UNIVERSITÀ
DI TRENTO

University of Trento

Department of Civil, Environmental and Mechanical Engineering

Computational mechanics II: end course project

Development and testing of FEM software for planar elastic frames and trusses analysis in large displacement conditions

Author:
Francesco Profico

Supervisor:
Prof. Andrea Piccolroaz

March 12, 2022

Contents

1 Introduction

In the present work a structural analysis program is developed, tested and results are illustrated on multiple case studies. The program is coded in Python and is capable of performing nonlinear analyses on two-dimensional planar structures involving geometric nonlinearities. The load-displacement structural response can be derived with the program under concentrated load solicitation and the nonlinear equilibrium path can be derived. The implementation of geometrically nonlinear truss element and Timoshenko beam-like two-node elements are presented. The formulations are analytically derived in a Total-Lagrangian formulation approach (TL). Furthermore a Newton-Rhapson resolution method is implemented in order to solve numerically the nonlinear equilibrium equations. The routine consists in a predictor-correction procedure. In order to correctly follow the load-displacement nonlinear paths, different methods have been developed. Both the load and displacement control methods are implemented. These fail following the equilibrium path in particular circumstances and can therefore present drawbacks. Therefore a Riks arclength path following method has been also developed. This method is particularly adapt to capture both snap through phenomena and snap-back behaviours in the structural response.

The developed routine is firstly tested by comparison of the load-displacements curve with the correspondents derived from literature benchmark problems or analytically derived solutions. Following the validation a variety of case studies are presented and discussed.

A deeper look into the coding structure and functioning is given in Annex A. Here the coding lines are shown and the analytically defined vectors, matrices and procedures are recognisable in the coding python language.

The main achievements and scopes of the present study will be:

- To successfully implement a programming code capable of solving planar structural geometric nonlinear problem involving frame and trusses;
- learn to code in Python, in particular, aside from the computational mechanics aspects, the program interface (Fig.1), user input window, internal structure and output tools have been a crucial part of the work as well;
- to better understand geometric nonlinear phenomena such as hardening and softening behaviours, snap-through and snap back occurence on multiple case studies;
- to better understand the widely exploited Newton Rhapson algorithm as root finder process;
- to understand how control strategies such as the load control, displacement control and arclength control methods work.

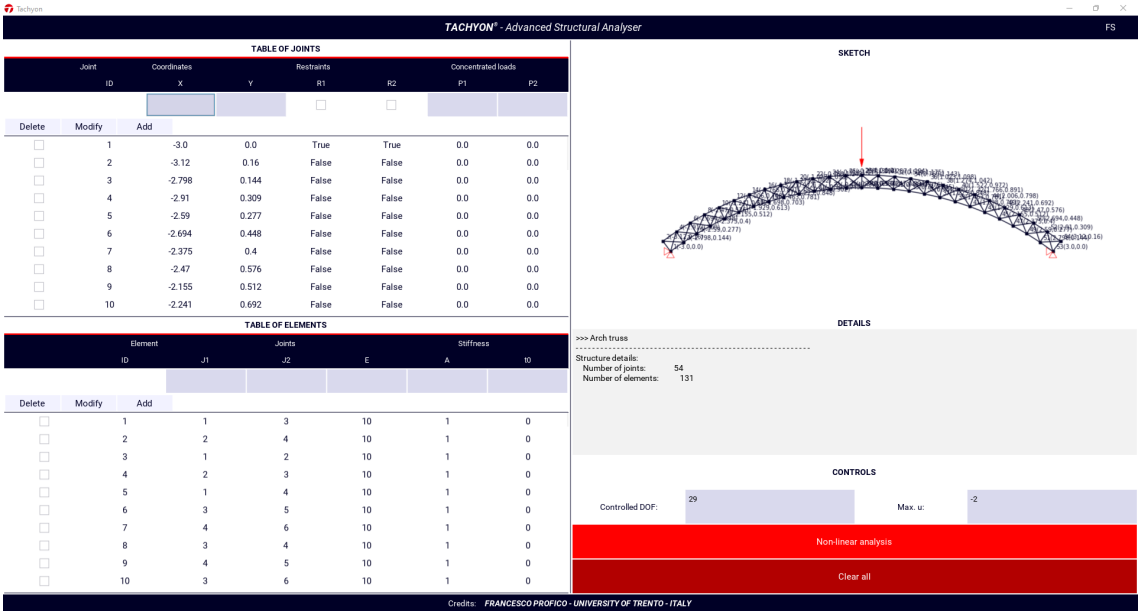

Figure 1: user interface of the developed python FEM program

2 General considerations

2.1 Geometric non-linearity

Sources of nonlinearities in mechanical behaviours can derive from:

- material law nonlinearity;
- geometric nonlinearity;
- contact problems (boundary conditions nonlinearity);
- load condition nonlinearity.

How various types of nonlinearities act in a mechanical problem is well depicted in the "Tonti diagram" (Fig. 2 and Fig.3).

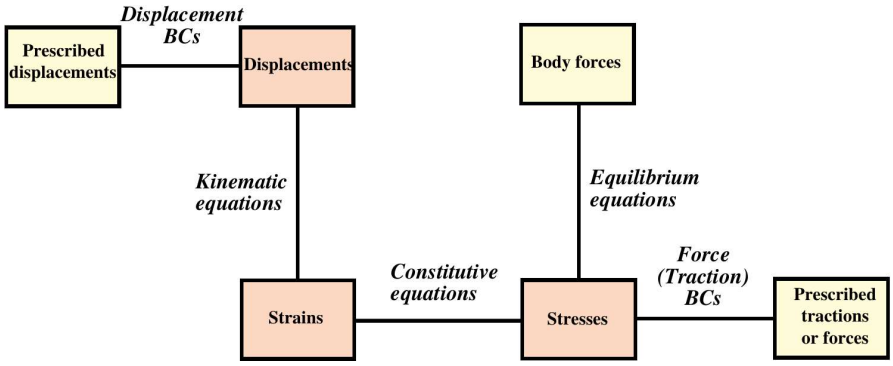

Figure 2: relevant relationships in mechanical problems: overview

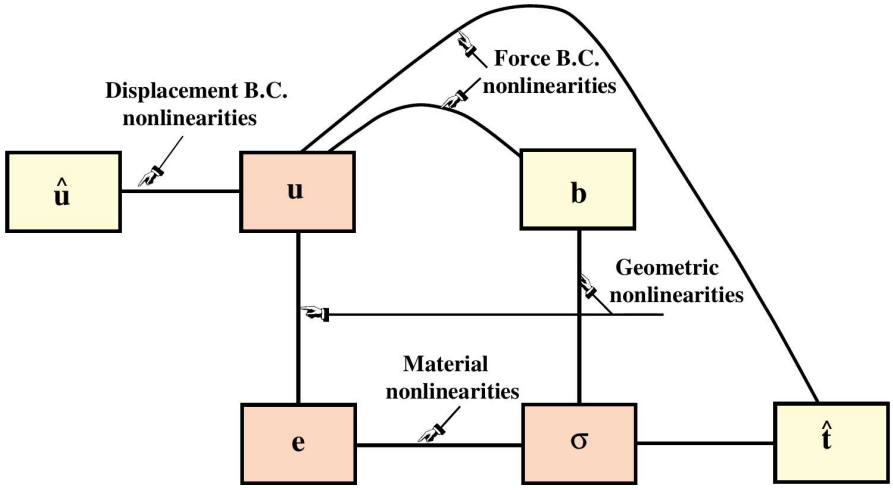

Figure 3: relevant relationships in mechanical problems: nonlinearity sources

The present work is focused on the study of the geometric nonlinearity and the main scope is to analyse this phenomena on different case studies with the implemented program. Other types of nonlinearities coming from material laws, boundary conditions or originated from load conditions are not considered. Practical application purposes in which taking into account for geometric nonlinearity is crucial are multiple. Some examples are here listed:

- TAVR (Transcatheter Aortic Valve Replacement) valves in biomedical applications (Fig.**??** from [10]).
- snap through behaviours of low curvature shells and arches.
- second order and postbuckling response in structural civil engineering frames. Taking into account second order effects is recommended by the international standard design codes like Eurocodes.

- airspace engineering: many commercial aircraft are designed so that fuselage skins can elastically buckle (Fig. 5, and Fig.6 from [4]) below limit load and continue to operate safely and efficiently
- the postbuckling favourable contribute shear buckling of steel beam webs (Fig.7 from [**?**]) is in bridge engineering checks taken into account according to Eurocode 3 part 1-5 [1].

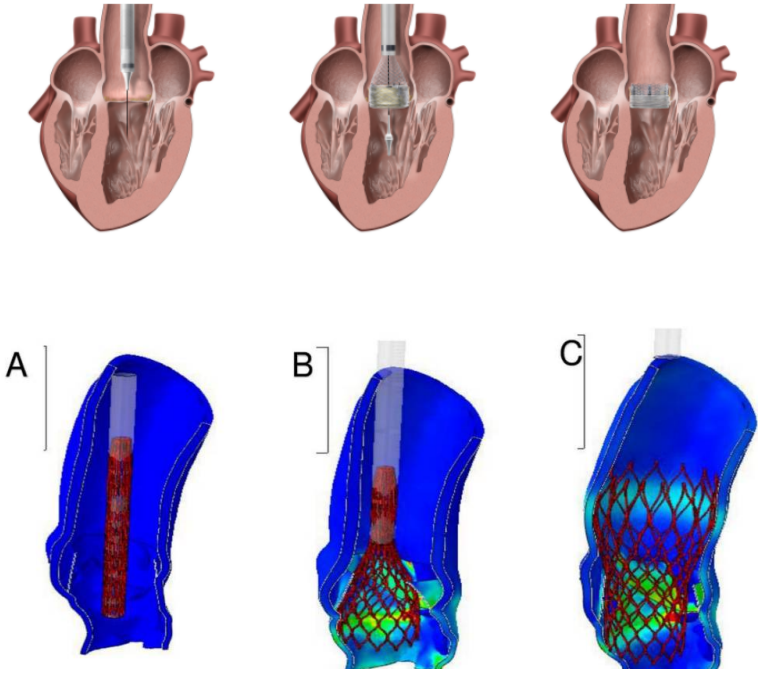

Figure 4: TAVR valve functioning [10]

Figure 5: buckling of fuselage in airspace engineering

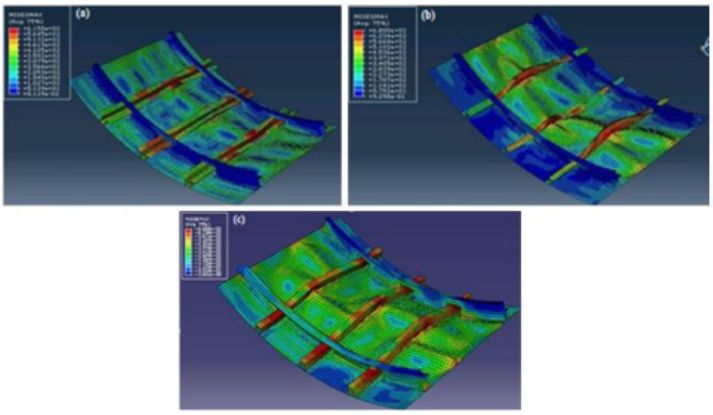

Figure 6: buckling of fuselage: finite element analysis (FEA) from [4]

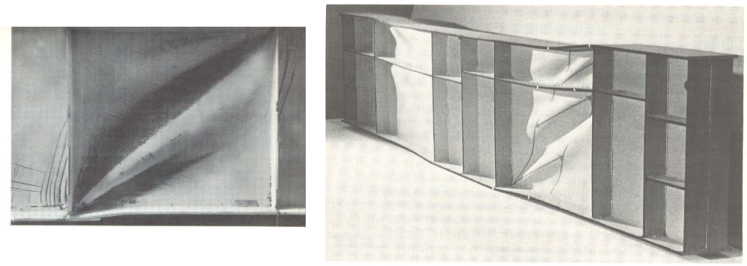

Figure 7: shear buckling of steel beam web from

2.2 Equilibrium path

In the present work no account is taken for the dynamical effects. The forces are assumed to be conservative. The load is assumed to be slowly applied in order that the successive equilibrium conditions which are derived reflect a static configuration under a prescribed load condition of the structure. The successive set of points are referred as "equilibrim path". Points belonging to the equilibrium path do satisfy the relation:

$$f_{int} = f_{ext} \qquad \text{Equilibrium condition} \tag{1}$$

This must be satisfied for every part of the structure. The condition may be expressed also as the stationarity of the Total Potential Energy. the equilibrium condition is therefore strictly related with the Potential Energy, which is a scalar quantity, and is satisfied when the gradient of the potential energy function is a zero vector.

$$\frac{\partial \Pi}{\partial \boldsymbol{u}} = 0 \qquad \text{Equilibrium condition} \tag{2}$$

In a FEM discretized approach $\boldsymbol{f}_{int}$ and $\boldsymbol{f}_{ext}$ are two vectors, which components represent the nodal internal and external forces respectively. In the $(\boldsymbol{u}, \Lambda)$ space, the set of points satisfying the equilibrium equation belong to the equilibrium path. The equilibrium path can be interpreted as the load-displacement relation of the structure and reflects the structural response under static conditions.

In nonlinear problem the equilibrium condition is no more derivable via an effects linear addition. Moreover the Kirchhoff uniqueness theorem is no more valid and more than one equilibrium configurations can exist under the same load. When a nonlinear problem is considered, the structural response and the associated load-deformation behaviour can exhibit critical points, softening behaviours, turning points and bifurcation phenomena. Moreover snap through or snap back can occur.

In general large deformation problems the notion of current configuration and reference configurations are introduced (Fig.8). In this work a Total Lagrangian description of the motion will be adopted. Thus the reference configuration will always be the the undeformed one.

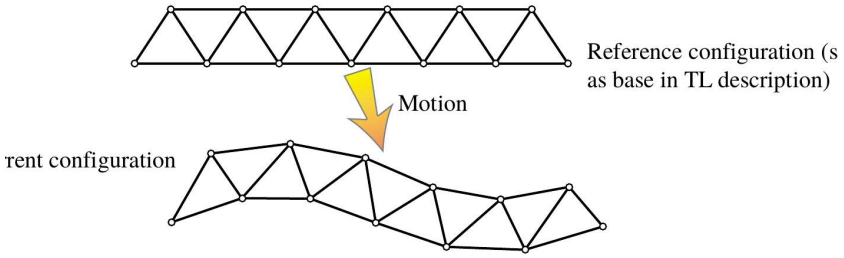

Figure 8: reference and current configuration

2.3 Discretization and FEM approach

In a FEM approach the structure is discretized. Thus, the infinite degrees of freedom describing the continua structure are reduced to a relevant finite number of degrees of freedom. In order to describe the displacement and strain field in the intermediate parts of the body, interpolating shape functions are introduced and the displacement field is expressed as function of the finite number of degrees of freedom. These relevant displacements, collected in the $\boldsymbol{u}$ vector, constitute the finite number of degrees of freedom of the discretized FEM structure. Moreover the generic structure is in a FEM approach reduced into elements, the elements contribution in terms of internal forces and stiffness are assembled in a global internal force vector and a global stiffness matrix of the structure.

In order to obtain the single elements contributions to the global internal force vector and tangential stiffness matrix, the element's formulation has to be developed. The first step in the subsequent passages will then be the elements formulation. In this work two geometrically nonlinear elements are developed. The first is a truss type element and the second is a beam type element. Both will be described in a Total Lagrangian approach.

2.4 Numerical solution

When having a global tangent stiffness matrix of the structure and the internal forces vector both dependent on the current configuration state, a step-by-step procedure of the equilibrium path derivation should be carried out. In order to do that, a step incrementation strategy should be implemented. Different control methods exist. In this work a load control, a displacement control and a Riks arclength control strategy will be implemented.

As last problem, in the resolving procedure, if a non purely incremental analysis is carried out, correction steps have to be performed in finding the solution to the nonlinear equilibrium equation after a first predictor step. In order to do that a Newton-Rhapson method is in the present work described and implemented in the program.

3 Elements formulations

3.1 Truss element in Total Lagrangian description

A Total Lagrangian formulation of a geometrically nonlinear truss element is here described. Reference is made to [2, 3].

3.1.1 Displacements field description

In the reference configuration:

$$A_0 \qquad \text{Cross sectional area in reference configuration} \tag{3}$$

$$L_0 \qquad \text{Element length in reference configuration} \tag{4}$$

In the current configuration:

$$A \qquad \text{Cross sectional area in current configuration} \tag{5}$$

$$L \qquad \text{Element length in current configuration} \tag{6}$$

The element's degree of freedoms are collected in the displacement vector $\boldsymbol{u}$. The nodal forces are collected in the vector $\boldsymbol{f}$.

$$\boldsymbol{u} = \begin{pmatrix} u_{X1} \\ u_{Y1} \\ u_{X2} \\ u_{Y2} \end{pmatrix} = \begin{pmatrix} x_1 - X_1 \\ y_1 - Y_1 \\ x_2 - X_2 \\ y_2 - Y_2 \end{pmatrix} \qquad \boldsymbol{f} = \begin{pmatrix} f_{X1} \\ f_{Y1} \\ f_{X2} \\ f_{Y2} \end{pmatrix} \tag{7}$$

The displacement field between the nodes is obtained with linear interpolation functions:

$$N_1(\xi) = \frac{1}{2}(1 - \xi) \tag{8}$$

$$N_2(\xi) = \frac{1}{2}(1 + \xi) \tag{9}$$

The generic point has coordinates $\boldsymbol{X}$ in the reference configuration:

$$\boldsymbol{X}(\xi) = \begin{pmatrix} X(\xi) \\ Y(\xi) \end{pmatrix} = \begin{pmatrix} N_1(\xi)X_1 + N_2(\xi)X_2 \\ N_1(\xi)Y_1 + N_2(\xi)Y_2 \end{pmatrix} \tag{10}$$

$$\boldsymbol{x}(\xi) = \begin{pmatrix} x(\xi) \\ y(\xi) \end{pmatrix} = \begin{pmatrix} N_1(\xi)x_1 + N_2(\xi)x_2 \\ N_1(\xi)y_1 + N_2(\xi)y_2 \end{pmatrix} \tag{11}$$

Here ξ is the isoparametric master element coordinate in the range $[-1, 1]$. The displacement field is expressed as:

$$\boldsymbol{u}(\xi) = \boldsymbol{x}(\xi) - \boldsymbol{X}(\xi) = \begin{pmatrix} u_X(\xi) \\ u_Y(\xi) \end{pmatrix} = \begin{pmatrix} N_1(\xi)u_{X1} + N_2(\xi)u_{X2} \\ N_1(\xi)u_{Y1} + N_2(\xi)u_{Y2} \end{pmatrix} \tag{12}$$

$$\boldsymbol{u}(\xi) = \begin{pmatrix} N_1(\xi) & 0 & N_2(\xi) & 0 \\ 0 & N_1(\xi) & 0 & N_2(\xi) \end{pmatrix} = \begin{pmatrix} u_{X1} \\ u_{X2} \\ u_{Y1} \\ u_{Y2} \end{pmatrix} = \boldsymbol{N}(\xi)\boldsymbol{u} \tag{13}$$

The element's lengths in current and reference configurations can be computed as:

$$L^2 = (X_{21} + u_{X21})^2 + (Y_{21} + u_{Y21})^2 \qquad L_0^2 = X_{21}^2 + Y_{21}^2 \tag{14}$$

3.1.2 Strain-displacements relation

The Green-Lagrange strain is:

$$e = \frac{L^2 - L_0^2}{2L_0^2} = \frac{1}{L_0}(c_{0X}u_{X21} + c_{0Y}u_{Y21}) + \frac{1}{2L_0^2}(u_{X21}^2 + u_{Y21}^2) \tag{15}$$

Here:

$$c_{0X} = \frac{X_{21}}{L_0} = \cos\psi_0 \tag{16}$$

$$c_{0Y} = \frac{Y_{21}}{L_0} = \sin\psi_0 \tag{17}$$

$$c_X = \frac{x_{21}}{L_0} = \cos\psi \tag{18}$$

$$c_{0X} = \frac{y_{21}}{L_0} = \sin\psi \tag{19}$$

The GL strain can be expressed in a matricial form:

$$e = \boldsymbol{B}_0\boldsymbol{u} + \frac{1}{2}\boldsymbol{u}^T\boldsymbol{M}\boldsymbol{u} = e_L + e_N \tag{20}$$

Here e_N is a nonlinear contribute and e_L is a linear contribute. Following matrices have been defined:

$$\boldsymbol{B}_0 = \frac{1}{L_0}\{-c_{0X}, -c_{0Y}, c_{0X}, c_{0Y}\} \tag{21}$$

$$\boldsymbol{M} = \frac{1}{L_0^2}\begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix} \tag{22}$$

$$\boldsymbol{B} = \frac{1}{L_0}\begin{bmatrix} -c_x & -c_y & c_x & c_y \end{bmatrix} = \boldsymbol{B}_0 + \boldsymbol{u}^T\boldsymbol{M} \tag{23}$$

3.1.3 Total potential energy

In the current configuration the total potential energy Π is:

$$\Pi = U - W = \int_{V_0} t_0 e + \frac{1}{2} E e^2 \, dV_0 - \boldsymbol{f}^T \boldsymbol{u} = \int_{L_0} A_0 (t_0 e + \frac{1}{2} E e^2) \, d\overline{X} - \boldsymbol{f}^T \boldsymbol{u} \tag{24}$$

In this last relation the integrand is constant. Therefore:

$$\Pi = U - W = A_0 L_0 \left(t_0 e + \frac{1}{2} E e^2 \right) - \boldsymbol{f}^T \boldsymbol{u} \tag{25}$$

3.1.4 Internal forces vector

The residual forces vector is:

$$\boldsymbol{r} = \frac{\partial \, \Pi}{\partial \boldsymbol{u}} = V_0 t \frac{\partial \, e}{\partial \boldsymbol{u}} - \boldsymbol{f} \tag{26}$$

The internal forces vector here consists in:

$$\boldsymbol{f}_{int,e} = \frac{\partial \, U}{\partial \boldsymbol{u}} = V_0 t \frac{\partial \, e}{\partial \boldsymbol{u}} \tag{27}$$

Where:

$$\frac{\partial \, e}{\partial \boldsymbol{u}} = \boldsymbol{B}^T \tag{28}$$

The code implementation can be appreciated in Appendix.A.

