

%----- function Nodal_Flux -----
function [FluxN]=Nodal_Flux(UN,Lambda_Vis,Mu_Vis,Dens,Grad_p);
% SPECIFICATION: This function serves to obtain the nodal Flux
%               from the vector of Nodal Unknowns UN

F=Get_F(UN);
P=zeros(2,2);
P_Vis=Piola_Vis(Lambda_Vis,Mu_Vis,Dens,F,Grad_p);

FluxN=zeros(6,2);
FluxN(1:2,1:2)=+1.0*P_Vis;

% End of procedure Put_FluxE_Vis

%----- function Get_F -----
function [F]=Get_F(UN);

% SPECIFICATION: This function serves to obtain teh deformation F
%               from the vector of Nodal Unknowns UN

F=zeros(2,2);
F(1,1)=UN(3);   F(1,2)=UN(4);
F(2,1)=UN(5);   F(2,2)=UN(6);

% End of function Get_F

%----- function Grad_Lin_Mom -----
function [Grad_p]=Grad_Lin_Mom(p1,p2,p3,B);
%
% SPECIFICATION: This Fluxction serves to calculate the gradient
%               of the vector of Unknowns
%
% DESCRIPTION OF MAIN VARIABLES:
%
% INPUT
%
% pi=   linear momentum of node i

```

```

%           p=[ px
%             py ]
% dt:      Step of time
% B:       Bij is the partial of Ni respect to Xj (Voigt matrix)
%
% OUTPUT
%
% Grad_p:  Gradient of Unknowns
%          | px,x  px,y |
%          | py,x  py,y |

Grad_p=zeros(2,2);

% First column: derivatives with respect to X
Grad_p(1:2,1)=p1(1:2)*B(1,1)+p2(1:2)*B(2,1)+p3(1:2)*B(3,1);

% Second column: derivatives with respect to Y
Grad_p(1:2,2)=p1(1:2)*B(1,2)+p2(1:2)*B(2,2)+p3(1:2)*B(3,2);

% End of Grad_Lin_Mom function

```

## Piola.m

```

%----- function Piola -----
function [P]=Piola(Young,Poisson,F);
%
% SPECIFICATION: This function serves to calculate the first Piola
%                stress tensor in 2D problems
%
% DESCRIPTION OF MAIN VARIABLES:
%
% INPUT:
%
% F :           Deformation gradient tensor
% Young:        Young elastic parameter
% Poisson:      Poisson elastic parameter
%
% OUTPUT:
%
% P:           First Piola-Kirchhoff stress tensor
%
%
% Mode
Model=4;

% 1-4: Nearly Incompressible
% 1 = Hookean Linear Elastic
% 2 = Classical Neo-Hookean
% 3 = Extended Neo-Hookean
% 4 = Bonet's model
% 5 = Compressible Blatz&Ko
% 6 = Small deformations

G=0.5*Young/(1.0+Poisson); % Shear modulus (also called Mu)
Lambda=2.0*Poisson*G/(1.0-2.0*Poisson);

% Calculation of Piola Kirchhoff stress tensor
J=det(F);
F_INV_T = inv(transpose(F));
C=transpose(F)*F; % Cauchy-Green tensor

if (Model==1) % Linear elastic (Saint-Venant&Kirchhoff)

    E=0.5*(C-ones(2,2)); % Green-Lagrange tensor
    P=Lambda*trace(E)*F+2*G*F*E;

```

```

elseif (Model==2) % Classical Isotropic Neo-Hookean
    P=Lambda*log(J)*F_INV_T+G*(F-F_INV_T);
elseif (Model==3) % Extended Isotropic Neo-Hookean (Simo)
    P=0.5*Lambda*(J*J-1)*F_INV_T+G*(F-F_INV_T);
elseif (Model==4) % Transversely Isotropic Neo-Hookean
    % (Bonet&Burton)
    Kappa=Lambda+(2/3)*G; % Bulk modulus
    P=Kappa*(J-1)*J*F_INV_T+...
    G*(J^(-2/3))*(F-((1/3)*(1+trace(C))*F_INV_T));
elseif (Model==5) % Simple Compressible (Blatz&Ko)
    B=F*transpose(F);
    P=G*(J*eye(2)-inv(B))*F_INV_T;
elseif (Model==6) % Small deformations
    Kappa=Lambda+(2/3)*G; % Bulk modulus
    P=G*(F+transpose(F)-(2/3)*(1+trace(F))*eye(2))+Kappa*(trace(F)-
2)*eye(2);
end;

% End of Piola function

```

## Piola\_Vis.m

```

%----- function Piola_Vis -----
function [P_Vis]=Piola_Vis(Lambda_Vis,Mu_Vis,Dens,F,Grad_p);
%
% SPECIFICATION: This function serves to calculate the viscous 1st
% Piola Kirchhoff stress tensor in 2D problems
%
% DESCRIPTION OF MAIN VARIABLES:
%
% INPUT:
% Lambda_Vis, Mu_Vis : Viscous parameters
%
% Grad_p: Gradient of Unknowns
%          | px,x  px,y |
%          | py,x  py,y |
% OUTPUT:
%
% Pv:      Viscous Piola-Kirchhoff stress tensor
%
% Cauchy-Green deformation tensor
C = transpose(F)*F;
% Jacobian
J = det(F);
% Inverse of the transpose of F
F_INV_T = inv(transpose(F));

```

```
% Viscous constitutive model (for numerical purposes only)

Cons1 = (Lambda_Vis/(J*Dens)); % Volumetric constant
Cons2 = (Mu_Vis/(J*Dens));    % Shear constant

P_Vis = Cons1*trace(F_INV_T*transpose(Grad_p))*F_INV_T+...
        Cons2*Grad_p*inv(C)+...
        Cons2*F_INV_T*transpose(Grad_p)*F_INV_T;

% End of Piola_Vis function
```

## Mesh\_Error.m

```
function [Error_RF,DE] = Mesh_Error(NOC,NE,UG,B,Young,Poisson)
%
% SPECIFICATION: This function serves to calculate some parameter to
%                 control the errors
%
% MAIN VARIABLES
%
% Error_RF: Average rotational of F (deformation gradient) per element
%           It must be zero because the rotational of a gradient is 0.
% DJ:      Jacobian of the transformation local-global
%           If counter clock wise notation of nodes DJ>0
% NOC:     Nodal Coordinates
% NE:     Number of elements
% UG:     Unknowns in global coordinates
% B:      Bij is the partial of Ni respect to Xj (Voigt matrix)
% Young,Poisson: elastic parameters

% Initialize
Error_RF=0;
DE=0;

for N=1:NE

    Node1 = NOC(N,1);
    Node2 = NOC(N,2);
    Node3 = NOC(N,3);

    % Voigt matrix
    BE = B(1:3,1:2,N);

    % Local unknowns
    [UE] = Local(UG,Node1,Node2,Node3);

    % Deformation gradient F

    F1(1,1) = UE(3);    F1(1,2) = UE(4);    % F Node1
    F1(2,1) = UE(5);    F1(2,2) = UE(6);
    F2(1,1) = UE(9);    F2(1,2) = UE(10);   % F Node2
    F2(2,1) = UE(11);   F2(2,2) = UE(12);
    F3(1,1) = UE(15);   F3(1,2) = UE(16);   % F Node3
    F3(2,1) = UE(17);   F3(2,2) = UE(18);

    F_Ave=(F1+F2+F3)/3;

    % 1st Piola-Kirchhoff stress tensor
    Piola1 = Piola(Young,Poisson,F1);
    Piola2 = Piola(Young,Poisson,F2);
    Piola3 = Piola(Young,Poisson,F3);
    Piola_Ave=(Piola1+Piola2+Piola3)/3;

    % Divergence of 1st P-K
```

A TWO-STEP TAYLOR-GALERKIN ALGORITHM APPLIED TO LAGRANGIAN DYNAMICS

```

[DP] = Div_P(Piola1,Piola2,Piola3,BE);

% Linear momentum
p1 = UE(1:2,1);
p2 = UE(7:8,1);
p3 = UE(13:14,1);
p_Ave=(p1+p2+p3)/3;

% Linear momentum gradient
[Grad_p]=Grad_Lin_Mom(p1,p2,p3,BE);

% Rotational error averaged
[ErrorE] = Error_Rot_E(F1,F2,F3,BE);
Error_RF = Error_RF+(ErrorE/NE);

% Total derivative of entropy
[EntropyE] = Entropy_E(Grad_p,DP,p_Ave,Piola_Ave);
DE = DE + EntropyE;

end;

% End of function Error

% ----- function Error_RotF_E -----
%
function [ErrorE]=Error_Rot_E(F1,F2,F3,B);
%
% SPECIFICATION: This Function serves to calculate the error
%                 of the vector of Unknowns
% DESCRIPTION OF MAIN VARIABLES:
%
% INPUT
%
% Fi:      Deformation gradient of node i
%          | Fxx  Fxy |
%          | Fyx  Fyy |
% dt:      Increment of ime step
% B:       Bij is the partial of Ni respect to Xj (Voigt matrix)
%
% OUTPUT
%
% RotF:    Rotational of the deformation gradient F
%          | Fxy,x - Fxx,y |
%          | Fyy,x - Fyx,y |

RotF=zeros(2,1);

DxFxy = F1(1,2)*B(1,1) + F2(1,2)*B(2,1) + F3(1,2)*B(3,1);
DyFxx = F1(1,1)*B(1,2) + F2(1,1)*B(2,2) + F3(1,1)*B(3,2);
DxFyy = F1(2,2)*B(1,1) + F2(2,2)*B(2,1) + F3(2,2)*B(3,1);
DyFyx = F1(2,1)*B(1,2) + F2(2,1)*B(2,2) + F3(2,1)*B(3,2);

RotF(1,1)=DxFxy-DyFxx;
RotF(2,1)=DxFyy-DyFyx;

ErrorE=sqrt(RotF(1)*RotF(1)+RotF(2)*RotF(2));