3.1.5 Tangent stiffness matrix

The tangent stiffness matrix is computed taking the first variation of the internal forces vector:

$$\boldsymbol{K} = \frac{\partial \, \boldsymbol{r}}{\partial \boldsymbol{u}} = \frac{\partial \, \boldsymbol{f_{int}}}{\partial \boldsymbol{u}} = \frac{\partial}{\partial \boldsymbol{u}} \left(V_0 t \frac{\partial e}{\partial \boldsymbol{u}} \right) = \boldsymbol{K}_M + \boldsymbol{K}_G \tag{29}$$

Here:

$$\boldsymbol{K}_L = \frac{E A_0}{L_0} \begin{bmatrix} c^2 & c\,s & -c^2 & -c\,s \\ c\,s & s^2 & -c\,s & -s^2 \\ -c^2 & -c\,s & c^2 & c\,s \\ -c\,s & -s^2 & c\,s & s^2 \end{bmatrix} \tag{30}$$

$$\boldsymbol{K}_G = \frac{F}{L_0} \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix} \tag{31}$$

The tangent stiffness matrix code implementation can be appreciated in Appendix.A.

3.2 Beam Timoshenko element in Total-Lagrangian description

A geometrically nonlinear beam element is formulated in this section. Reference is done to [5]. This is done under the assumption of small strains that implies linear-elastic behaviour of the material. The assumptions of straight element and prismatic beam are considered. The cross-section is therefore uniform along the longitudinal axis of the element. The element has two nodes and is described in a Total Lagrangian (TL) kinematics.

3.2.1 Introduction

In order to introduce the TL description, a current and a reference configuration have to be introduced. In the reference configuration the X and Y coordinates are introduced. The element has two nodes. The cross-section rotation from reference to current configuration is called θ.

In Euler-Bernoulli Beam model (Fig.9) no strain energy due to shear stresses is taken into account. While the longitudinal axis of the element will deform, sections will remain plane and normal to the longitudinal axis. Here $\gamma = \theta - \psi = 0$ (Fig.10) holds.

In the Timoshenko model (Fig.9) shear deformation effects are included. The cross section remains plane but it does not remain orthogonal to the deformed longitudinal axis. The transverse shear stress is assumed to be constant over the cross section. The shear distortion can be computed as $\gamma = \theta - \psi$ (Fig.10).

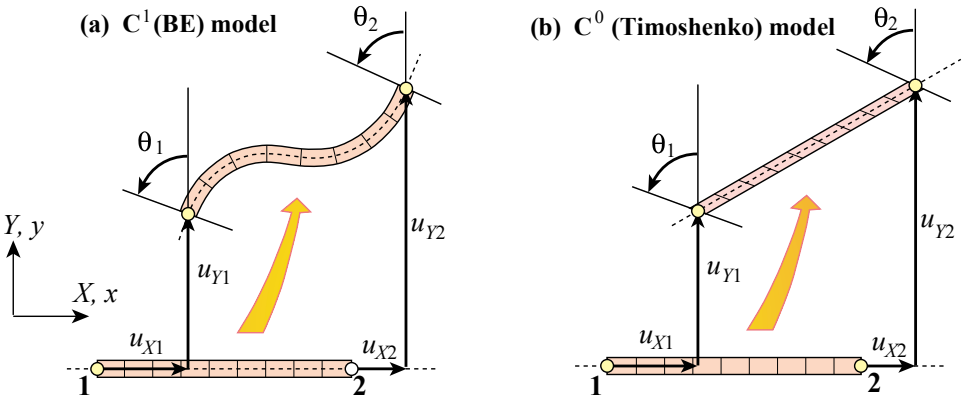

Figure 9: Comparison of Euler-Bernoulli and Timoshenko beam models

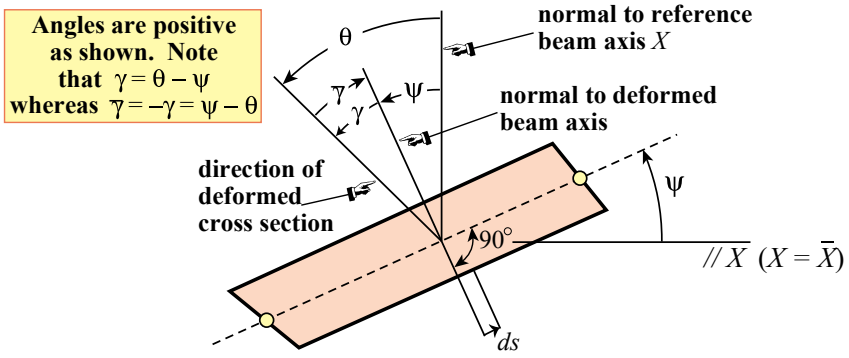

Figure 10: Sectional rotation in relation with element longitudinal axis

Despite of the fact that in Timoshenko models the inclusion of the shear deformation appears to complicate the formulation, this kind of element is simpler to be constructed. This is because $\theta(X)$ can be described indipendently from the $u(X)$ and $v(X)$ fields. Thus a linear variation can be assumed to map both the rotational field and the displacement field. The linear transverse variation matches the

commonly assumed for the axial deformation like in the previous done description of the truss element. The transverse and axial displacements are called consistent. In the Euler-Bernoulli element a cubic interpolation of the displacement field needs to be exploited. This element is often called Hermitian because from the cubic shape functions used. While in the Euler-Bernoulli formulation at least C^1 functions are needed, in Timoshenko the requirement drops to C^0 functions. The internal kinematics of the Timoshenko model is therefore simpler.

An important aspect that needs to be taken into account in the Timoshenko formulation is the shear locking. This occurs because of the high shear strain energy that would result in an exact integration with the actual shear properties in order to match the element actual deformation with the assumed linear field of displacements. To avoid locking some corrections are needed. A selective integration is done for the shear energy and a residual energy balancing is taken into account. Most of the shear energy is therefore removed. In this model the modeling of the actual shear deformation is not the main focus. The actual scope is to capture the correct beam behaviour. Authors underline that it would be more appropriate to denote the element as "$C^0 element$" rather than "Timoshenko element".

A geometrically nonlinear Timoshenko beam is formulated. In order to formulate the element a strain measure needs to be exploited by mean of displacement gradients. Work-conjugate stresses respect to the strain measures are introduced. The final objective is to derive the expression of the tangent stiffness matrix and internal forces vector as function of the nodal degrees of freedom of the elements.

The following passages that will be carried out are summarized in the flowchart (Fig.11).

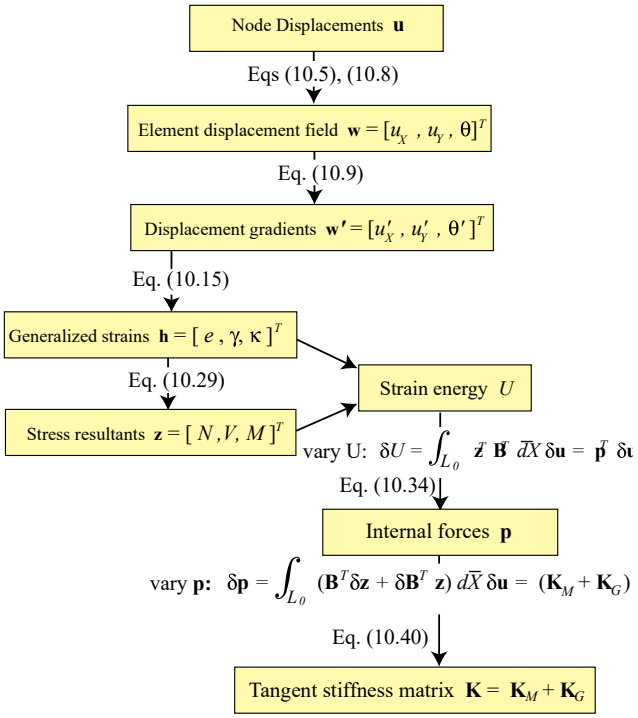

Figure 11: Derivation of tangent stiffness matrix and internal nodal forces vector: general scheme

3.2.2 Displacements field description

Each node (i) has three degrees of freedom which are two translational DOFs (u_i and v_i) and one rotational DOF θ_i. In order to obtain the element formulation, the reference configuration is considered aligned with the X axis with origin at node 1. The element has length L_0 and cross sectional area A_0 and second moment I_0. These are defined as:

$$A_0 = \int_{A_0} dA \qquad I_0 = \int_{A_0} Y^2 dA. \tag{32}$$

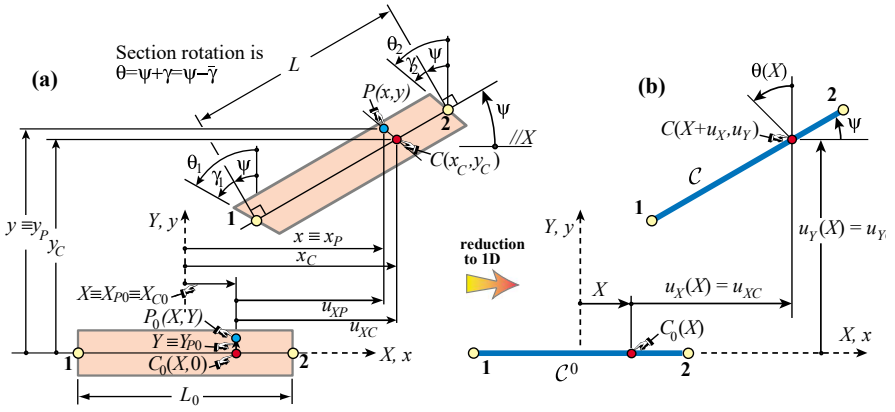

Figure 12: Displacements mapping

In the current configuration the area is A, the second moment is I and the element's length is L. The six degrees of freedom of the element are collected in a single vector $\boldsymbol{u}$:

$$\boldsymbol{u} = \{u_1, v_1, \theta_1, u_2, v_2, \theta_2\}^T \tag{33}$$

The nodal force vector components are collected in the vector $\boldsymbol{f}$:

$$\boldsymbol{f} = \{f_{X1}, f_{Y1}, f_{\theta 1}, f_{X2}, f_{Y2}, f_{\theta 2}\}^T \tag{34}$$

The internal motion is described by:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_C - Y(\sin\psi + \sin\gamma\cos\psi) \\ y_C + Y(\cos\psi - \sin\gamma\sin\psi) \end{pmatrix} = \begin{pmatrix} x_C - Y(\sin\theta + (1-\cos\gamma)\sin\psi) \\ y_C + Y(\cos\theta + (1-\cos\gamma)\cos\psi) \end{pmatrix} \tag{35}$$

with:

$$\theta = \gamma + \psi \qquad x = X + u \qquad y = v \tag{36}$$

The motion description can be expanded into Taylor series in γ up to $O(\gamma^4)$.

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} u + X + 1/2Y\gamma^3\cos\theta - Y(1 + 1/2\cdot\gamma^2 + 7/24\gamma^4)\sin\theta \\ v + Y(1 + 1/2\gamma^2 - 7/24\gamma^4)\cos\theta + 1/2Y\gamma^3\sin\theta \end{pmatrix} \tag{37}$$

Under the assumption of small shear strains $\theta \to 0$:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} X + u - Y\sin\theta \\ v + Y\cos\theta \end{pmatrix} \tag{38}$$

Note that $u(X)$ and $y(X)$ are functions of X only. An internal extended displacement vector $\boldsymbol{w}$ can be defined as:

$$\boldsymbol{w} = \{u(X), v(X), \theta(X)\}^T \qquad \boldsymbol{w}' = \frac{d\boldsymbol{w}}{dX} = \{\frac{d\,u}{dX}, \frac{d\,v}{dX}, \frac{d\,\theta}{dX}\}^T = \{u', v', \theta'\}^T \tag{39}$$

An arclength coordinate s can be introduced in the current configuration. With this coordinate the following useful relations hold:

$$1 + u' = s'\cos\psi \qquad v' = s'\sin\psi \qquad s' = \frac{d\,s}{dX} = \sqrt{(1+u')^2 + (v')^2} \tag{40}$$

If u', v' are constant over X:

$$s' = L/L_0 \qquad 1 + u' = L\cos\psi/L_0 \qquad v' = L\sin\psi/L_0 \tag{41}$$

A linear interpolation of the generalized displacements is used in order to describe the displacement field. Here $\xi = 2X/L_0$ is the isoparametric coordinate ranging form -1 to 1:

$$\boldsymbol{w} = \begin{pmatrix} u(X) \\ v(X) \\ \theta(X) \end{pmatrix} = 1/2 \cdot \begin{pmatrix} 1-\xi & 0 & 0 & 1+\xi & 0 & 0 \\ 0 & 1-\xi & 0 & 0 & 1+\xi & 0 \\ 0 & 0 & 1-\xi & 0 & 0 & 1+\xi \end{pmatrix} \begin{pmatrix} u_1 \\ v_1 \\ \theta_1 \\ u_2 \\ v_2 \\ \theta_2 \end{pmatrix} = \boldsymbol{N}\boldsymbol{u} \tag{42}$$

The displacement gradient interpolation can be expressed as:

$$\boldsymbol{w}' \begin{pmatrix} u'(X) \\ v'(X) \\ \theta'(X) \end{pmatrix} = 1/L_0 \cdot \begin{pmatrix} -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_1 \\ v_1 \\ \theta_1 \\ u_2 \\ v_2 \\ \theta_2 \end{pmatrix} = \boldsymbol{N}'\boldsymbol{u} \tag{43}$$

3.2.3 Strain-displacements relation

The strain-displacement relation has to be introduced. In order to do that the deformation gradient matrix is introduced:

$$\boldsymbol{F} = \begin{pmatrix} \frac{\partial x}{\partial X} & \frac{\partial x}{\partial Y} \\ \frac{\partial y}{\partial X} & \frac{\partial y}{\partial Y} \end{pmatrix} = \begin{pmatrix} 1 + u' - Y\chi\cos\theta & -\sin\theta \\ v' - Y\chi\sin\theta & \cos\theta \end{pmatrix} \tag{44}$$

Here $\chi = \theta'$ is the curvature. The displacement matrix is:

$$\boldsymbol{G} = \boldsymbol{F} - \boldsymbol{I} \tag{45}$$

the Green-Lagrange (GL) strain tensor is computed as:

$$\underline{\boldsymbol{e}} = \begin{pmatrix} e_{XX} & e_{XY} \\ e_{YX} & e_{YY} \end{pmatrix} = 1/2(\boldsymbol{F}^T\boldsymbol{F} - \boldsymbol{I}) = 1/2(\boldsymbol{G} + \boldsymbol{G}^T) \tag{46}$$

The non-zero axial and shear strains are collected in the strain vector:

$$\boldsymbol{e} = \begin{pmatrix} e_1 \\ e_2 \end{pmatrix} = \begin{pmatrix} e_{XX} \\ 2e_{XY} \end{pmatrix} = \begin{pmatrix} (1+u')\cos\theta + v'\sin\theta - Y\theta' - 1 \\ -(1+u')\sin\theta + v'\cos\theta \end{pmatrix} = \begin{pmatrix} e - Y\chi \\ \gamma \end{pmatrix} \tag{47}$$

Therefore, the strain quantities are:

$$e = (1+u')\cos\theta + v'\sin\theta - 1 \qquad \gamma = -(1+u')\sin\theta + v'\cos\theta \qquad \chi = \theta' \tag{48}$$

These are arranged in the vector $\boldsymbol{h}$:

$$\boldsymbol{h} = \{e, \gamma, \chi\}^T \tag{49}$$

3.2.4 Arbitrarily oriented reference configuration

In the general case the reference configuration (Fig.13) is not aligned with the X axis. Considering an angle ϕ of rotation:

$$\begin{aligned} \cos\phi = X_{21}/L_0 \qquad & \sin\phi = Y_{21}/L_0 \\ Y_{21} = Y_2 - Y_1 \qquad & L_0^2 = X_{21}^2 + Y_{21}^2 \end{aligned} \tag{50}$$