% End of function Error_Rot_F_E

%----- function Local -----
%
function [UE]=Local(UG,Node1,Node2,Node3);
%
% SPECIFICATION: This function serves to obtain
%                 the local unknowns from global coordinates
%

```

A TWO-STEP TAYLOR-GALERKIN ALGORITHM APPLIED TO LAGRANGIAN DYNAMICS

```

% UG: Global vector
% UE: Local vector for the element

UE=zeros(18,1);

% Node 1
Start1 = 6*(Node1-1)+1;
End1   = Start1+5;
UE(1:6,1) = UG(Start1:End1,1);

% Node 2
Start2 = 6*(Node2-1)+1;
End2   = Start2+5;
UE(7:12,1) = UG(Start2:End2,1);

% Node 3
Start3 = 6*(Node3-1)+1;
End3   = Start3+5;
UE(13:18,1) = UG(Start3:End3,1);

% End of function Local

%----- function Grad_Lin_Mom -----
function [Grad_p]=Grad_Lin_Mom(p1,p2,p3,B);
%
% SPECIFICATION: This Fluxction serves to calculate the gradient
%                 of the vector of Unknowns
%
% DESCRIPTION OF MAIN VARIABLES:
%
% INPUT
%
% pi=   linear momentum of node i
%       p=[ px
%           py ]
% dt:   Step of time
% B:    Bij is the partial of Ni respect to Xj (Voigt matrix)
%
% OUTPUT
%
% Grad_p: Gradient of Unknowns
%         | px,x   px,y |
%         | py,x   py,y |

Grad_p=zeros(2,2);

% First column: derivatives with respect to X
Grad_p(1:2,1)=p1(1:2)*B(1,1)+p2(1:2)*B(2,1)+p3(1:2)*B(3,1);

% Second column: derivatives with respect to Y
Grad_p(1:2,2)=p1(1:2)*B(1,2)+p2(1:2)*B(2,2)+p3(1:2)*B(3,2);

% End of Grad_Lin_Mom function

%----- function Div_P -----
%
function [DP] = Div_P(P1,P2,P3,B)
%
% SPECIFICATION: This function serves to calculate the divergence
%                 of the 1st Piola Kirchhoff stress tensor

DP=zeros(2,1);

DP=P1(1:2,1)*B(1,1)+P2(1:2,1)*B(2,1)+P3(1:2,1)*B(3,1)+...

```

```

P1(1:2,2)*B(1,2)+P2(1:2,2)*B(2,2)+P3(1:2,2)*B(3,2);

% End of Div_P function

%----- function Entropy_E -----
%
function [EntropyE] = Entropy_E(Grad_p,DP,p_Ave,Piola_Ave)
%
% SPECIFICATION: This function serves to calculate the variation of the
%                 entropy with respect to time dE/dt in each element

EntropyE=trace(transpose(Grad_p)*Piola_Ave)+transpose(p_Ave)*DP;

% End of Entropy_E function

```

## Traction.m

```

%----- function Traction -----

function [Stress]=Traction(Time,Dim);
%
% SPECIFICATION: This function serves to specify the applied
%                 Stress on the boundary sides.
%

Nodal_Load_X = 0.025;
Nodal_Load_Y = 0.00;
T0            = 0.1;

%-----
Stress      =0.0;
Stress_Max=0.0;
if          (Dim==1)
    Stress_Max=Nodal_Load_X;
elseif     (Dim==2)
    Stress_Max=Nodal_Load_Y;
end;

% Linear function
if          (Time==0)
    Stress=0;
elseif and((Time < T0),(Time > 0))
    Stress=Stress_Max*(T0-(T0-Time))/T0;
else
    Stress=Stress_Max;
end;

% End of Load function

```

## Write\_Data.m

```

%----- function Write_Data -----

function [Done]=Write_Data(LOUT,NN,T,Displ,Vel,...
    Strain,Stress,Press,Error_Press);

fprintf(LOUT,'   Node#       Counter#       Time           DispX
DispY       VelX         VelY           EXX           EYY
EYX         SXX          SYX           SXY           Press
ErrorPress\n');

```

```

for Counter=1:length(T)

    for Node = 1:NN
        fprintf(LOUT,' %4d %15.3E %15.3E %15.3E %15.3E %15.3E %15.3E %15.3E %15.3E
%15.3E %15.3E %15.3E %15.3E %15.3E %15.3E\n',...
            Node,Counter,T(Counter),...
            Displ(Node,1,Counter),Displ(Node,2,Counter),...
            Vel(Node,1,Counter),Vel(Node,2,Counter),...
            Strain(Node,1,Counter),Strain(Node,2,Counter),Strain(Node,3,Counter),...
            Stress(Node,1,Counter),Stress(Node,2,Counter),Stress(Node,3,Counter),...
            Press(Node,1,Counter),Error_Press(Node,1,Counter));
    end;

end;

fclose(LOUT);
Done=1;

% End of function Write_data

```

## Output.m

```

function [Done] = Output(NN,NE,X,NOC,T,Disp,Vel,...
                        Strain,Stress,Pressure,Error_Press);

Continue=1;

while not(Continue==0)

    disp('-----');
    disp(' 1 : Displacement X');
    disp(' 2 : Displacement Y');
    disp(' 3 : Velocity X');
    disp(' 4 : Velocity Y');
    disp(' 5 : Normal Cauchy Stress X');
    disp(' 6 : Normal Cauchy Stress Y');
    disp(' 7 : Tangent Cauchy Stress XY');
    disp(' 8 : Normal Almansi Strain X');
    disp(' 9 : Normal Almansi Strain Y');
    disp('10 : Tangent Almansi Strain XY');
    disp('11 : Volumetric Pressure');
    disp('12 : Error on Volumetric Press');
    disp('-----');

    Key1=0;
    while not((Key1==1) | (Key1==2) | (Key1==3)...
              | (Key1==4) | (Key1==5) | (Key1==6)...
              | (Key1==7) | (Key1==8) | (Key1==9)...
              | (Key1==10) | (Key1==11) | (Key1==12))
        Key1=input('enter chosen option ');
    end;
    disp('-----');
    disp(' 1 : Nodal representation');
    disp(' 2 : Evolution of variable');
    disp('-----');
    Key2=0;
    while not((Key2==1) | (Key2==2))
        Key2=input('enter chosen option ');
    end;

    if (Key2==1) % Nodal representation

        Percentage = -1.0;
        while or((Percentage<0),(Percentage>100))
            Percentage=input('enter % of total time ');

```

```

end;

Counter=round(Percentage*length(T)/100);

% Deformed geometry
XD=zeros(NN,2);
XD(1:NN,1)=X(1:NN,1)+Disp(1:NN,1,Counter);
XD(1:NN,2)=X(1:NN,2)+Disp(1:NN,2,Counter);

% Chosen variable
if (Key1==1)
    Value=Disp(1:NN,1,Counter);
elseif (Key1==2)
    Value=Disp(1:NN,2,Counter);
elseif (Key1==3)
    Value=Vel(1:NN,1,Counter);
elseif (Key1==4)
    Value=Vel(1:NN,2,Counter);
elseif (Key1==5)
    Value=Stress(1:NN,1,Counter);
elseif (Key1==6)
    Value=Stress(1:NN,2,Counter);
elseif (Key1==7)
    Value=Stress(1:NN,3,Counter);
elseif (Key1==8)
    Value=Strain(1:NN,1,Counter);
elseif (Key1==9)
    Value=Strain(1:NN,2,Counter);
elseif (Key1==10)
    Value=Strain(1:NN,3,Counter);
elseif (Key1==11)
    Value=Pressure(1:NN,1,Counter);
elseif (Key1==12)
    Value=Error_Press(1:NN,1,Counter);
end;
disp('-----');
disp(' 1 : Contour colours plot & original geometry');
disp(' 2 : Filled colours plot & original geometry');
disp(' 3 : Contour colours plot & deformed geometry');
disp(' 4 : Filled colours plot & deformed geometry');
disp('-----');
Key3=0;
while not((Key3==1)|(Key3==2)|(Key3==3)|(Key3==4))
    Key3 = input('enter chosen option ');
end;

% Chosen plot
if (Key3==1)
    [Done] = Contour1(NN,NE,3,X,NOC,Value);
elseif (Key3==2)
    [Done] = Contour2(NN,NE,3,X,NOC,Value);
elseif (Key3==3)
    [Done] = Contour1(NN,NE,3,XD,NOC,Value);
elseif (Key3==4)
    [Done] = Contour2(NN,NE,3,XD,NOC,Value);
end;

elseif (Key2==2) % Evolution of variable in time

    Counter_Max=length(T);

    Node=0;
    while or((Node<1),(Node>NN))
        Node = input('enter Node to be analyzed ');
    end;

    if (Key1==1)
        for Counter = 1: Counter_Max

```

```

        Value(Counter)=Disp(Node,1,Counter);
    end;
elseif (Key1==2)
    for Counter = 1: Counter_Max
        Value(Counter)=Disp(Node,2,Counter);
    end;
elseif (Key1==3)
    for Counter = 1: Counter_Max
        Value(Counter)=Vel(Node,1,Counter);
    end;
elseif (Key1==4)
    for Counter = 1: Counter_Max
        Value(Counter)=Vel(Node,2,Counter);
    end;
elseif (Key1==5)
    for Counter = 1: Counter_Max
        Value(Counter)=Stress(Node,1,Counter);
    end;
elseif (Key1==6)
    for Counter = 1: Counter_Max
        Value(Counter)=Stress(Node,2,Counter);
    end;
elseif (Key1==7)
    for Counter = 1: Counter_Max
        Value(Counter)=Stress(Node,3,Counter);
    end;
elseif (Key1==8)
    for Counter = 1: Counter_Max
        Value(Counter)=Strain(Node,1,Counter);
    end;
elseif (Key1==9)
    for Counter = 1: Counter_Max
        Value(Counter)=Strain(Node,2,Counter);
    end;
elseif (Key1==10)
    for Counter = 1: Counter_Max
        Value(Counter)=Strain(Node,3,Counter);
    end;
elseif (Key1==11)
    for Counter = 1: Counter_Max
        Value(Counter)=Pressure(Node,1,Counter);
    end;
elseif (Key1==12)
    for Counter = 1: Counter_Max
        Value(Counter)=Error_Press(Node,1,Counter);
    end;
end;

plot(T,Value);

end;