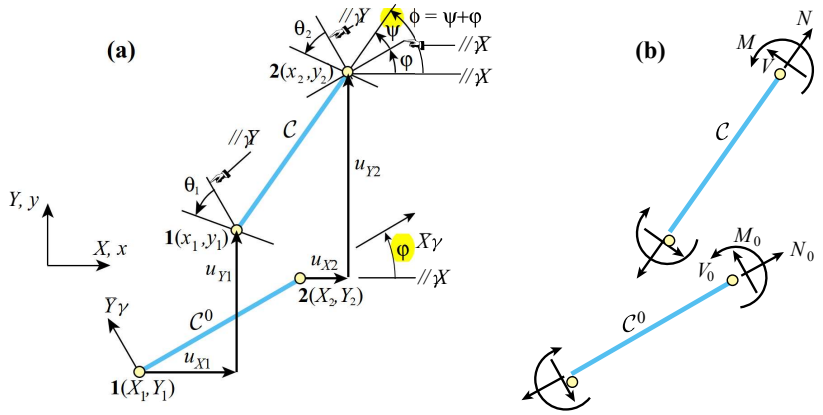

Figure 13: Arbitrarily oriented reference configuration

The angle $\Phi = \psi + \phi$ is derived by:

$$\cos\Phi = \cos\psi + \phi = x_{21}/L \qquad \sin\Phi = \sin\psi + \phi = y_{21}/L \tag{51}$$

With:

$$x_{21} = X_{21} + u_2 - u_1 \qquad y_{21} = Y_{21} + v_2 - v_1 \tag{52}$$

The length L in the current configuration is obtained by:

$$L^2 = x_{21}^2 + y_{21}^2 \tag{53}$$

3.2.5 Constitutive equations

The material is assumed to be elastic, homogeneous and isotropic. The only non-zero Piola-Kirchhoff stresses are the s_{XX} and s_{XY} components. The stress vector $\boldsymbol{s}$ is related to the GL strains with the following relation:

$$\boldsymbol{s} = \begin{pmatrix} s_{XX} \\ s_{XY} \end{pmatrix} = \begin{pmatrix} s_1 \\ s_2 \end{pmatrix} = \begin{pmatrix} s_1^0 + Ee_1 \\ s_2^0 + G_e 2 \end{pmatrix} = \begin{pmatrix} s_1 \\ s_2 \end{pmatrix} + \begin{pmatrix} E & 0 \\ 0 & G \end{pmatrix} \begin{pmatrix} e_1 \\ e_2 \end{pmatrix} = \boldsymbol{s}^0 + \boldsymbol{E}\boldsymbol{e} \tag{54}$$

In the current configuration the axial force N, the transverse shear force V and the bending moment M can be expressed as:

$$N = N^0 + EA_0 e \qquad V = V^0 + GA_0\gamma \qquad M = M^0 + EI_0\chi \tag{55}$$

The stress resultant vector $\boldsymbol{z}$ is:

$$\boldsymbol{z} = \{N, V, M\}^T \tag{56}$$

3.2.6 Strain energy

Due to the fact that the separability of the residual equation holds, the strain internal energy is:

$$U = \int_{V_0} (\boldsymbol{s}^0)^T \boldsymbol{e} + 1/2 \boldsymbol{e}^T \boldsymbol{E} \boldsymbol{e} dV \tag{57}$$

By computing the area integrals this gets:

$$U = \int_{L_0} N^0 e + 1/2 E A_0 e^2 \, d\overline{X} + \int_{L_0} V^0 \gamma + 1/2 G A_0 \gamma^2 \, d\overline{X} + \int_{L_0} M^0 \chi + 1/2 E I_0 \chi^2 \, d\overline{X} \tag{58}$$

3.2.7 Internal force vector

The internal force vector is derived by considering the first variation of the internal energy U.

$$\delta U = \boldsymbol{p}^T \delta \boldsymbol{u} = \int_{L_0} (N \delta e + V \delta \gamma + M \delta \chi) \, d\overline{X} = \int_{L_0} \boldsymbol{z}^T \delta \boldsymbol{h} \, d\overline{X} = \int_{L_0} \boldsymbol{z}^T \boldsymbol{B} \, d\overline{X} \, \delta U \tag{59}$$

This expression can be computed by a one point Gauss integration with sample point at $\xi = 0$. In the next passages the subscript m stands for "midpoint" of the beam element. The following quantities are defined:

$$\theta_m = (\theta_1 + \theta_2)/2 \qquad \omega_m = \theta_m + \phi \qquad c_m = \cos \omega_m \qquad s_m = \sin \omega_m \tag{60}$$

$$e_m = L \, \cos(\theta_m - \psi)/L_0 - 1 \qquad \gamma_m = L \sin \psi - \theta_m / L_0 \tag{61}$$

The matrix $\boldsymbol{B}$ is computed as:

$$\boldsymbol{B}_m = \boldsymbol{B}(\xi = 0) = \frac{1}{L_0} \begin{pmatrix} -c_m & -s_m & -1/2 L_0 \gamma_m & c_m & s_m & -1/2 L_0 \gamma_m \\ s_m & -c_m & 1/2 L_0 (1 + e_m) & s_m & -c_m & 1/2 L_0 (1 + e_m) \\ 0 & 0 & -1 & 0 & 0 & 1 \end{pmatrix} \tag{62}$$

The following expression is valid in order to compute the internal forces vector:

$$\boldsymbol{p} = L_0 \boldsymbol{B}_m^T \boldsymbol{z} = \begin{pmatrix} -c_m & -s_m & -1/2 L_0 \gamma_m & c_m & s_m & -1/2 L_0 \gamma_m \\ s_m & -c_m & 1/2 L_0 (1 + e_m) & s_m & -c_m & 1/2 L_0 (1 + e_m) \\ 0 & 0 & -1 & 0 & 0 & 1 \end{pmatrix}^T \begin{pmatrix} N \\ V \\ M \end{pmatrix} \tag{63}$$

3.2.8 Tangent stiffness matrix

The first variation of the internal force $\boldsymbol{p}$ defines the tangent stiffness matrix:

$$\delta \boldsymbol{p} = \int_{L_0} \boldsymbol{B}^T \delta \boldsymbol{z} + \delta \boldsymbol{B}^T \boldsymbol{z} \, d\overline{X} = (\boldsymbol{K}_M + \boldsymbol{K}_G) \delta \boldsymbol{u} = \boldsymbol{K} \delta \boldsymbol{u} \tag{64}$$

Here $\boldsymbol{K}_M$ and $\boldsymbol{K}_G$ are respectively the material and the geometric stiffness matrix. The material stiffness matrix is derived by considering the variation of the stress while keeping constant the term $\boldsymbol{B}$. Thus:

$$\delta \boldsymbol{z} = \begin{pmatrix} \delta N \\ \delta V \\ \delta M \end{pmatrix} = \begin{pmatrix} E A_0 & 0 & 0 \\ 0 & G A_0 & 0 \\ 0 & 0 & E I_0 \end{pmatrix} \begin{pmatrix} \delta e \\ \delta \gamma \\ \delta \chi \end{pmatrix} = \boldsymbol{S} \delta \boldsymbol{h} \tag{65}$$

By substituting the expressions of $\delta \boldsymbol{h} = \boldsymbol{B}\delta \boldsymbol{u}$ and the term $\boldsymbol{B}^T \delta \boldsymbol{z}$:

$$\boldsymbol{K}_M = \int_{L_0} \boldsymbol{B}^T \boldsymbol{S} \boldsymbol{B}\, d\overline{X} = \boldsymbol{K}_M^a + \boldsymbol{K}_M^b + \boldsymbol{K}_M^s \tag{66}$$

Again the integral is computed by exploiting a one Gauss integration point at $\xi = 0$. Here $a_1 = 1 + e_m$:

$$\boldsymbol{K}_M^a = \frac{EA_0}{L_0} \begin{pmatrix} c_m^2 & c_m s_m & -c_m \gamma_m L_0/2 & -c_m^2 & -c_m s_m & -c_m \gamma_m L_0/2 \\ c_m s_m & s_m^2 & -\gamma_m L_0 s_m/2 & -c_m s_m & -s_m^2 & -\gamma_m L_0 s_m/2 \\ -c_m \gamma_m L_0/2 & -\gamma_m L_0 s_m/2 & \gamma_m^2 L_0^2/4 & c_m \gamma_m L_0/2 & \gamma_m L_0 s_m/2 & \gamma_m^2 L_0^2/4 \\ -c_m^2 & -c_m s_m & c_m \gamma_m L_0/2 & c_m^2 & c_m s_m & c_m \gamma_m L_0/2 \\ -c_m s_m & -s_m^2 & \gamma_m L_0 s_m/2 & c_m s_m & s_m^2 & \gamma_m L_0 s_m/2 \\ -c_m \gamma_m L_0/2 & -\gamma_m L_0 s_m/2 & \gamma_m^2 L_0^2/4 & c_m \gamma_m L_0/2 & \gamma_m^2 L_0^2/4 & \end{pmatrix} \tag{67}$$

$$\boldsymbol{K}_M^b = \frac{EI_0}{L_0} \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 \end{pmatrix} \tag{68}$$

$$\boldsymbol{K}_M^s = \frac{GA_0}{L_0} \begin{pmatrix} s_m^2 & -c_m s_m & -a_1 L_0 s_m/2 & -s_m^2 & c_m s_m & -a_1 L_0 s_m/2 \\ -c_m s_m & c_m^2 & c_m a_1 L_0/2 & c_m s_m & -c_m^2 & c_m a_1 L_0/2 \\ -a_1 L_0 s_m/2 & c_m a_1 L_0/2 & a_1^2 L_0^2/4 & a_1 L_0 s_m/2 & -c_m a_1 L_0/2 & a_1^2 L_0^2/4 \\ -s_m^2 & c_m s_m & a_1 L_0 s_m/2 & s_m^2 & -c_m s_m & a_1 L_0 s_m/2 \\ c_m s_m & -c_m^2 & -c_m a_1 L_0/2 & -c_m s_m & c_m^2 & -c_m a_1 L_0/2 \\ -a_1 L_0 s_m/2 & c_m a_1 L_0/2 & a_1^2 L_0^2/4 & a_1 L_0 s_m/2 & -c_m a_1 L_0/2 & a_1^2 L_0^2/4 \end{pmatrix} \tag{69}$$

The geometric stiffness matrix $\boldsymbol{K}_G$ derives from the variation of $\boldsymbol{B}$ with fixed stress vector $\boldsymbol{z}$. In the next expression the index summation from Einstein is exploited:

$$K_{Gij}\delta u_j = \int_{L_0} \delta \boldsymbol{B}^T \boldsymbol{z}\, dX = \int_{L_0} \frac{\partial\, B_{ki}}{\partial u_j} \delta u_j z_k\, dX = \int_{L_0} A_{ki}^j z_k dX \delta u_j \tag{70}$$

In the next relation V_m and N_m are the values of V and N computed in $\xi = 0$. The geometric stiffness matrix can be computed as $\boldsymbol{K}_G = \boldsymbol{K}_{GN} + \boldsymbol{K}_{GV}$, where these are defined as follows:

$$\boldsymbol{K}_{GN} = \frac{N_m}{2} \begin{pmatrix} 0 & 0 & s_m & 0 & 0 & s_m \\ 0 & 0 & -c_m & 0 & 0 & -c_m \\ s_m & -c_m & -1/2 L_0(1+e_m) & -s_m & c_m & -1/2 L_0(1+e_m) \\ 0 & 0 & -s_m & 0 & 0 & -s_m \\ 0 & 0 & c_m & 0 & 0 & c_m \\ s_m & -c_m & -1/2 L_0(1+e_m) & -s_m & c_m & -1/2 L_0(1+e_m) \end{pmatrix} \tag{71}$$

$$\boldsymbol{K}_{GV} = \frac{V_m}{2}\begin{pmatrix} 0 & 0 & c_m & 0 & 0 & c_m \\ 0 & 0 & -s_m & 0 & 0 & -s_m \\ c_m & s_m & -1/2L_0\gamma_m & -c_m & -s_m & -1/2L_0\gamma_m \\ 0 & 0 & -c_m & 0 & 0 & -c_m \\ 0 & 0 & -s_m & 0 & 0 & -s_m \\ c_m & s_m & -1/2L_0\gamma_m & -c_m & -s_m & -1/2L_0\gamma_m \end{pmatrix} \tag{72}$$

4 Solution method

In general the analytical solution to the problem of deriving the load-displacement curve is represented by a system of nonlinear partial differential equations.

Once the system is discretized the obtained system of N equations can be written as:

$$\boldsymbol{r}(\boldsymbol{u}, \boldsymbol{\Lambda}) = \boldsymbol{0} \tag{73}$$

Or equivalently:

$$r_i(u_1, u_2, ...u_N, \Lambda_1, \Lambda_2, ..., \Lambda_M) = 0 \qquad \text{for} \quad i = 1, ..., N \tag{74}$$

Here $\boldsymbol{u}$ is the state vector, which is the displacements vector containing as components all the generalized coordinates u_j for $j = 1, ..., N$ of the studied structure, $\boldsymbol{\Lambda}$ is the vector containing as components the control parameters Λ_k for $k = 1, ..., m$. In the present study, just one control parameter is considered and the control parameters vector $\boldsymbol{\Lambda}$ contains just one component, namely λ. A further hypotheses here introduced is that the external nodal forces vector is proportional to the load parameter λ:

$$\boldsymbol{f}_{ext}(\lambda) = \lambda \cdot \boldsymbol{f}_{ref} \tag{75}$$

Under these circumstances the hypotheses of proportionality holds. Note that in this case also the external work will be proportional to the load parameter:

$$W(\boldsymbol{u}, \lambda) \propto \lambda \tag{76}$$

In this case the incremental load vector is constant and equal to the reference vector $\boldsymbol{f}_{ext}$:

$$\boldsymbol{f}'_{ext} = \frac{\partial f_{ext}(\lambda)}{\partial \lambda} = \boldsymbol{f}_{ref} \tag{77}$$

Another hypotheses valid for the present study purposes is that the external forces nodal vector $\boldsymbol{f}_{ext}$ does not depend on the generalized coordinates. It does not in fact depend on the state vector $\boldsymbol{u}$. Thus, the vector is just dependent on the control parameter λ:

$$\boldsymbol{f}_{ext}(\lambda) \tag{78}$$

In this case the hypothesis of separable equations stands, thus the residuals are separable. The equations system can be written as:

$$\boldsymbol{f}_{int}(\boldsymbol{u}) = \boldsymbol{f}_{ext}(\lambda) \qquad \rightarrow \boldsymbol{r}(\boldsymbol{u}, \lambda) = \boldsymbol{f}_{int}(\boldsymbol{u}) - \boldsymbol{f}_{ext}(\lambda) \tag{79}$$

This hypotheses would not hold for example in case of follower load. In this case the external load forces vector would depend also on the state of the system $\boldsymbol{u}$.

Differently, under separable equations hypotheses, the tangent stiffness matrix $\boldsymbol{K}$ can be identified as:

$$\boldsymbol{K}_{tan}(\boldsymbol{u}) = \frac{\partial \boldsymbol{r}}{\partial \boldsymbol{u}} = \frac{\partial \boldsymbol{f}_{int}}{\partial \boldsymbol{u}} \tag{80}$$

In order to follow the load displacement curve, the general method is structured in order to see the state vector $\boldsymbol{u}$ and the load parameter λ as function of an evolution parameter t. This parameter is referred as pseudotime parameter. Note that in the present study no dynamical problem is considered. Thus the reference to time does not have to be confused to the real time flowing. In this hypotheses the system evolution $\{\boldsymbol{u}(t), \lambda(t)\}$ is a curve in the displacements-load space described by the pseudotime parameter t. Here is convenient to introduce a compact notation in order to identify the curve in the displacements-load space:

$$\boldsymbol{x}(t) = \{\boldsymbol{u}(t), \lambda(t)\} \tag{81}$$

The vector $\boldsymbol{x}(t)$ contains all the $N+1$ unknowns of the system. If the derivative with respect to the pseudotime t is considered, a velocity vector can be derived at each point of the curve $\boldsymbol{x}(t)$ provided that the curve is smooth (C^1). The velocity vector is:

$$\dot{\boldsymbol{x}}(t) = \{\dot{\boldsymbol{u}}, \dot{\lambda}\} \tag{82}$$

If the displacement vector $\boldsymbol{u}$ is interpreted as a function of the control parameter λ, the composed function can be derived as:

$$\boldsymbol{u}(t) = \boldsymbol{u}(\lambda(t)) \tag{83}$$

An important observation is that while the values $u_i(t)$ and $\lambda(t)$ are all univocally defined for one value of t and are functions, the values $u_i(\lambda)$ can loose the function property as to one value of λ more than one values of u_i can be related.

Taking into account the composition of these relations, the following can be written exploiting the chain rule of derivatives:

$$\dot{\boldsymbol{u}} = \frac{\partial \boldsymbol{u}}{\partial t} = \frac{\partial \boldsymbol{u}}{\partial \lambda} \cdot \frac{\partial \lambda}{\partial t} = \boldsymbol{v} \cdot \dot{\lambda} \tag{84}$$

$$\dot{\lambda} = \frac{\partial \lambda}{\partial t} = \frac{\partial \lambda}{\partial \lambda} \cdot \frac{\partial \lambda}{\partial t} = 1 \cdot \dot{\lambda} \tag{85}$$

In the previous relation the vector $\boldsymbol{v}$ is the incremental velocity vector and can also be written as:

$$\boldsymbol{v} = \boldsymbol{u}' \tag{86}$$

Here the prime indicates the derivative respect to the control parameter λ. Thus, the velocity vector $\dot{\boldsymbol{x}}(t)$ can be written as:

$$\dot{\boldsymbol{x}}(t) = \{\dot{\boldsymbol{u}}, \dot{\lambda}\} = \{\boldsymbol{v}, 1\} \cdot \dot{\lambda} \tag{87}$$

Here we define:

$$\boldsymbol{t} = \{\boldsymbol{v}, 1\} \tag{88}$$

Note that if the curve is smooth the velocity vector $\dot{\boldsymbol{x}}(t)$ exists and it is tangent to the curve $\boldsymbol{x}(t)$. Due to the fact that $\dot{\lambda}$ is a scalar, also the vector $\boldsymbol{t}$ is tangent to the curve. A local tangent versor can be derived via normalization of this vector:

$$\hat{\boldsymbol{t}} = 1/f \cdot \{\boldsymbol{v}, 1\} \tag{89}$$

Where f is the norm of the vector:

$$f = \sqrt{1 + \boldsymbol{v} \cdot \boldsymbol{v}} \tag{90}$$

If the first order incremental form of the residual equations is considered a relation linking the reference external nodal force vector $\boldsymbol{f}_{ref}$ and the incremental velocity $\boldsymbol{v}$ can be derived:

$$\dot{\boldsymbol{r}} = \boldsymbol{0} \qquad \rightarrow \qquad \dot{\boldsymbol{f}}_{int} = \dot{\boldsymbol{f}}_{ext} \tag{91}$$

$$\boldsymbol{K}_{tan}\dot{\boldsymbol{u}} = \dot{\lambda}\boldsymbol{f}_{ref} \qquad \rightarrow \qquad \boldsymbol{K}_{tan}\boldsymbol{u}' \cdot \frac{\partial \lambda}{\partial t} = 1 \cdot \frac{\partial \lambda}{\partial t}\boldsymbol{f}_{ref} \tag{92}$$

Thus the following equation holds:

$$\boldsymbol{K}_{tan}\boldsymbol{u}' = \boldsymbol{f}_{ref} \tag{93}$$

By inverting this relation:

$$\boldsymbol{u}' = (\boldsymbol{K}_{tan})^{-1}\boldsymbol{f}_{ref} \tag{94}$$

4.1 Newton-Rhapson method

Finding the solution to the system $\boldsymbol{r}(\boldsymbol{u}, \lambda) = \boldsymbol{0}$ for the current increment in the current load step means finding the particular state vector $\boldsymbol{u}$ and control parameter λ that satisfy the condition of balancement for a given pseudotime value. In order to find the solution of a nonlinear system of equations many methods can be found in literature. The various procedure can be distinguished on the order of the algorithm:

- zero order methods do not need the numerical or analytical providing of the gradient (jacobian) of the objective function.
- in first order methods the gradient of the objective function has to be provided. This can be done by exploiting an analytical expression of the matrix or a numerical computation of it. Obviously an explicit expression can favourably condition the computational efficiency.
- second order method, exploiting the Hessian of the functions. This takes into account also the local curvatures.