Continue = input('0: EXIT | 1: CONTINUE plotting ');

end; % End of while not(Continue==0)

Done=1;
save T.txt T -ascii -tabs
save Value.txt Value -ascii -tabs
% End of function Output

```

## Postprocess.m

```

clear all;
clc;

```

```

%----- Head of the program

disp('*****');
disp('*          POST PROCESS PROGRAM          *');
disp('*          LAGRANGIAN DYNAMICS           *');
disp('* using the 2-STEP TAYLOR-GALERKIN      *');
disp('* X.M.Carreira: xmcarreira@yahoo.com *');
disp('*****');

%----- Reading data of input/output files from keyboard

disp(blanks(1));

% NOTE: Only geometrical information is extracted from input file

FILE1 = input('Input Data File Name ','s');
LINP  = fopen(FILE1,'r');

FILE2 = input('Output Data File Name ','s');
LOUT  = fopen(FILE2,'r');

[TITLE,NN,NE,NQ,X,NOC,NBC,NBE,NB1,NB2,BT,NIC,NV,IC,...
 Mat_Prop,dt,NT,PM,WF]=Read_Data(LINP);

%----- Reading written output file

    Dummy= fgets(LOUT); % Dummy line for the head of the output file
    Line = fgets(LOUT); % 1st row
while notfeof(LOUT)
    for J=1:NN
        TMP = str2num(Line);
        [Node, Counter, T(Counter),...
         Displ(Node,1:2,Counter), Vel(Node,1:2,Counter),...
         Strain(Node,1:3,Counter), Stress(Node,1:3,Counter), ...
         Press(Node,1,Counter), Error_Press(Node,1,Counter)]=...
         deal(TMP(1),TMP(2),TMP(3),TMP(4:5),TMP(6:7),TMP(8:10),...
              TMP(11:13),TMP(14),TMP(15));
        Line = fgets(LOUT);
    end;
end;

fclose(LOUT);

%----- Postprocessing data
[Done] = Output(NN,NE,X,NOC,T,Displ,Vel,...
               Strain,Stress,Press,Error_Press);

% End of Postprocessing program

```

## V. Quad 2-D code

### Global\_RHS.m

```

%----- function Global_RHS -----
function [RHS,Error_P]=Global_RHS(DJ0,DJ1,DJ2,DJ3,DJ4,B0,B1,B2,B3,B4,...
                                Normal,Length,N1,N2,NE,NQ,NOC,NBE,...
                                BT,NU,Mat_Prop,UG,Time,dt);

%
% SPECIFICATION: This procedure serves to create the system
%
% DESCRIPTION OF MAIN VARIABLES:
%
% NN:      Number of nodes

```

## A TWO-STEP TAYLOR-GALERKIN ALGORITHM APPLIED TO LAGRANGIAN DYNAMICS

```

% NE:      Number of elements
% NOC:     Number of Components
% RHS:     Right hand side vector in global system
% RHSE:    Contribution to the right hand side vector of one element
% UE:      Unknowns=transpose[ p1 F1 | p2 F2 | ... ] 6*NN rows, 1 column
%          (for quad2d, 24x1)
% UG:      Unknowns in global coordinates
% NDIM:    Number of coordinates per node (i.e. NDIM=2 in 2D)
% dt:      Step of time
% B:       Bij is the partial of Ni respect to Xj (Voigt matrix)
% BT:      Boundary type
% BT=      0 : Clamped wall                px=0, py=0
% BT=     -1 : Hinges parallel to X        py=0
% BT=     -2 : Hinges parallel to Y        px=0
% BT=    -10 : Movement parallel to X      py=0, Pn=0
% BT=    -20 : Movement parallel to Y     px=0, Pn=0
% BT=      1 : Free boundary               Pn=0
% BT=     10 : Load                        Traction=Pn
% DJ:      Determinant of the Jacobian of the transformation
%          Area=4*DJO exactly evaluated with 1 point
%          Area=DJ1+DJ2+DJ3+DJ4

% ---- Initializing global RHS and Error variables
RHS      = zeros(NQ,1);
NN=NQ/6;
P_Max    = -1e300*ones(NN,1);
P_Min    = 1e300*ones(NN,1);
Error_P= zeros(NN,1);

% ---- Material properties
Young     = Mat_Prop(1);
Poisson   = Mat_Prop(2);
Kappa=Young/(3-6*Poisson);
Dens      = Mat_Prop(3);
Lambda_Vis = Mat_Prop(4);
Mu_Vis    = Mat_Prop(5);

for N = 1:NE

    % ---- Geometric properties
    % Global nodes mumeration
    Node1 = NOC(N,1);
    Node2 = NOC(N,2);
    Node3 = NOC(N,3);
    Node4 = NOC(N,4);

    % Other geometric properties

    DJE0=DJ0(1,N);
    BE0=B0(1:4,1:2,N);