The balance force equation $\boldsymbol{r} = \boldsymbol{f}_{int}(\boldsymbol{u}) - \boldsymbol{f}_{ext}(\lambda)$ is a system of N equations. The system at the considered increment of the given step is described by the vector $\boldsymbol{u}$ and the control parameter λ. Therefore there are $N + 1$ unknowns to be determined. In order to close the equations system an additional equation has to be enclosed, this is the constraint equation, describing how the load

increment $\Delta\lambda$ should relate with the displacement increment $\Delta\boldsymbol{u}$ following the pseudotime evolution. This relation is generally given by an equation of the following form:

$$c(\Delta\boldsymbol{u}, \Delta\lambda) = 0 \tag{95}$$

The constraints equations will be explicitly introduced in the subsequent sections. The adopted relation is strictly linked with the control strategy exploited.

By collecting the equations $\boldsymbol{r} = 0$ and $c = 0$ in one single vector we define an extended residual vector $\tilde{\boldsymbol{r}}$:

$$\tilde{\boldsymbol{r}} = \begin{pmatrix} \boldsymbol{r} \\ c \end{pmatrix} \tag{96}$$

In the present study a Newton-Rhapson method is implemented in order to find the root of the nonlinear system at the given increment. The solution at the considered step for the increment $j-1$ is known. The objective is to find the solution $\boldsymbol{x}_j = \{\boldsymbol{u}_j, \lambda_j\}$ for the next increment j that satisfies the balance equation. In order to do that an iterative process is used. The index k indicates in the following passages the iteration number. In a particular iteration k of the process the state vector $\boldsymbol{u}_{j,k}$ and the value of $\lambda_{j,k}$ are known. The scope is to correct the point $\boldsymbol{x}_{j,k} = \{\boldsymbol{u}_{j,k}, \lambda_{j,k}\}$ into $\boldsymbol{x}_{j,k+1} = \{\boldsymbol{u}_{j,k+1}, \lambda_{j,k+1}\}$ in order to get closer to the point that satisfy $\boldsymbol{r} = 0$. At the current iteration the residual condition is not in general satisfied:

$$\tilde{\boldsymbol{r}}_{j,k}(\boldsymbol{x}_{j,k}) = \boldsymbol{f}_{int}(\boldsymbol{u}_{j,k}) - \boldsymbol{f}_{ext}(\lambda_{j,k}) \neq \boldsymbol{0} \tag{97}$$

The residual function is expanded locally into Taylor series functions. If the first order is considered the $n-th$ component of the residual will be:

$$\tilde{\boldsymbol{r}}(\boldsymbol{u} + d\boldsymbol{u}, \lambda + d\lambda) \approx \tilde{\boldsymbol{r}}(\boldsymbol{u}, \lambda) + \frac{\partial \tilde{\boldsymbol{r}}}{\partial \boldsymbol{u}} \cdot d\boldsymbol{u} + \frac{\partial \tilde{\boldsymbol{r}}}{\partial \lambda} \cdot d\lambda \tag{98}$$

In order to do that a gradient following (Fig.14) approach is used:

$$\boldsymbol{x}_{j,k+1} = \boldsymbol{x}_{j,k} - \left(\frac{\partial \tilde{\boldsymbol{r}}}{\partial \boldsymbol{x}}\right)^{-1}_{j,k} \tilde{\boldsymbol{r}}_{j,k} \tag{99}$$

More explicitly:

$$\Delta\boldsymbol{x}_{j,k+1} = \boldsymbol{x}_{j,k+1} - \boldsymbol{x}_{j,k} = \begin{pmatrix} \Delta\boldsymbol{u}_{j,k+1} \\ \Delta\lambda_{j,k+1} \end{pmatrix} = -\left(\frac{\partial \tilde{\boldsymbol{r}}}{\partial \boldsymbol{x}}\right)^{-1}_{j,k} \tilde{\boldsymbol{r}}_{j,k} = -\left(\frac{\partial \tilde{\boldsymbol{r}}}{\partial \boldsymbol{x}}\right)^{-1}_{j,k} \begin{pmatrix} \boldsymbol{r}_{j,k} \\ c_{j,k} \end{pmatrix} \tag{100}$$

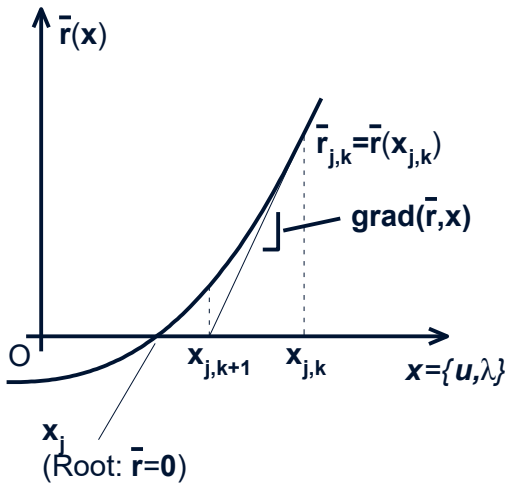

Figure 14: Newton-Rhapson solution update iteration scheme

This equation allows to compute the increments in the displacements and the load parameter λ. Here the matrix multiplying the residual is the jacobian of the function $\tilde{\boldsymbol{r}}(\boldsymbol{x})$:

$$\boldsymbol{J} = \frac{\partial \tilde{\boldsymbol{r}}}{\partial \boldsymbol{x}} \qquad \text{Jacobian} \tag{101}$$

The jacobian is a $(N+1) \times (N+1)$ matrix. The components are:

$$J_{nm} = \frac{\partial \tilde{r}_n}{\partial x_m} \qquad \text{for} \quad n, m = 1, 2, ..., N+1 \tag{102}$$

As the function $\tilde{\boldsymbol{r}}(\boldsymbol{x})$ and the variables $\boldsymbol{x}$ are divided in two blocks, so will be the jacobian. For the indexes $m, n = 1, .., N$ the $\boldsymbol{x}$ components are the displacements $\boldsymbol{u}$ and the residuals $\tilde{\boldsymbol{r}}$ are the balancement equation components $\boldsymbol{r}$. Therefore the jacobian components will coincide with the components of the tangent stiffness matrix $\boldsymbol{K}_{tan}$, in fact:

$$\frac{\partial \tilde{r}_n}{\partial x_m} = \frac{\partial r_n}{\partial u_m} = \frac{\partial (f_{int,n}(\boldsymbol{u}) - f_{ext,n}(\lambda))}{\partial u_m} = \frac{\partial f_{int,n}(\boldsymbol{u})}{\partial u_m} = K_{tan,nm} \qquad \text{for} \quad n, m = 1, ..., N \tag{103}$$

For $n = 1, ..., N$ and $m = N+1$ the $\tilde{\boldsymbol{r}}$ vector consists in the vector $\boldsymbol{r}$ and the $N+1-th$ component of the vector $\boldsymbol{x}$ is the control parameter λ. Thus:

$$\frac{\partial \tilde{r}_n}{\partial x_m} = \frac{\partial r_n}{\partial \lambda} = \frac{\partial (f_{int,n}(\boldsymbol{u}) - f_{ext}(\lambda))}{\partial \lambda} = -\frac{\partial f_{ext,n}(\lambda)}{\partial \lambda} = -\frac{\partial (\lambda \cdot f_{ref,n}))}{\partial \lambda} = -f_{ref,n} \qquad \text{for} \quad n = 1, ..., N \quad m = N+1 \tag{104}$$

For $n = N+1$ and $m = 1, ..., N$ the considerd component of the residual vector $\tilde{\boldsymbol{r}}$ coincides with the constraint function $c(\boldsymbol{u}, \lambda)$ and the considered components of $\boldsymbol{x}$ are those of the state vector $\boldsymbol{u}$. Therefore:

$$a_m := \frac{\partial \tilde{r}_n}{\partial x_m} = \frac{\partial c(\boldsymbol{u}, \lambda)}{\partial u_m} \qquad \text{for} \quad n = N+1 \quad m = 1, ..., N \tag{105}$$

Here a vector of dimension N named $\boldsymbol{a}$ was defined. For $n = N+1$ and $m = N+1$ the considered component of the residual vector $\tilde{\boldsymbol{r}}$ coincides with the constraint function $c(\boldsymbol{u}, \lambda)$ and the considered

component of $\boldsymbol{x}$ is the control parameter λ. Therefore:

$$g := \frac{\partial \tilde{r}_n}{\partial x_m} = \frac{\partial c(\boldsymbol{u}, \lambda)}{\partial \lambda} \qquad \text{for} \quad n = N+1 \quad m = N+1 \tag{106}$$

Here a scalar quantity g was defined.

$$\boldsymbol{J} = \left(\begin{array}{c|c} \boldsymbol{K}_{tan} & -\boldsymbol{f}_{ref} \\ \hline \boldsymbol{a}^T & g \end{array}\right) = \left(\begin{array}{ccc|c} K_{11} & ... & K_{1N} & -f_{ref,1} \\ ... & ... & ... & ... \\ K_{N1} & ... & K_{NN} & -f_{ref,N} \\ \hline a_1 & ... & a_N & g \end{array}\right) \tag{107}$$

4.2 Control strategies

In order to follow the load-displacement path, different types of control methods can be found in the literature. These can be load control, displacement control or arclength control methods. The arclength control strategy can be subdivided into normal plane, hyperspherical, global hyperelliptical or local hyperelliptical approach. Here a predictor-corrector method is used. The examined strategies are load, displacement and Riks arclength control methods.

4.2.1 Load control method

In the load control method the load increment $\Delta\lambda$ is constrained to be a specified value. Therefore, the constraint equation is:

$$c(\Delta\lambda) = \Delta\lambda - \Delta l = 0 \tag{108}$$

The predictor step trivially consists in updating the value of λ to the successive value. The NR method successively corrects the $\boldsymbol{x}$ solution. Note that in this case the constraint function c is not dependent by the displacements vector $\boldsymbol{u}$. The vector $\boldsymbol{u}$ and the parameter λ can be written more explicitly:

$$c(\lambda - \lambda_k) = \lambda - \lambda_k - \Delta l = 0 \tag{109}$$

In these circumstances the vector $\boldsymbol{a}$ and the scalar g are:

$$\boldsymbol{a} = \frac{\partial c}{\partial \boldsymbol{u}} = \boldsymbol{0} \tag{110}$$

$$\boldsymbol{g} = \frac{\partial c}{\partial \lambda} = 1 \tag{111}$$

This procedure is not capable of passing the critical points.

4.2.2 Displacement control method

In the displacement control method the displacement increment Δu_m of one the $m-th$ degree of freedom is constrained to be a specified value. Therefore, the constraint equation is:

$$c(\Delta u_m) = \Delta u_m - \Delta l = 0 \tag{112}$$

The predictor step trivially consists in updating the value of the prescribed displacement u_i to the successive value. The NR method successively corrects the $\boldsymbol{x}$ solution. Note that in this case the constraint function c is not dependent by the load parameter λ. The vector $\boldsymbol{u}$ and the parameter λ can be written more explicitly:

$$c(u_m - u_{m,k}) = u_m - u_{m,k} - \Delta l = 0 \tag{113}$$

In these circumstances the vector $\boldsymbol{a}$ and the scalar g are such that:

$$a_i = \delta_{im} = \begin{cases} 1 & if \quad i = m \\ 0 & if \quad i \neq m \end{cases} \qquad \text{for} \quad i = 1, ..., N \tag{114}$$

$$\boldsymbol{g} = 0 \tag{115}$$

This procedure is capable of passing the critical points but will not pass the turning points.

4.2.3 Arclength control method

In the Riks arclength method a plane normal to the local tangent vector computed in the last known configuration that satisfies $\tilde{\boldsymbol{r}} = 0$ is considered. The constraints equation impose that the considered increment should have a prescribed distance with respect to the local normal plane (Fig.15). In order to do that the component $x_{//}$ of the incremental vector projected on the local tangent vector is considered:

$$x_{//} = \{\Delta \boldsymbol{u}, \Delta\lambda\} \cdot \{\boldsymbol{v}/f, 1/f\} \tag{116}$$

The value of $x_{//}$ can be interpreted as the distance from the considered point from the local normal plane. This component is constrained to be equal to a prescribed distance with respect the local normal plane:

$$c(\Delta \boldsymbol{u}, \Delta\lambda) = 1/f \cdot (\boldsymbol{v} \cdot \Delta \boldsymbol{u} + 1 \cdot \Delta\lambda) - l = 0 \tag{117}$$

In this case:

$$\boldsymbol{a} = \boldsymbol{v}/f \tag{118}$$

$$g = 1/f \tag{119}$$

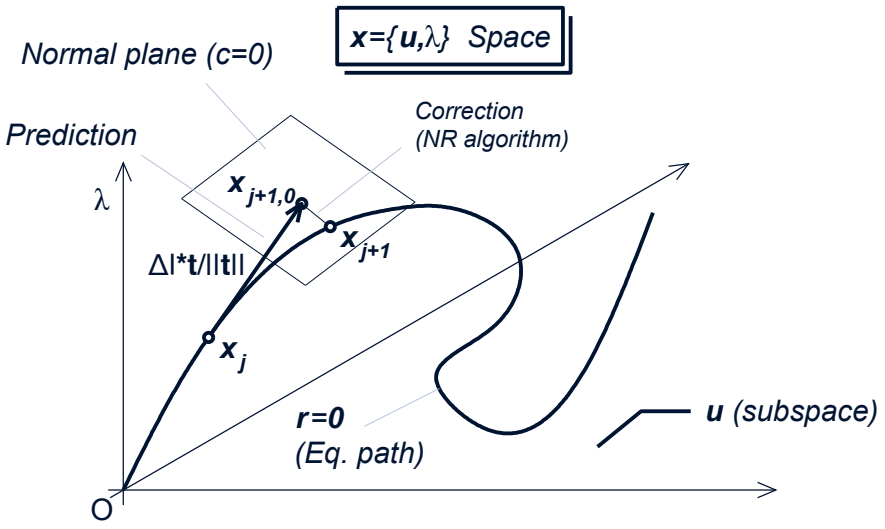

Figure 15: Arclength method solution predictor-correction scheme

In [13] a predictor-corrector approach is described. This method is here implemented in order to run the nonlinear analyses with a Riks arclength control pathfollowing method.

5 Case studies: truss structures

5.1 Von Mieses truss

A simple truss structure is here studied, known in literature as Von Mieses (VM). The structure is composed of two elements. Two different h/a ratios are examined reflecting the intent to study the different behaviour of a deep VM truss and a shallow VM truss. Following properties are used:

Properties	
$a = 1000\,mm$	Half basis
h/a	Height-half basis ratio
$A = 100\,mm^2$	Cross-sectional area
$E = 100\,N/mm^2$	Elastic modulus

5.1.1 Shallow truss

In this case $h/a = 0.3$ ratio is used. The expected behaviour can be derived analytically. The member length as function of the vertical displacement v is:

$$L^2(v) = (h - v)^2 + a^2 \tag{120}$$

The length of the member in the reference (non-deformed) configuration is:

$$L_0^2(v) = L^2(v = 0) = h^2 + a^2 \tag{121}$$

The Green-Lagrange strain can be derived as follows:

$$e(v) = \frac{L(v)^2 - L_0^2}{2L_0^2} \qquad \text{GL strain} \tag{122}$$

The member compression force is:

$$N(v) = EA \cdot e(v) \tag{123}$$

The applied load should be equal to the internal forces, which are obtained by considering the vertical component of the member compression force:

$$P(v) = 2\,(h - v)/L(v)N(v) \tag{124}$$

The force $P(v)$ is plotted as function of v in the range $[0, 2h]$ in Fig.19.

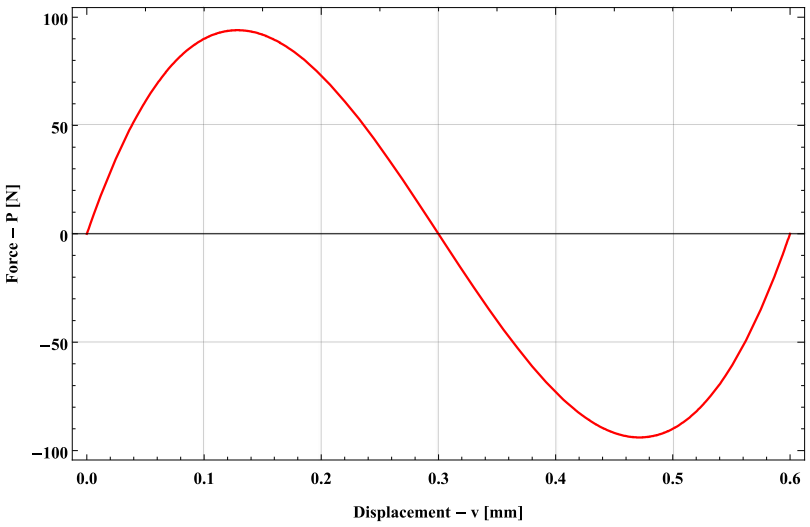

Figure 16: Analytical solution of load-displacement curve for the shallow Von Mieses truss problem

The structure is discretized with one truss FEM element per each element (Fig. 17). Resulting charts are shown in Fig.18. Results are in good agreement with the analytical solution. Note that the determinant presents significant reduction in correspondence of the two critical points. No other reductions are reported. The different animation frames are illustrated in Fig.19.