    DJE1=DJ1(1,N);
    BE1=B1(1:4,1:2,N);

    DJE2=DJ2(1,N);
    BE2=B2(1:4,1:2,N);

    DJE3=DJ3(1,N);
    BE3=B3(1:4,1:2,N);

    DJE4=DJ4(1,N);
    BE4=B4(1:4,1:2,N);

    % ----- At time n ...
    % Getting local element unknowns from global unknowns...
    [UE] = Local(UG,Node1,Node2,Node3,Node4);

```

A TWO-STEP TAYLOR-GALERKIN ALGORITHM APPLIED TO LAGRANGIAN DYNAMICS

```

% Calculation of old flux
[FluxE] = Flux_Unk(Young,Poisson,Dens,UE);

% ----- At time n+1/2 ...
% Prediction of new local element unknowns
[dUE] = Unk_Predict(FluxE,BE0,dt);
UE=UE+dUE;

% ----- Error control structure over pressure
F1(1,1) = UE(3);    F1(1,2) = UE(4);    % F Node1
F1(2,1) = UE(5);    F1(2,2) = UE(6);
F2(1,1) = UE(9);    F2(1,2) = UE(10);   % F Node2
F2(2,1) = UE(11);   F2(2,2) = UE(12);
F3(1,1) = UE(15);   F3(1,2) = UE(16);   % F Node3
F3(2,1) = UE(17);   F3(2,2) = UE(18);
F4(1,1) = UE(21);   F4(1,2) = UE(22);   % F Node4
F4(2,1) = UE(23);   F4(2,2) = UE(24);
% Volumetric Stress: Pressure
P1=Kappa*(det(F1)-1);          % PV1
P_Max(Node1) = max(P_Max(Node1),P1);
P_Min(Node1) = min(P_Min(Node1),P1);
P2=Kappa*(det(F2)-1);          % PV2
P_Max(Node2) = max(P_Max(Node2),P2);
P_Min(Node2) = min(P_Min(Node2),P2);
P3=Kappa*(det(F3)-1);          % PV3
P_Max(Node3) = max(P_Max(Node3),P3);
P_Min(Node3) = min(P_Min(Node3),P3);
P4=Kappa*(det(F4)-1);          % PV4
P_Max(Node4) = max(P_Max(Node4),P4);
P_Min(Node4) = min(P_Min(Node4),P4);
% ----- End of error control structure

% Prediction of new local element flux
[FluxE] = Flux_Unk(Young,Poisson,Dens,UE);

% ----- Viscous flux
[FluxE_Vis]=Flux_Vis_Unk(Lambda_Vis,Mu_Vis,Dens,UE,BE0);

% Total flux without viscosity
FluxE=FluxE-FluxE_Vis;

% Getting internal contribution of the element
RHSE=Internal_RHS(FluxE,DJE1,DJE2,DJE3,DJE4,BE1,BE2,BE3,BE4,dt);

% Getting external contribution of the boundary

for side=1:length(NBE)

    if (N==NBE(side))

        NormalE=Normal(1:2,1,side);
        LengthE=Length(1,side);
        N1E=N1(1,side);
        N2E=N2(1,side);

        Start1 = 6*(N1E-1)+1;
        End1    = Start1+5;
        Start2 = 6*(N2E-1)+1;
        End2    = Start2+5;

        FluxN(1:6 ,1) = FluxE(Start1:End1,1:2)*NormalE;          % Node1
        FluxN(1:6 ,1) = Modify_Flux(FluxN(1:6 ,1),BT(side),Time);

        FluxN(7:12,1) = FluxE(Start2:End2,1:2)*NormalE;        % Node2
        FluxN(7:12,1) = Modify_Flux(FluxN(7:12,1),BT(side),Time);

        % Calculation of external contribution of element to RHS
        [External]=External_RHS(FluxN,N1E,N2E,LengthE,dt);
    end
end

```

A TWO-STEP TAYLOR-GALERKIN ALGORITHM APPLIED TO LAGRANGIAN DYNAMICS

```

    RHSE=RHSE+External;

    end; % End of if (N==NBE(side))

    end; % End of for side=1:length(NBE)

% ---- Assembling
    [RHS] = Assemble(RHS,Node1,Node2,Node3,Node4,RHSE);

end; % End of general loop for N = 1:NE

% ---- Modify RHS for Wall conditions (Strong form)
[RHS] = Modify_Wall_BC(RHS,NU);

% ---- Calculate deviation error of volumetric pressure
Error_P = P_Max - P_Min;