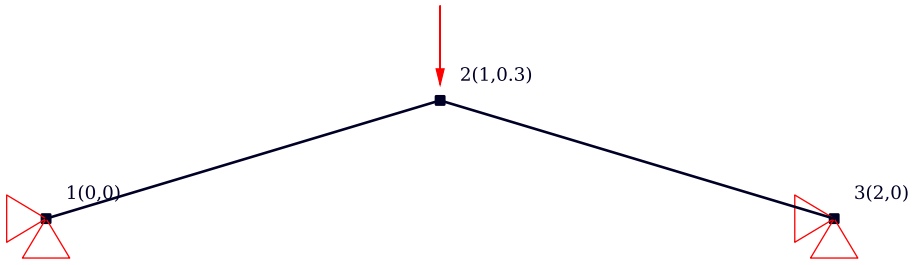

Figure 17: Shallow Von Mieses truss: geometry, discretization , restraints and loading conditions

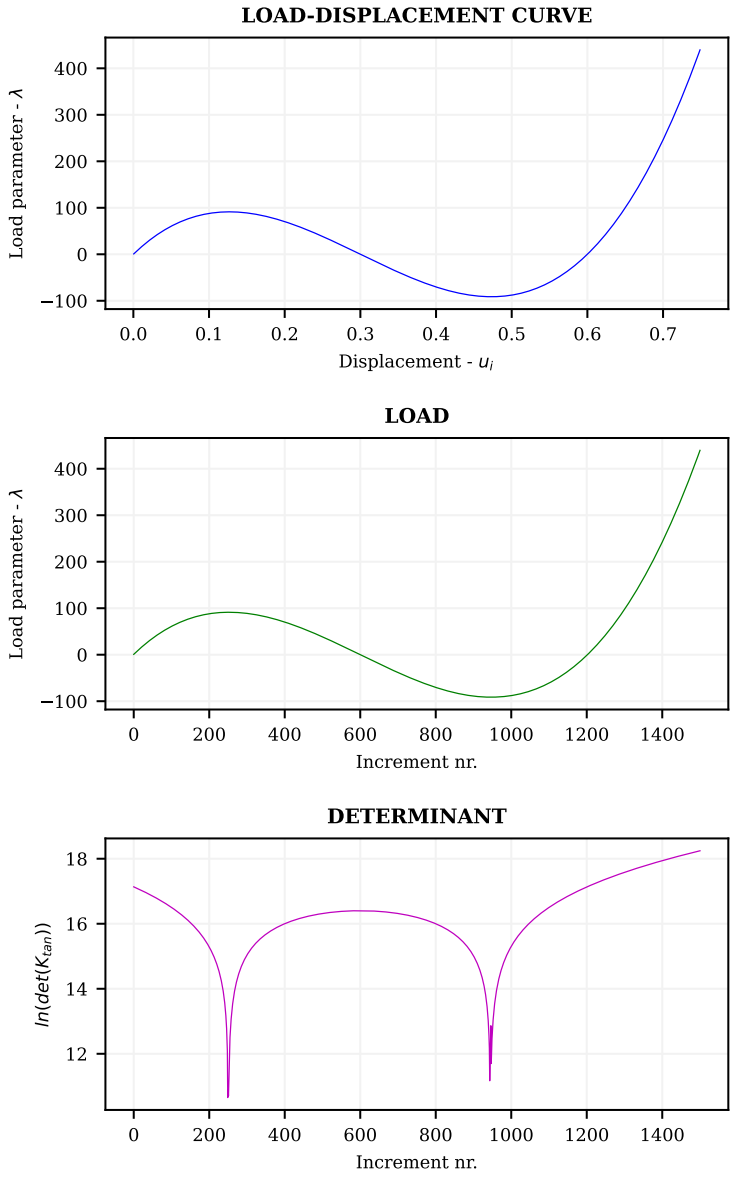

Figure 18: Shallow Von Mieses truss: resulting charts

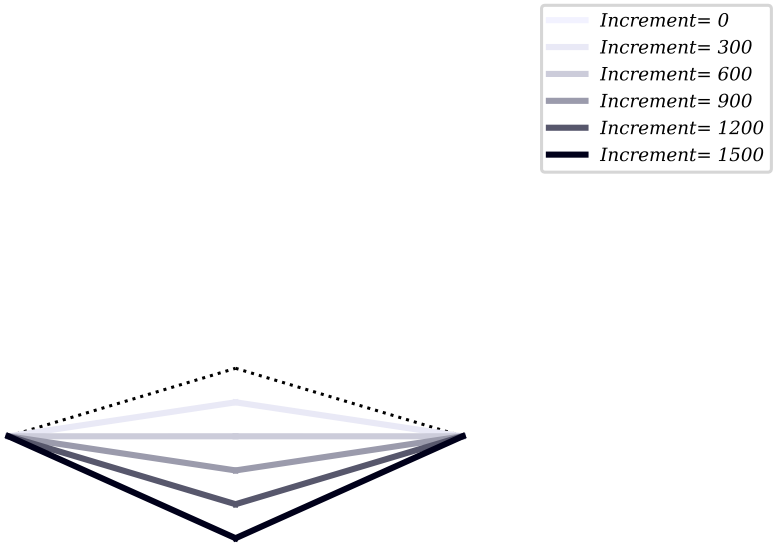

Figure 19: Shallow Von Mieses truss: animation frames for different increments

5.1.2 Deep truss

In this case $h/a = 3$ ratio is used. Geometry is shown in Fig. 20. Results are shown in Fig.21 with the progressive motion shown in Fig.22. The structure exhibits snap through behaviour. Differently from the shallow truss case, the determinant shows a significant reduction in the first hardening branch. This fact is given by a bifurcation path that is further analysed by giving the truss a positive small horizontal force at the top node. Results of this case are shown in Fig. 23 and the animation frames are in Fig.24.

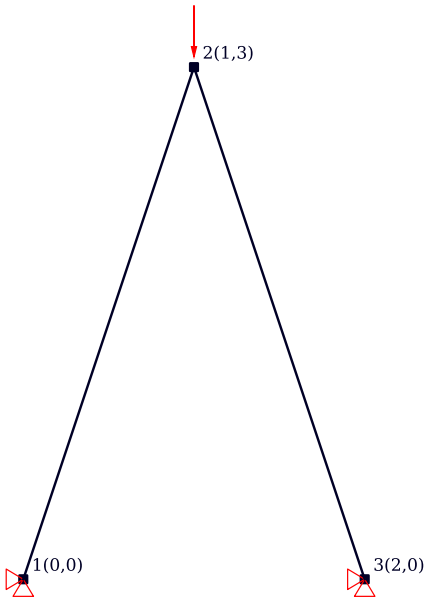

Figure 20: Deep Von Mieses truss: geometry, discretization, restraints and loading conditions

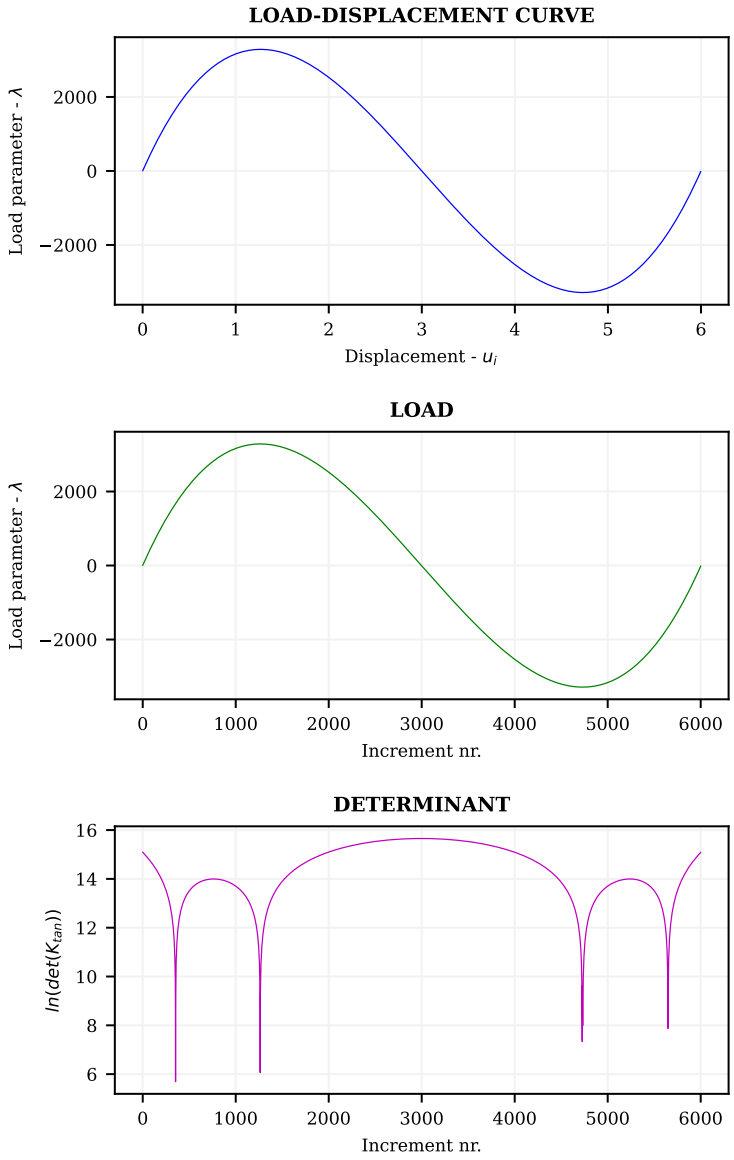

Figure 21: Deep Von Mieses truss: resulting charts

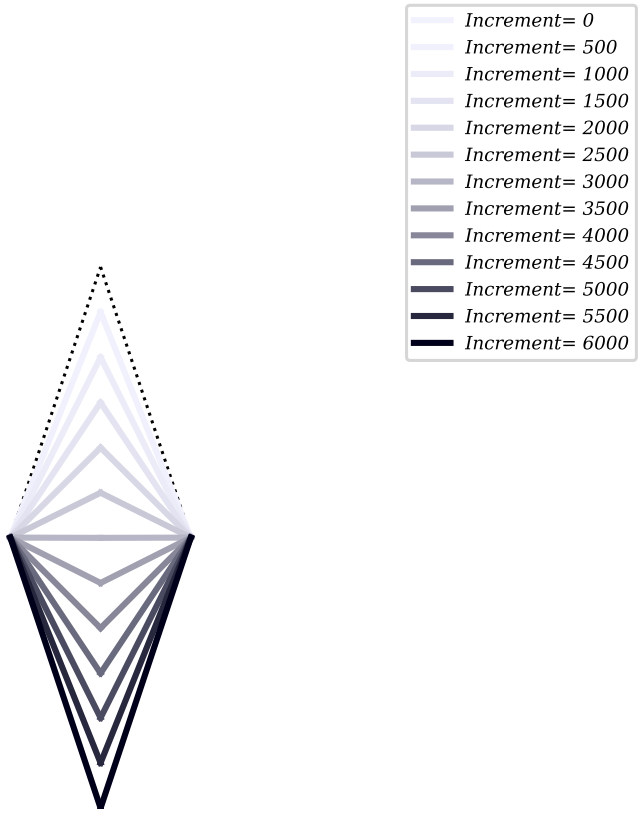

Figure 22: Deep Von Mieses truss: animation frames for different increments

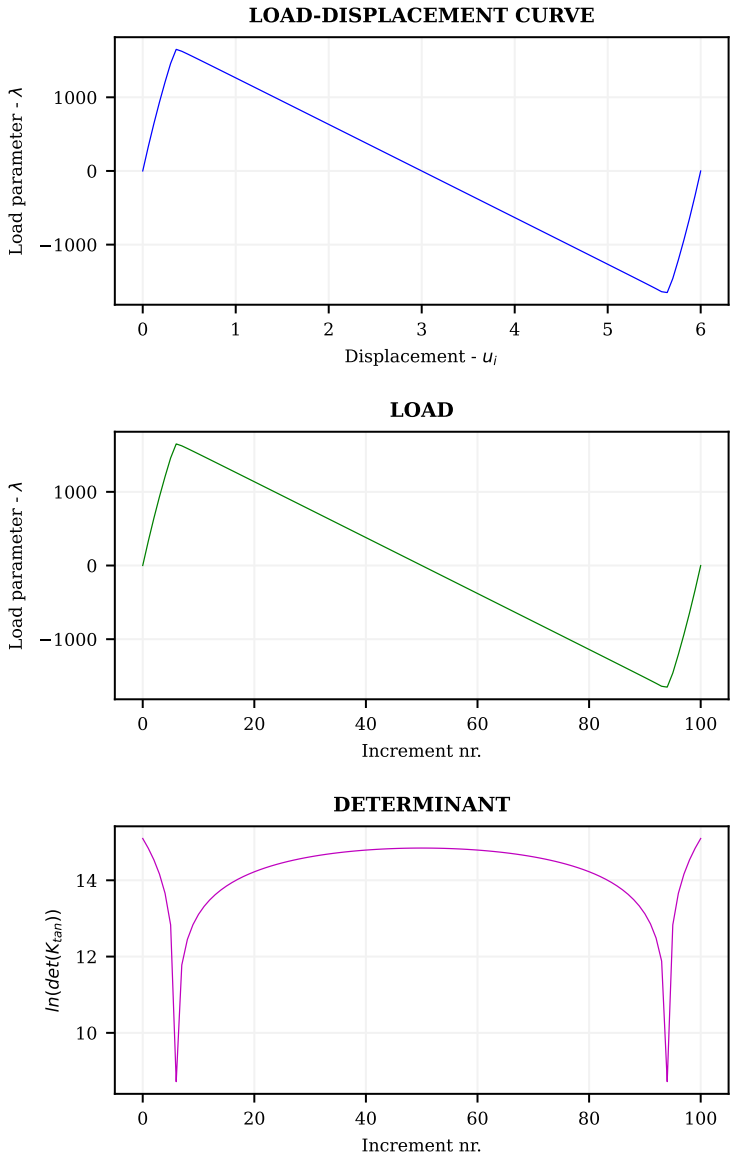

Figure 23: Deep Von Mieses truss with small horizontal force: resulting charts

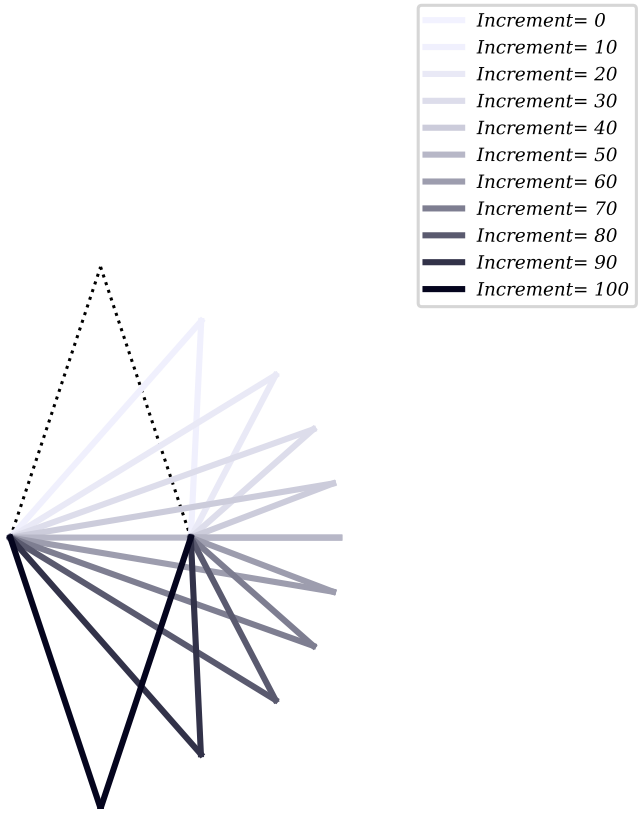

Figure 24: Deep Von Mieses truss: animation frames for different increments

5.2 Shallow arch truss

A shallow arch truss structure is here studied. The truss structure is composed of 131 truss elements. The arch ends are clamped. A vertical load is applied at the top node. A Riks arclength control method is exploited. Following properties are used:

Properties	
$\Delta R = 0.2\,m$	Arch thickness
$1/R = 0.2\,1/m$	Arch curvature
$h = 1\,m$	Hogging height
$A = 0.001\,m^2$	Cross-sectional area
$E = 10000\,kN/m^2$	Elastic modulus

The structure is discretized with one truss FEM element per each element (Fig. 25). Resulting charts are shown in Fig.26. The structure presents a snap through behaviour. Note that the determinant presents significant reduction in correspondence of the two critical points. No other reductions are reported. The different animation frames are illustrated in Fig.19.

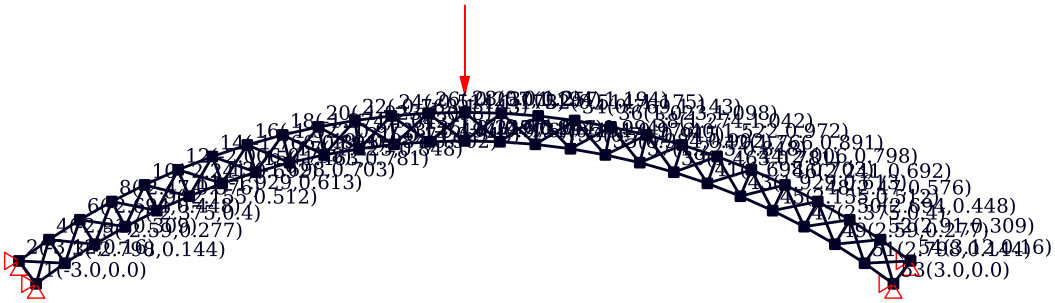

Figure 25: Shallow arch truss: geometry, discretization, restraints and loading conditions

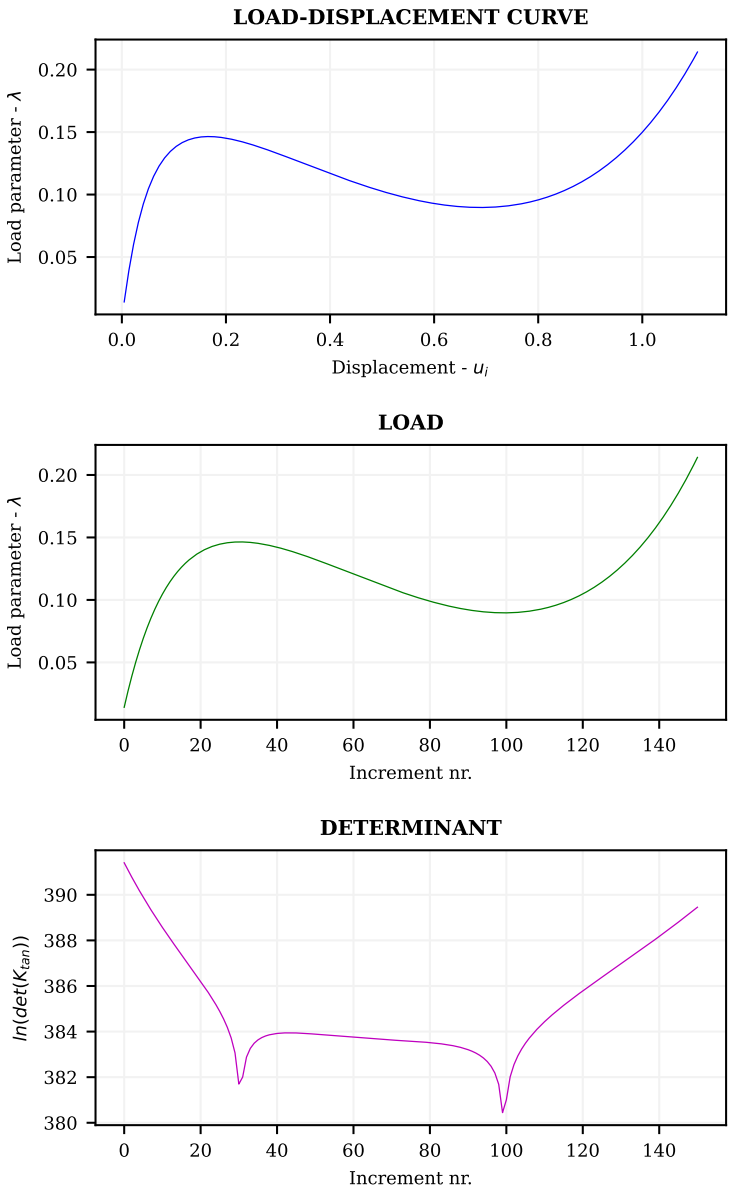

Figure 26: Shallow arch truss: resulting charts

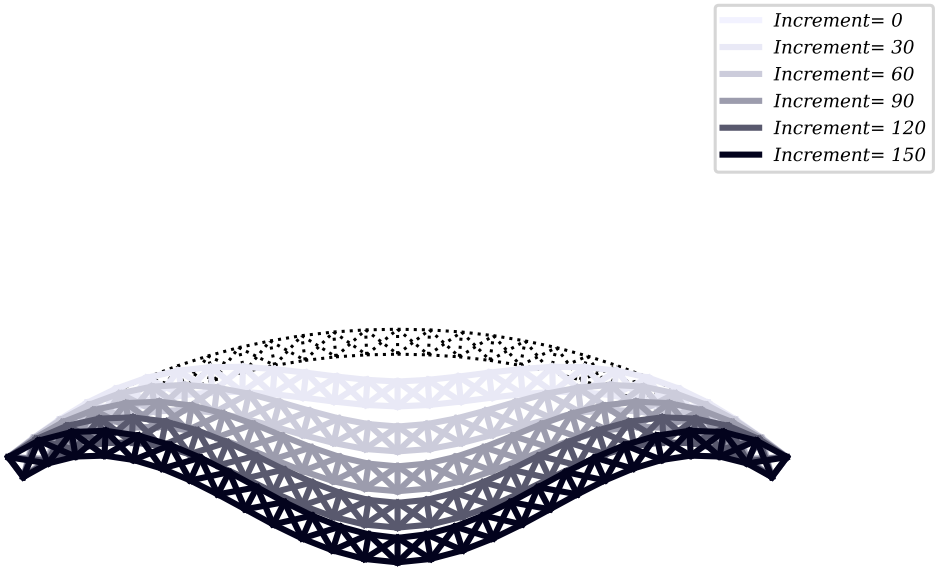

Figure 27: Shallow arch truss: animation frames for different increments

5.3 Cantilever truss

A cantilever restrained truss structure is studied. The structure is loaded with a force at the top end node. The displacement control method is used. Properties are:

Properties	
$L = 4.5\,m$	Total truss length
$h = 0.5\,m$	Truss height
$A = 0.01\,m^2$	Cross-sectional area
$E = 1000\,kN/m^2$	Elastic modulus

The structure is discretized with one truss FEM element per each element (Fig. 28). Resulting charts are shown in Fig.29. The structure presents hardening behaviour as the fibres align to the load axis. The different animation frames are illustrated in Fig.30.

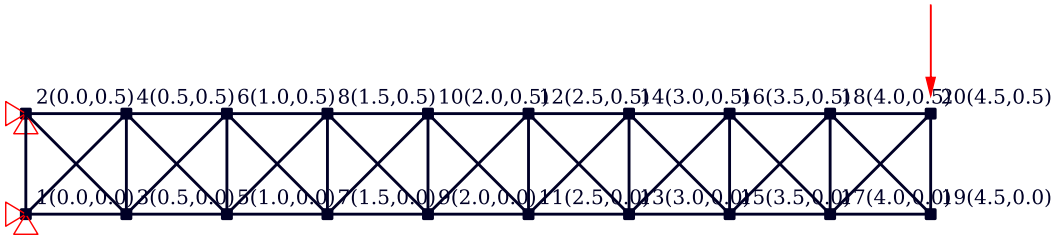

Figure 28: Cantilever truss: geometry, discretization, restraints and loading conditions

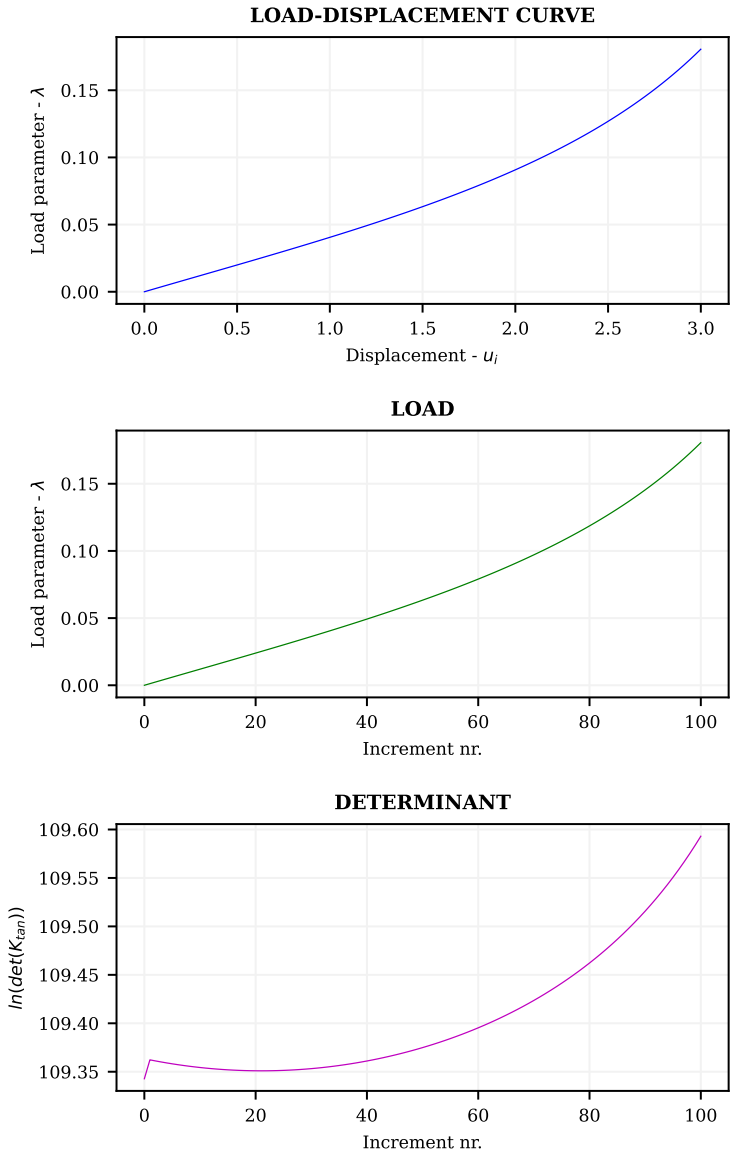

Figure 29: Cantilever truss: resulting charts

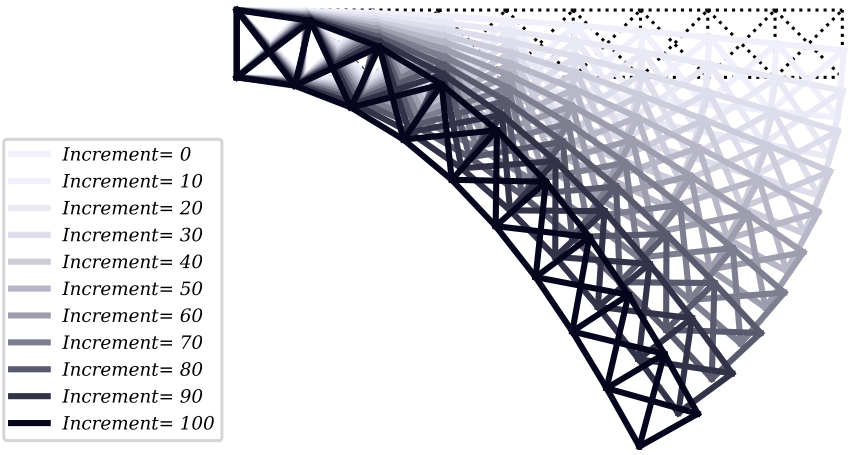

Figure 30: Cantilever truss: animation frames for different increments

6 Case studies: frame structures

6.1 Cantilever beam under increasing end moment

A first validation consists in verifying that under an increasing concentrated moment at end-node the structure will deform in a circular manner after a rotation of the end-node of $2 \cdot \pi$. This is a benchmark problem often exploited that can be found for example in []. Validation is also driven by commune own experience: trying to bend an elastic rod with a uniform moment results in a circular configuration. The structure consists in a single clamped rod with an end moment. The considered properties are:

Properties	
$L = 10\,m$	Rods length
$A = 1\,m^2$	Cross-sectional area
$J = 1\,m^4$	Second moment of section
$E = 1\,kN/m^2$	Elastic modulus
$\nu = 0.3$	Poisson ratio

The load is positioned at the end node. The structure is discretized with 40 Timoshenko elements. The geometry, constraints and load condition is depicted in Fig.31. A displacement control method is exploited. The controlled degree of freedom is the rotational of the end node. The number of increments is set to 100. The rotation is incremented from 0 to 2π. Results are shown in Fig.32. The deformed configurations for different increments are depicted in Fig.33. At the last increment the expected circular configuration is reached. The structure exhibits pure linear hardening behaviour and a load control method could had been used as well.

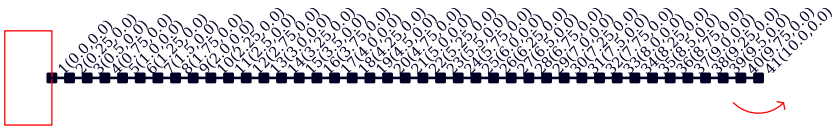

Figure 31: Cantilever beam with end moment: geometry, discretization, restraints and loading conditions

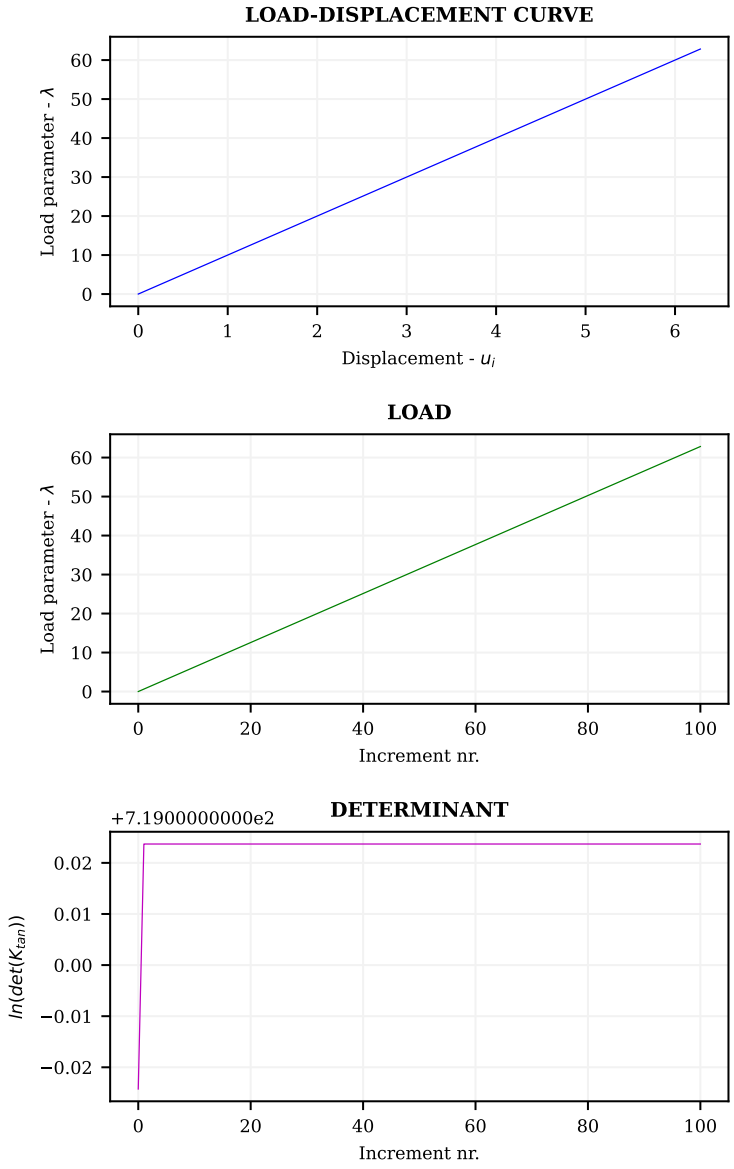

Figure 32: Cantilever beam with end moment: resulting charts

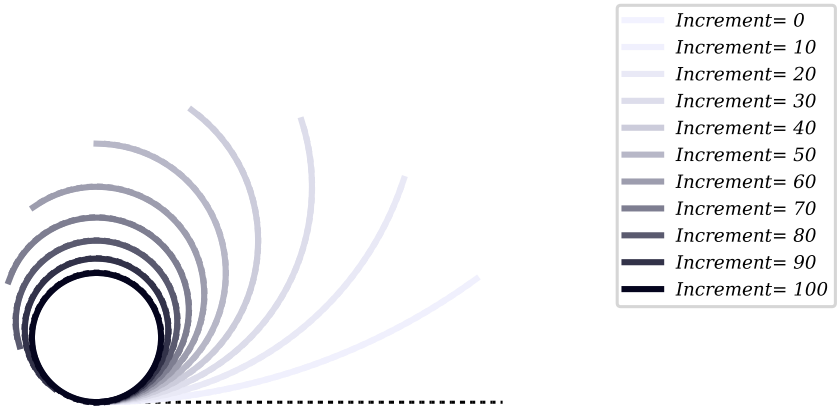

Figure 33: Cantilever beam with end moment: animation frames for different increments

6.2 Cantilever beam under increasing end force

A second case examines the same structure as the one studied in the previous case but under different load conditions. Here a point load is considered at the end node. A load control method is used in order to derive the beam response. Results are compared with those in [9] and [8]. Following properties are used:

Properties	
$L = 1\,m$	Column length
$A = 0.012\,m^2$	Cross-sectional area
$J = 1\,m^4$	Second moment of section
$E = 1\,kN/m^2$	Elastic modulus
$\nu = 0.3$	Poisson ratio

The structure is discretized with 20 elements as shown in Fig.35. Results are outlined in Fig.34 and the different configurations for increasing increments are shown in Fig.36. The results are in good agreement with those presented in [9, 8] and shown in figure Fig.37. As the load increases the structure fibres align to the load axis. The member starting from a purely flexural behaviour correctly captured in a small displacements elastic linear analysis, gradually enters in a membrane behaviour and starts working in a tensile state. This is the reason of the hardening behaviour of the structure. In an analogy this is also what happens when a polymeric structure is stressed under a tensile state as the polymeric chains forming the microstructure of the materials (firstly randomly oriented) start to align to the loading axis.

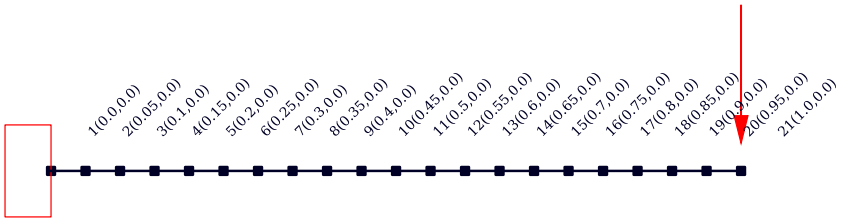

Figure 34: Cantilever beam with end force: geometry, discretization, restraints and loading conditions

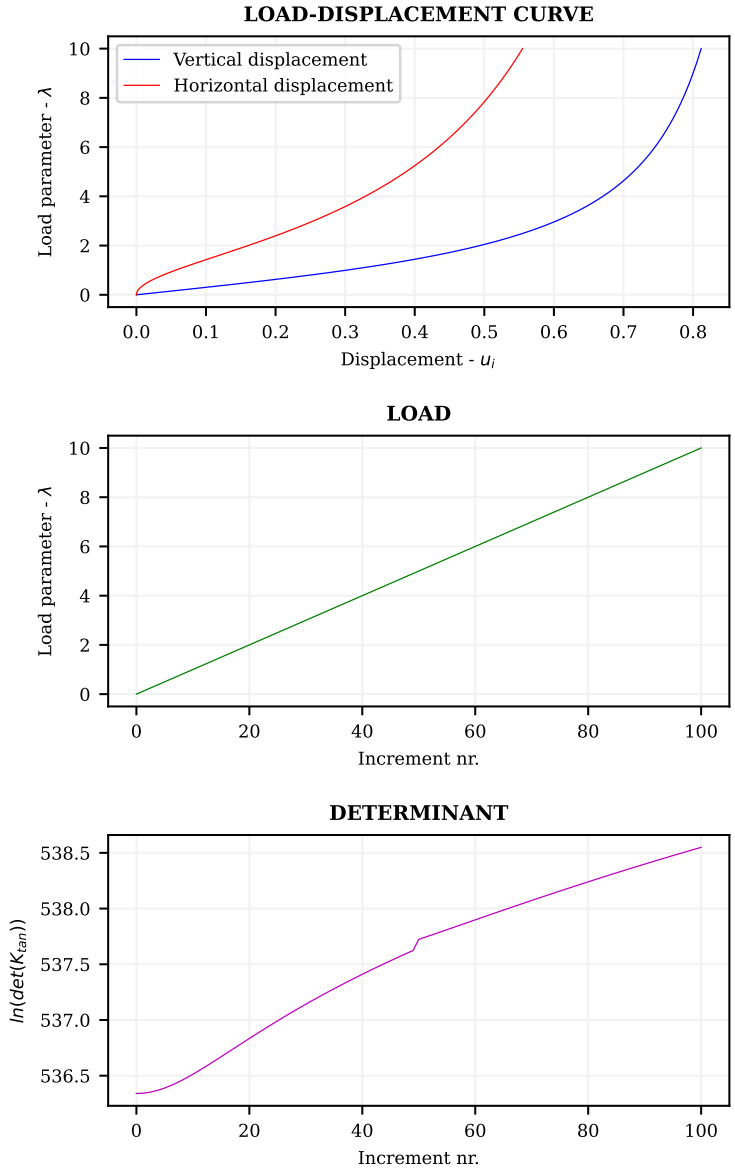

Figure 35: Cantilever beam with end force: resulting charts

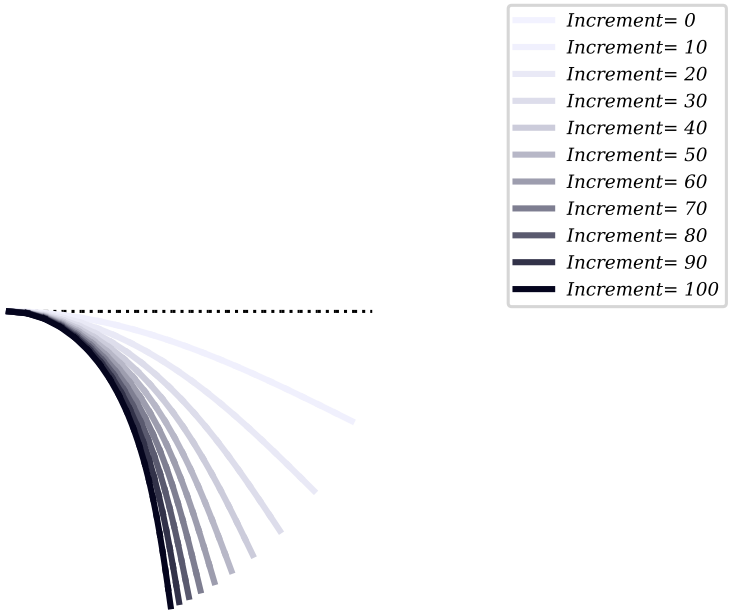

Figure 36: Cantilever beam with end force: animation frames for different increments

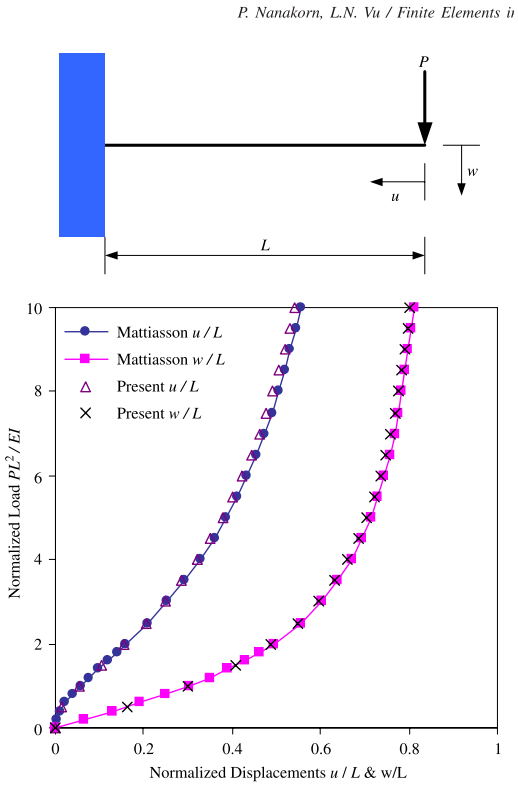

Figure 37: Cantilever beam with end force: literature results from [9, 8]

6.3 Buckling of a pinned-roller compressed column

One of the common applications of a nonlinear analysis is to examine the buckling and post-buckling behaviour of compressed members. A typical problem is represented by the compressed rod restrained by pinned-roller configuration. In this case an analytical comparison is possible. The analytical solution can be derived by mean of elliptic integrals. Image Fig.38 (source [6]) represents the load-end rotation behaviour of the column derived analytically. Black circle points are bifurcation points.