% End of function Global_RHS

%----- function Modify_Flux -----
%
function [Flux] = Modify_Flux(Flux,BT,Time)
%
% SPECIFICATION: This function modifies the Flux of the Element
%                 FluxE depending on the Boundary Type BT
%
% BT=      0 : Clamped wall                px=0, py=0
% BT=     -1 : Hinges parallel to X        py=0
% BT=     -2 : Hinges parallel to Y        px=0
% BT=    -10 : Movement parallel to X      py=0, Pn=0
% BT=    -20 : Movement parallel to Y     px=0, Pn=0
% BT=      1 : Free boundary               Pn=0
% BT=     10 : Load                        Traction=Pn

if (BT==0) % px=0, py=0

    Flux(3,1)=0;
    Flux(4,1)=0;
    Flux(5,1)=0;
    Flux(6,1)=0;

elseif (BT==-1) % py=0

    Flux(5,1)=0;
    Flux(6,1)=0;

elseif (BT==-2) % px=0

    Flux(3,1)=0;
    Flux(4,1)=0;

elseif (BT==-10) % py=0 Pn=0

    Flux(1,1)=0;
    Flux(2,1)=0;
    Flux(5,1)=0;
    Flux(6,1)=0;

elseif (BT==-20) % px=0 Pn=0

    Flux(1,1)=0;
    Flux(2,1)=0;
    Flux(3,1)=0;
    Flux(4,1)=0;

```

A TWO-STEP TAYLOR-GALERKIN ALGORITHM APPLIED TO LAGRANGIAN DYNAMICS

```

elseif (BT==1)    % Free condition Pn=0

    Flux(1,1)=0;
    Flux(2,1)=0;

elseif (BT==10)  % Load condition  Pn=traction

    Flux(1,1)=-1*Traction(Time,1);
    Flux(2,1)=-1*Traction(Time,2);

end;

% End of function Modify_Flux

%----- function Internal_RHS -----
function [RHSE]=Internal_RHS(FluxE,DJ1,DJ2,DJ3,DJ4,B1,B2,B3,B4,dt);
%
% SPECIFICATION: This function serves to calculate the contribution
%                 to RHS of each element considering the internal
%                 flux and external loss through the boundary.
%
%
Flux1 = FluxE( 1:6,  1:2);
Flux2 = FluxE( 7:12, 1:2);
Flux3 = FluxE(13:18, 1:2);
Flux4 = FluxE(19:24, 1:2);

% Average_Flux0 = zeros(6,2);
% Average_Flux0 = (Flux1+Flux2+Flux3+Flux4)/4;

c1=0.25*(1.0+(1.0/sqrt(3.0)))*(1.0+(1.0/sqrt(3.0)));
c2=0.25*(1.0+(1.0/sqrt(3.0)))*(1.0-(1.0/sqrt(3.0)));
c3=0.25*(1.0-(1.0/sqrt(3.0)))*(1.0-(1.0/sqrt(3.0)));
c4=c2;

Average_Flux1 = zeros(6,2);
Average_Flux1 = c1*Flux1+c2*Flux2+c3*Flux3+c4*Flux4;

Average_Flux2 = zeros(6,2);
Average_Flux2 = c4*Flux1+c1*Flux2+c2*Flux3+c3*Flux4;

Average_Flux3 = zeros(6,2);
Average_Flux3 = c3*Flux1+c4*Flux2+c1*Flux3+c2*Flux4;

Average_Flux4 = zeros(6,2);
Average_Flux4 = c2*Flux1+c3*Flux2+c4*Flux3+c1*Flux4;

Int_Rhs=zeros(24,1);

% Int_Rhs( 1:6 ,1) = dt*(4*DJ0)*(Average_Flux0(1:6,1)*B0(1,1)+...
%                               Average_Flux0(1:6,2)*B0(1,2)); % Node 1

Int_Rhs( 1:6 ,1) = dt*(DJ1)*(Average_Flux1(1:6,1)*B1(1,1)+ ...
                             Average_Flux1(1:6,2)*B1(1,2))+...
dt*(DJ2)*(Average_Flux2(1:6,1)*B2(1,1)+ ...
           Average_Flux2(1:6,2)*B2(1,2))+...
dt*(DJ3)*(Average_Flux3(1:6,1)*B3(1,1)+ ...
           Average_Flux3(1:6,2)*B3(1,2))+...
dt*(DJ4)*(Average_Flux4(1:6,1)*B4(1,1)+ ...
           Average_Flux4(1:6,2)*B4(1,2)); % Node 1

% Int_Rhs( 7:12,1) = dt*(4*DJ0)*(Average_Flux0(1:6,1)*B0(2,1)+...
%                               Average_Flux0(1:6,2)*B0(2,2)); % Node 2

Int_Rhs( 7:12,1) = dt*(DJ1)*(Average_Flux1(1:6,1)*B1(2,1)+ ...

```

A TWO-STEP TAYLOR-GALERKIN ALGORITHM APPLIED TO LAGRANGIAN DYNAMICS

```

        Average_Flux1(1:6,2)*B1(2,2))+...
dt*(DJ2)*(Average_Flux2(1:6,1)*B2(2,1)+ ...
        Average_Flux2(1:6,2)*B2(2,2))+...
dt*(DJ3)*(Average_Flux3(1:6,1)*B3(2,1)+ ...
        Average_Flux3(1:6,2)*B3(2,2))+...
dt*(DJ4)*(Average_Flux4(1:6,1)*B4(2,1)+ ...
        Average_Flux4(1:6,2)*B4(2,2));           % Node 2

% Int_Rhs(13:18,1) = dt*(4*DJO)*(Average_Flux0(1:6,1)*B0(3,1)+...
%                               Average_Flux0(1:6,2)*B0(3,2)); % Node 3

Int_Rhs(13:18,1) = dt*(DJ1)*(Average_Flux1(1:6,1)*B1(3,1)+ ...
        Average_Flux1(1:6,2)*B1(3,2))+...
dt*(DJ2)*(Average_Flux2(1:6,1)*B2(3,1)+ ...
        Average_Flux2(1:6,2)*B2(3,2))+...
dt*(DJ3)*(Average_Flux3(1:6,1)*B3(3,1)+ ...
        Average_Flux3(1:6,2)*B3(3,2))+...
dt*(DJ4)*(Average_Flux4(1:6,1)*B4(3,1)+ ...
        Average_Flux4(1:6,2)*B4(3,2));           % Node 3

% Int_Rhs(19:24,1) = dt*(4*DJO)*(Average_Flux0(1:6,1)*B0(4,1)+...
%                               Average_Flux0(1:6,2)*B0(4,2)); % Node 4

Int_Rhs(19:24,1) = dt*(DJ1)*(Average_Flux1(1:6,1)*B1(4,1)+ ...
        Average_Flux1(1:6,2)*B1(4,2))+...
dt*(DJ2)*(Average_Flux2(1:6,1)*B2(4,1)+ ...
        Average_Flux2(1:6,2)*B2(4,2))+...
dt*(DJ3)*(Average_Flux3(1:6,1)*B3(4,1)+ ...
        Average_Flux3(1:6,2)*B3(4,2))+...
dt*(DJ4)*(Average_Flux4(1:6,1)*B4(4,1)+ ...
        Average_Flux4(1:6,2)*B4(4,2));           % Node 4

RHSE=Int_Rhs;

% End of function Internal_RHS

%----- function External_RHS -----

function [External] = External_RHS(FluxN,N1,N2,Length,dt);
%
% SPECIFICATION: This function serves to calculate the contribution
%                 to RHS of each element considering the external
%                 loss through the boundary N1-N2
%
if (N1==N2)
    External=zeros(24,1);
else
    % Calculation of starting d.o.f. for N1 and N2

    External = zeros(24,1);

    Start1 = 6*(N1-1)+1;
    End1   = Start1+5;

    Start2 = 6*(N2-1)+1;
    End2   = Start2+5;

    Flux1 = FluxN(1:6 ,1);
    Flux2 = FluxN(7:12,1);

    External(Start1:End1,1) = -0.5*dt*((2*Flux1+Flux2)/3)*Length;
    External(Start2:End2,1) = -0.5*dt*((2*Flux2+Flux1)/3)*Length;

```

```

end;

% End of function External_RHS

%----- function Local -----
%
function [UE]=Local(UG,Node1,Node2,Node3,Node4);
%
% SPECIFICATION: This function serves to obtain
%                 the local unknowns from global coordinates
%
% UG: Global vector
% UE: Local vector for the element

    UE=zeros(24,1);

    % Node 1
    Start1 = 6*(Node1-1)+1;
    End1   = Start1+5;
    UE(1:6,1) = UG(Start1:End1,1);

    % Node 2
    Start2 = 6*(Node2-1)+1;
    End2   = Start2+5;
    UE(7:12,1) = UG(Start2:End2,1);

    % Node 3
    Start3 = 6*(Node3-1)+1;
    End3   = Start3+5;
    UE(13:18,1) = UG(Start3:End3,1);

    % Node 4
    Start4 = 6*(Node4-1)+1;
    End4   = Start4+5;
    UE(19:24,1) = UG(Start4:End4,1);

% End of function Local

%----- function Modify_Wall_BC -----
function [RHS]=Modify_Wall_BC(RHS,NU);

% SPECIFICATION: This function serves to modify global RHS vector
%                 for the Wall Boundary Conditions in the nodes
%                 given by NU

for I = 1:length(NU)
    N = NU(I);
    RHS(N) = 0;
end;

% End of function Modify_Wall_BC

%----- function Assemble -----

function [RHS]=Assemble(RHS,Node1,Node2,Node3,Node4,RHSE)
%
% SPECIFICATION: This function adds the contribution
%                 of elemental RHS to the global RHS
%
%
% Node 1
    Start1 = 6*(Node1-1)+1;
    End1   = Start1+5;
    RHS(Start1:End1,1) = RHS(Start1:End1,1)+RHSE( 1:6 ,1);

```

## A TWO-STEP TAYLOR-GALERKIN ALGORITHM APPLIED TO LAGRANGIAN DYNAMICS

```
% Node 2
Start2 = 6*(Node2-1)+1;
End2   = Start2+5;
RHS(Start2:End2,1) = RHS(Start2:End2,1)+RHSE( 7:12,1);

% Node 3
Start3 = 6*(Node3-1)+1;
End3   = Start3+5;
RHS(Start3:End3,1) = RHS(Start3:End3,1)+RHSE(13:18,1);

% Node 4
Start4 = 6*(Node4-1)+1;
End4   = Start4+5;
RHS(Start4:End4,1) = RHS(Start4:End4,1)+RHSE(19:24,1);

% End of function Assemble
```