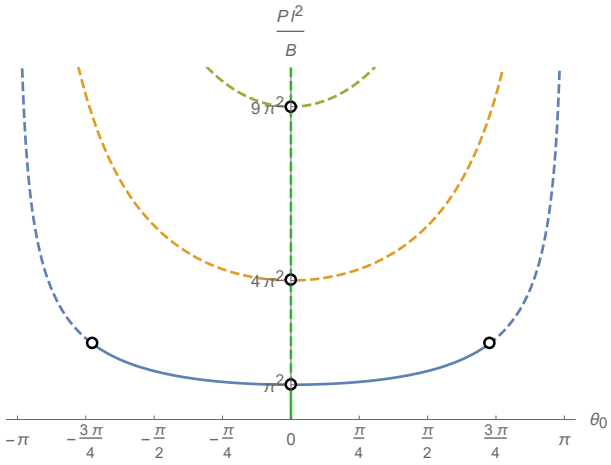

Figure 38: Buckling of column, analytical solution (from [6])

Following properties are considered:

Properties	
$L = 10\,m$	Column length
$A = 100\,m^2$	Cross-sectional area
$J = 10\,m^4$	Second moment of section
$E = 1\,kN/m^2$	Elastic modulus
$G = 1\,kN/m^2$	Shear modulus

The exploited control method is a load-control method. This is possible because the post buckling behaviour has a pure hardening branch. No softening behaviour occurs. The analysed load multiplier range will be changed in relation with the examined buckling mode.

The buckling mode is triggered by giving the column a sinusoidal imperfection:

$$y(x) = e_0 \cdot \sin\left(m \cdot \pi x / L \cdot\right) \tag{125}$$

Here x and y are the nodal coordinates, with x ranging in 0 to L. Where $e_0 =$ and the parameter m is an integer that refers to the mode that needs to be triggered. Thus for buckling mode 1 $m = 1$ will be set, for buckling mode 2 $m = 2$ and so on. For each buckling mode the relevant critical load can be computed exploiting the Euler formula:

$$P_{cr} = \frac{m^2\pi^2}{L^2} \cdot EJ \tag{126}$$

By substituting the used values:

$$P_{cr} = \frac{m^2\pi^2}{100} \cdot 10 \approx m^2 \tag{127}$$

The structure is discretized with a fine mesh of 100 Timoshenko elements.

6.3.1 Buckling mode 1

The first buckling mode is triggered with an imperfection with parameter $e_0 = 0.05$. The parameter $m = 1$ is used. A divergence instability is expected due to the small imperfection, with the main load path lying progressively to the bifurcated path. The meshing and load and restraint condition are shown in figure Fig.39. Critical load is expected at around $P_{cr} = m^2 = 1\ kN$. the results and the deformed configurations for different increments are shown respectively in figures Fig.40 and Fig.41. The load increases linearly for increasing increments as a consequence of the load control method choice. The determinant appears having a first decrease in correspondence of $\lambda = P_{cr} \approx 1\ kN$. A second sharper decrease happens at around $\lambda = 2\ kN$. There, probably the bifurcation showed before (Fig.38) happens. This was however not further examined in this study. Note that before buckling arises, a small deformation appears for loads smaller than P_{cr}. This is consequence of the finite axial stiffness EA of the column.

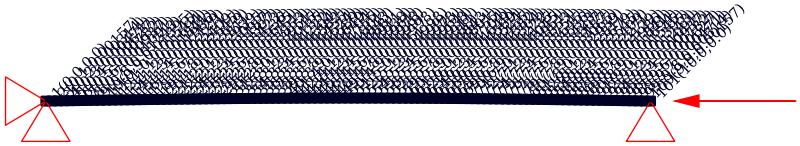

Figure 39: Column buckling mode 1: geometry, discretization, restraints and loading conditions

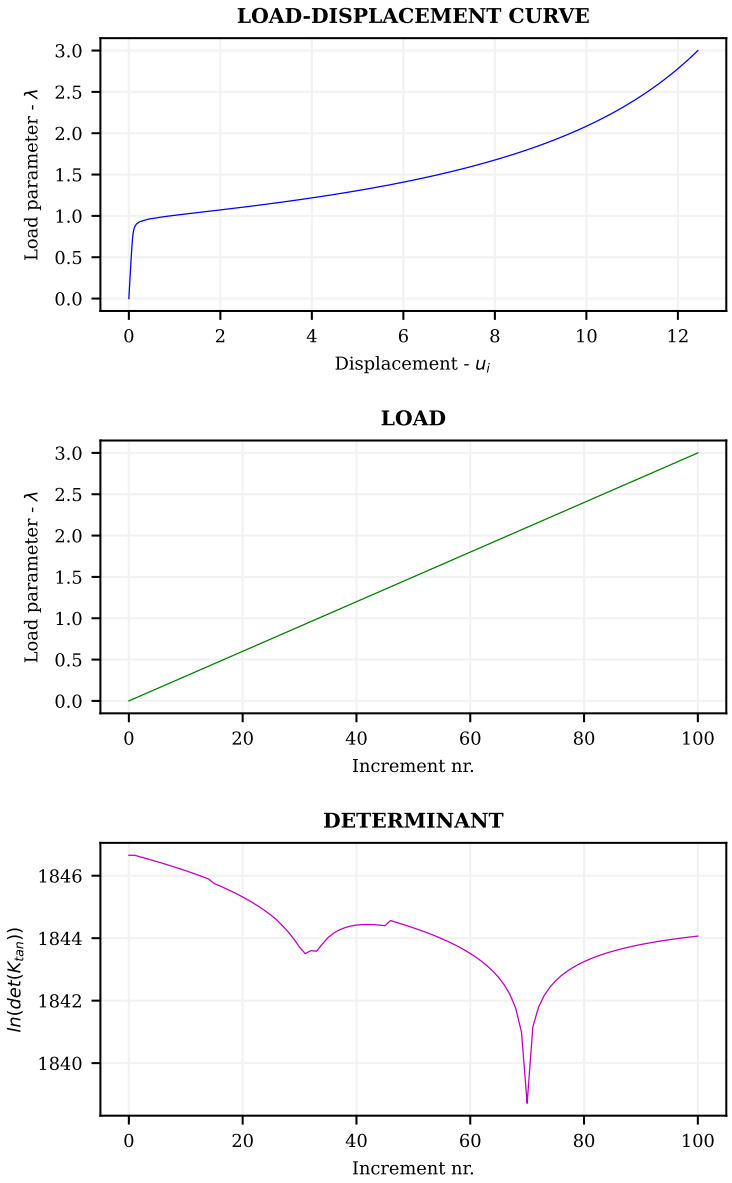

Figure 40: Column buckling mode 1: resulting charts

Figure 41: Column buckling mode 1: animation frames for different increments

6.3.2 Buckling mode 2

The second buckling mode is triggered with an imperfection with parameter $e_0 = 0.05$. The parameter $m = 2$ is used. A divergence instability is expected due to the small imperfection, with the main load path lying progressively to the bifurcated path. The meshing and load and restraint condition are shown in figure Fig.42. Critical load is expected at around $P_{cr} = m^2 = 4\,kN$. the results and the deformed configurations for different increments are shown respectively in figures Fig.43 and Fig.44. The load increases linearly for increasing increments as a consequence of the load control method choice. The determinant appears having a first decrease in correspondence of $\lambda = P_{cr}(m = 1) \approx 1\,kN$. This is because the first buckling mode (studied before) is skipped by the structure. A second minimum appears for $\lambda = P_{cr}(m = 2) \approx 4\,kN$. This is the second buckling load. A third sharper decrease happens at around $\lambda = 8\,kN$. This was however not further examined in this study. The same further considerations can be done as the mode 1 case.

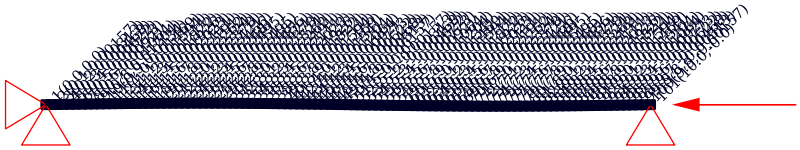

Figure 42: Column buckling mode 2: geometry, discretization, restraints and loading conditions

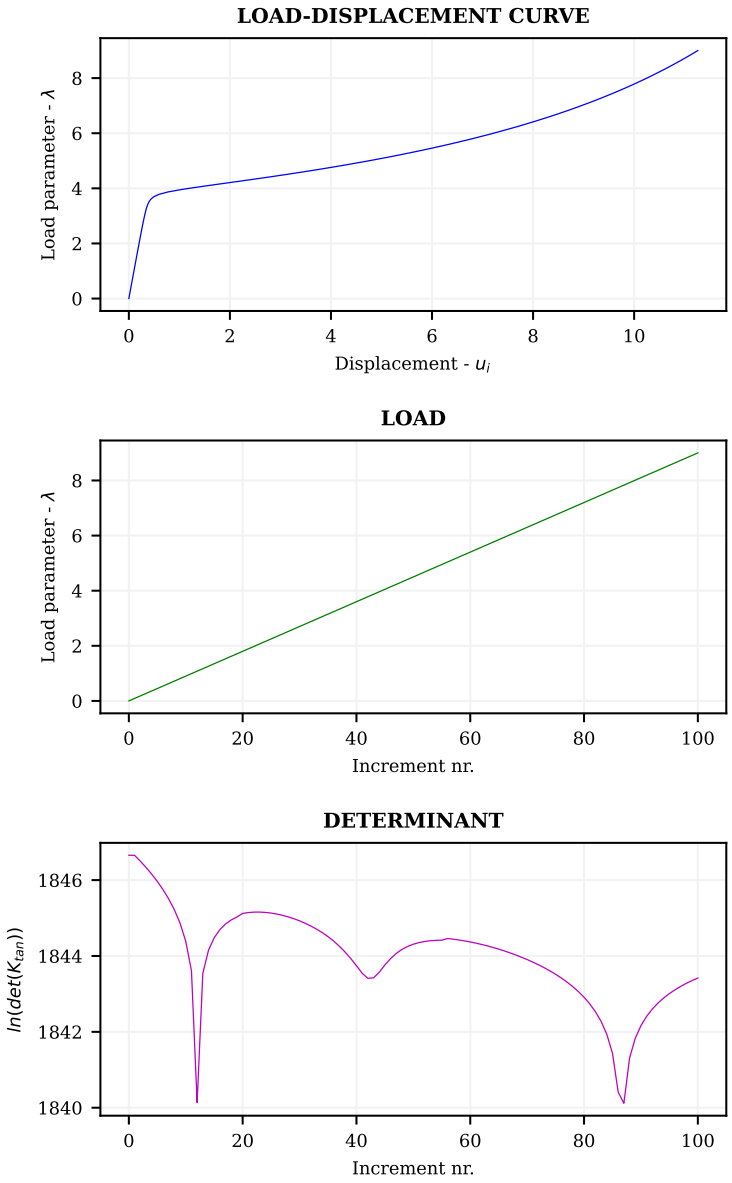

Figure 43: Column buckling mode 2: resulting charts

Figure 44: Column buckling mode 2: animation frames for different increments

6.3.3 Buckling mode 3

The third buckling mode is triggered with an imperfection with parameter $e_0 = 0.05$. The parameter $m = 3$ is used. A divergence instability is expected due to the small imperfection, with the main load path lying progressively to the bifurcated path. The meshing and load and restraint condition are shown in figure 45. Critical load is expected at around $P_{cr} = m^2 = 9kN$. The results and the deformed configurations for different increments are shown respectively in figures Fig.46 and Fig.47. The load increases linearly for increasing increments as a consequence of the load control method choice. The determinant clearly decreases with the structure passing at the load levels of the previous two buckling modes. Another sharp decrease is observed in the post-buckling branch.

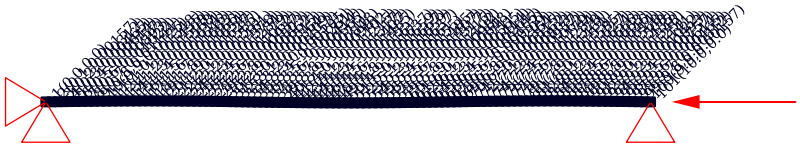

Figure 45: Column buckling mode 3: geometry, discretization, restraints and loading conditions

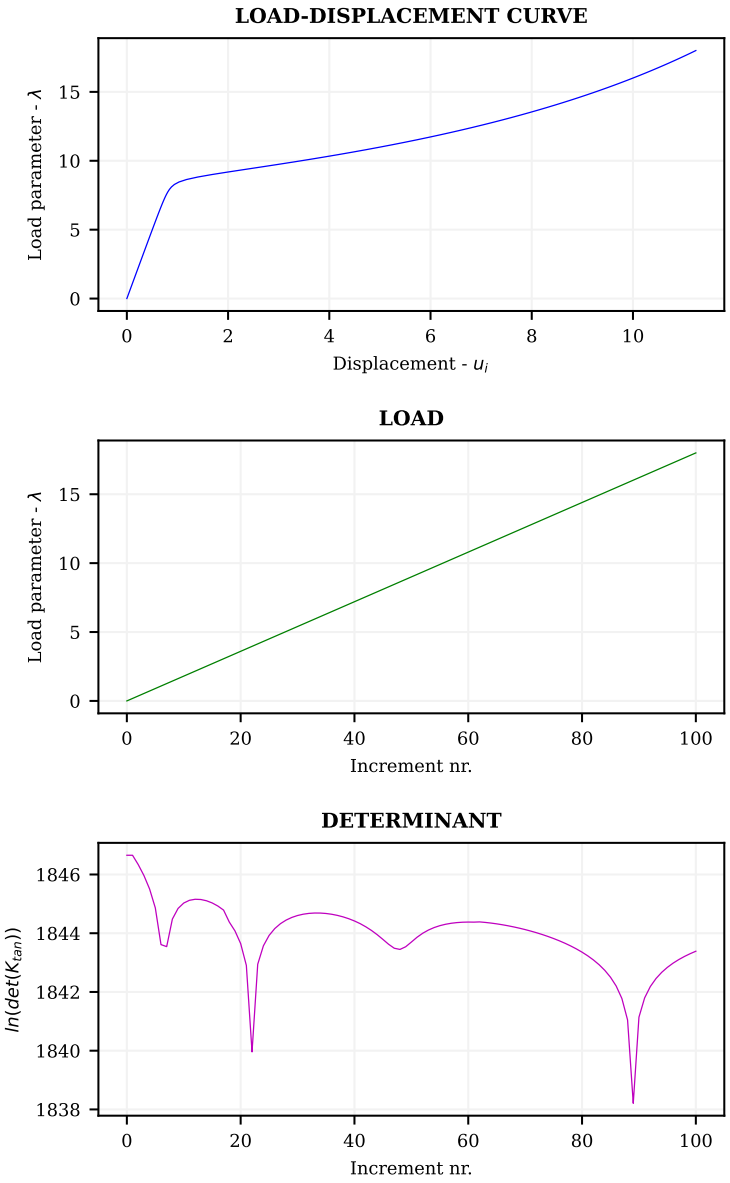

Figure 46: Column buckling mode 3: resulting charts

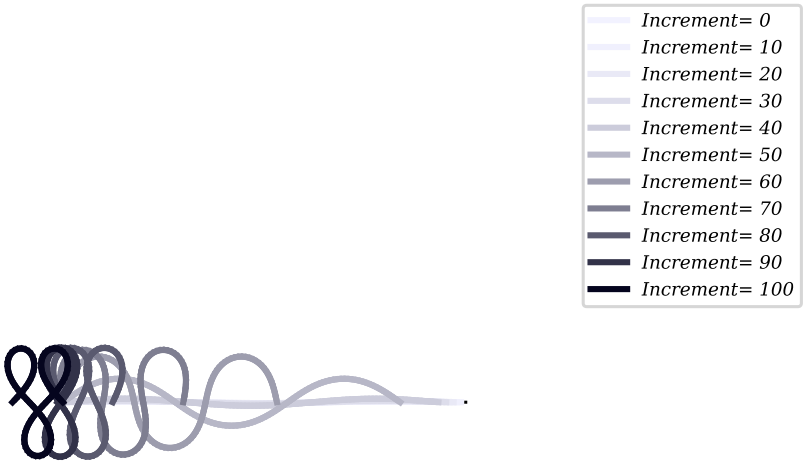

Figure 47: Column buckling mode 3: animation frames for different increments

6.4 Lee's frame

In this section the resolution of a benchmark problem often referred in literature as "Lee's frame" is solved. Reference to the results obtained in [7] is made. This work was validated on the basis of [12] that contains other useful reference. Further reference can be made to [11]. The structure consists in two perpendicular elastic rods. These have the following properties:

Properties	
$L = 120\,cm$	Rods length
$A = 6\,cm^2$	Cross-sectional area
$J = 2\,cm^4$	Second moment of section
$E = 720\,kN/cm^2$	Elastic modulus
$\nu = 0.3$	Poisson ratio

The load is positioned at a distance of $24\,cm$ from the node connecting the two rods. In analogy with the work of [7], the structure is discretized with ten Timoshenko elements for each rod. The geometry, constraints and load condition is depicted in Fig.48. The monitored degree of freedom is the vertical displacement of the loaded node. The structure is solved exploiting the implemented Riks arclength pathfollowing method with a single step and a total of 500 increments. The value of arc length increment is set to $\Delta s = 1.3$. Results are shown in Fig.49. The deformed configurations for different increments are depicted in Fig.50. Results are in good agreement with the reference works of [7] and [12]. The structure exhibits snap back behaviour with multiple turning points and critical points. Note that the determinant presents significant reduction in correspondence of the critical points.

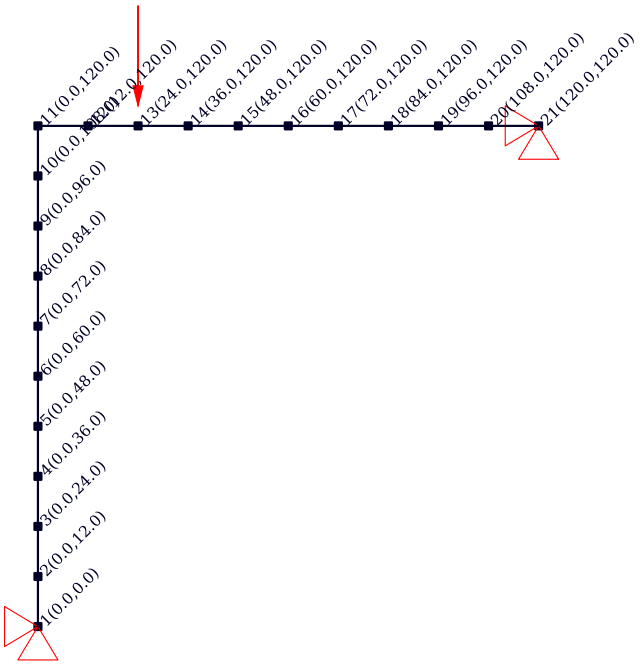

Figure 48: Lee's frame: geometry, discretization, restraints and loading conditions

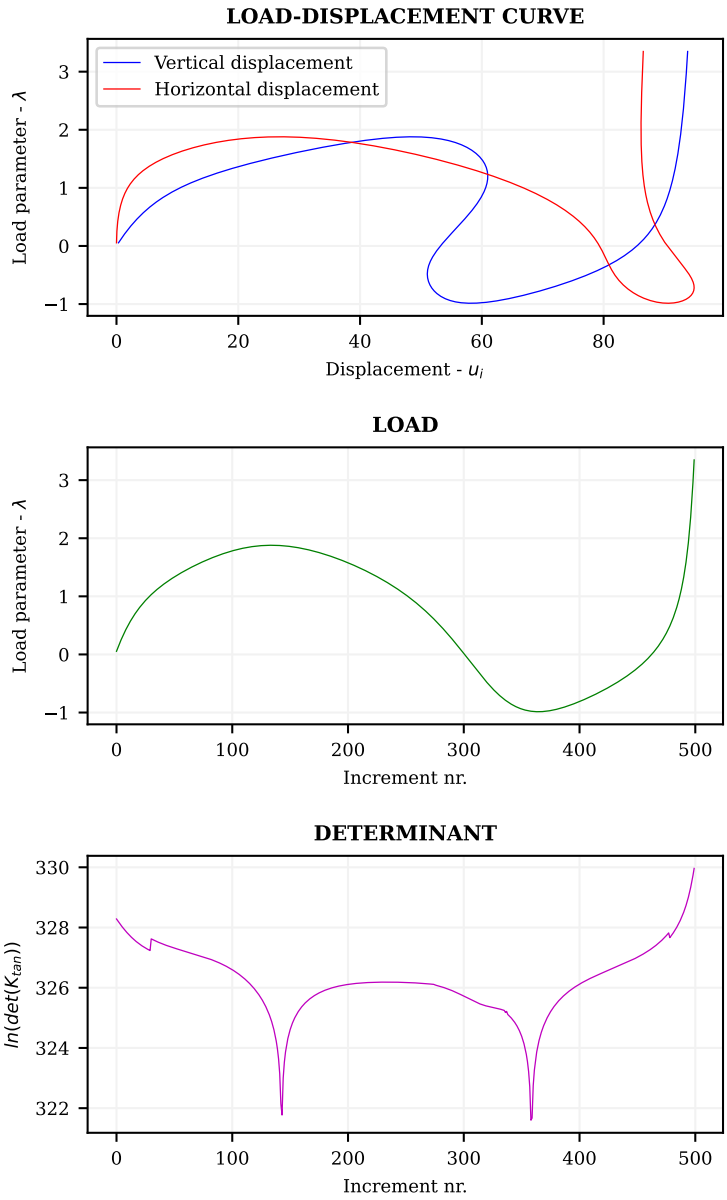

Figure 49: Lee's frame: resulting charts

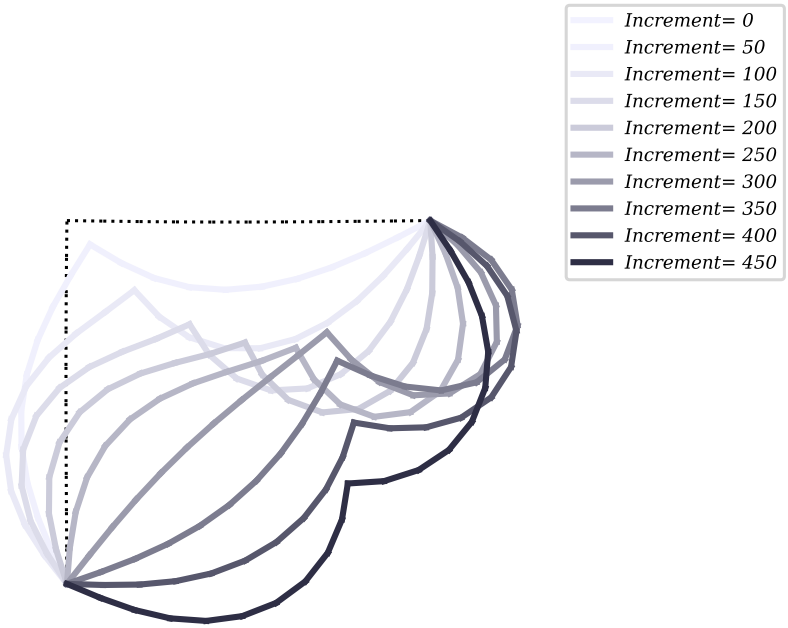

Figure 50: Lee's frame: animation frames for different increments

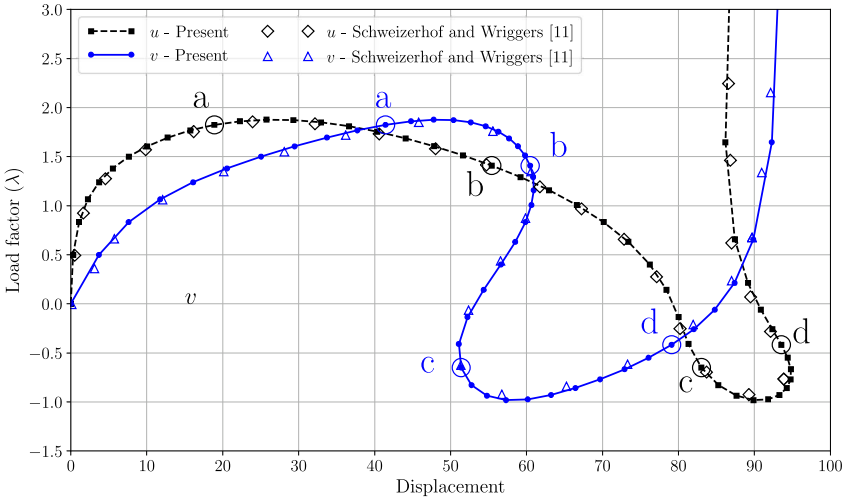

Figure 51: Lee's frame: literature results from [7]. Good agreement with the present study results can be seen from comparison of the vertical displacement evolution

6.5 Frame lateral instability

A frame composed of two vertical columns and one horizontal beam is studied. Base restraints consist in pinned ends. The purpose is to study the frame's lateral instability behaviour and the post-buckling response. In order to trigger the lateral instability a small horizontal load is set. The lateral instability consists in a bifurcated path respect to the main load path. The main load path will also be shown. The Riks arclenght control stratecy is exploited. The load is applied at mid-span of the beam. Following properties are exploited:

Properties	
$L = 10000\,mm$	Rods length
$A = 100\,cm^2$	Cross-sectional area
$J = 10000\,cm^4$	Second moment of section
$E = 200000\,MPa$	Elastic modulus
$\nu = 0.3$	Poisson ratio

Geometric configuration, load disposition and meshing are shown in figure **??**. The mesh consists in 30 Timoshenko elements per each rod. The load path is derived with 500 increments.

6.5.1 Lateral instability

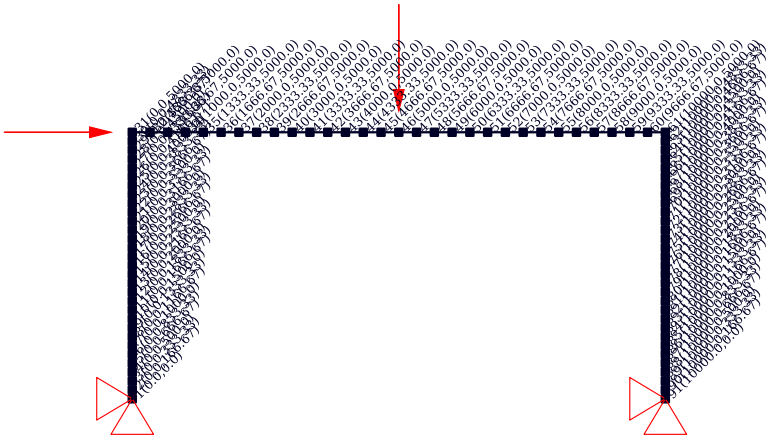

Figure 52: Portal lateral instability: geometry, discretization, restraints and loading conditions

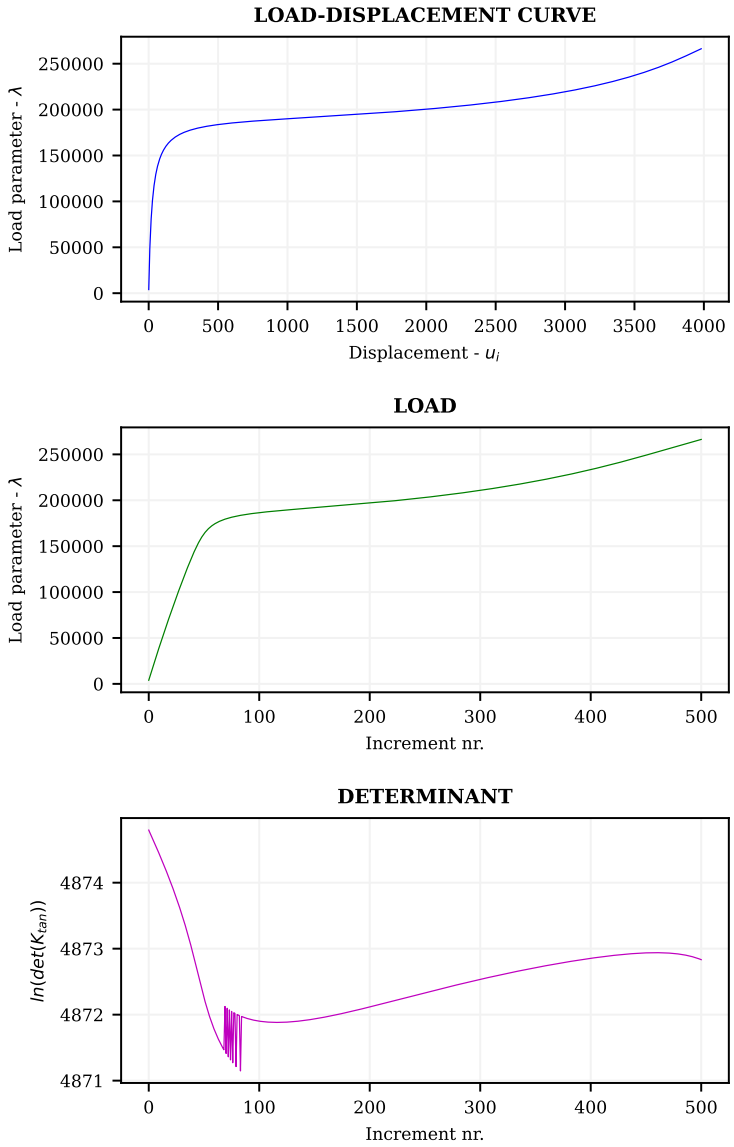

Figure 53: Portal lateral instability: resulting charts

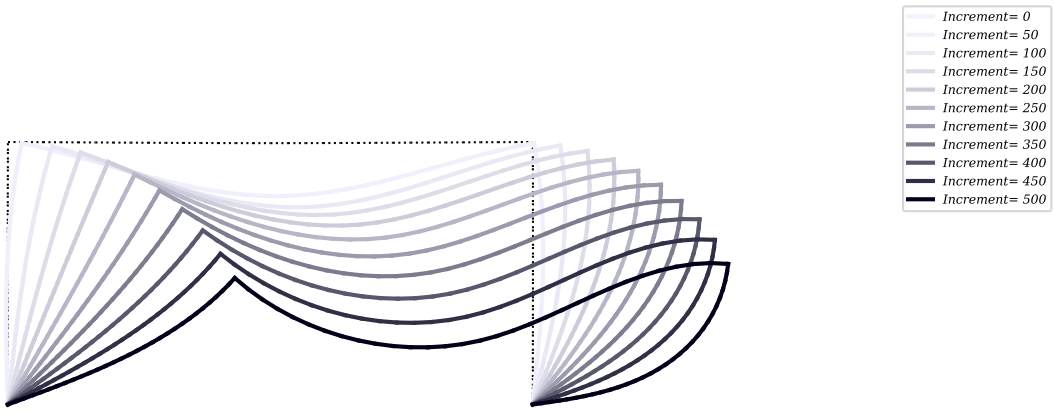

Figure 54: Portal lateral instability: animation frames for different increments

6.5.2 Main load path

In this case no lateral force is applied in order to trigger the lateral instability. Results are shown in figures Fig.56 and Fig.57. The structure exhibits a softening behaviour after a critical load. Note that the determinant decreases in correspondence of the load level at which the previous studied case was exhibiting the lateral instability starting to take the bifurcated path.

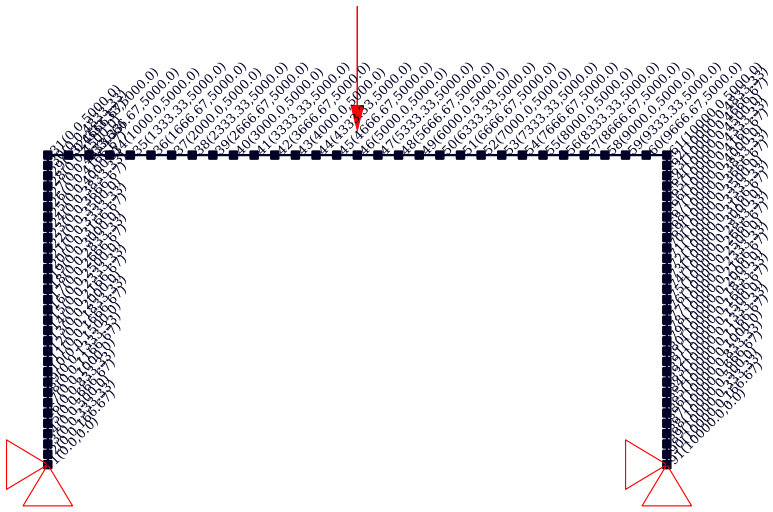

Figure 55: Portal main load path: geometry, discretization, restraints and loading conditions

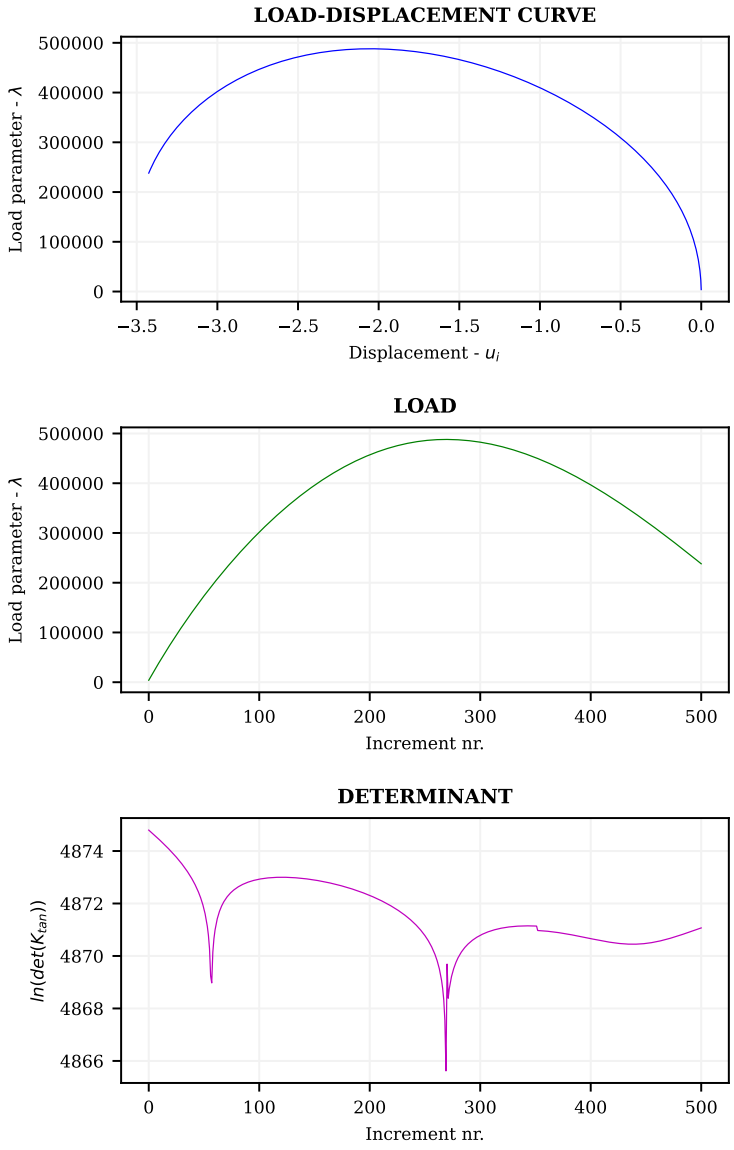

Figure 56: Portal main load path: resulting charts

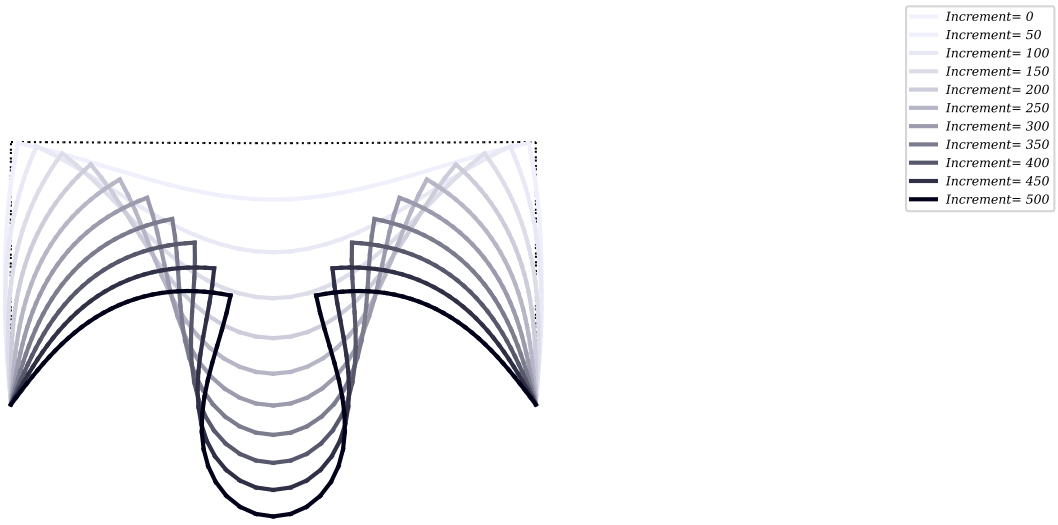

Figure 57: Portal main load path: animation frames for different increments

6.6 Shallow arch

A shallow arch is resolved. The arch is loaded with a vertical force impressed at the top node. Following properties are exploited:

Properties	
$R = 100\,cm$	Radius
$\theta = 2 \cdot 20.3 = 40.6$	Arch angle
$J = 1\,cm^4$	Second moment of section
$A = 1\,cm^2$	Second moment of section
$E = 200\,N/cm^2$	Elastic modulus
$\nu = 0.5$	Poisson ratio

6.6.1 Main path

The number of elements used is 50 Timoshenko elements (Fig.58). The structure exhibits snap through behaviour Fig.59. The animation frames can be appreciated in Fig.60. The determinant has a reduction on the first loading branch before the critical point occurs. This consists in a bifurcation point which is further examined below.

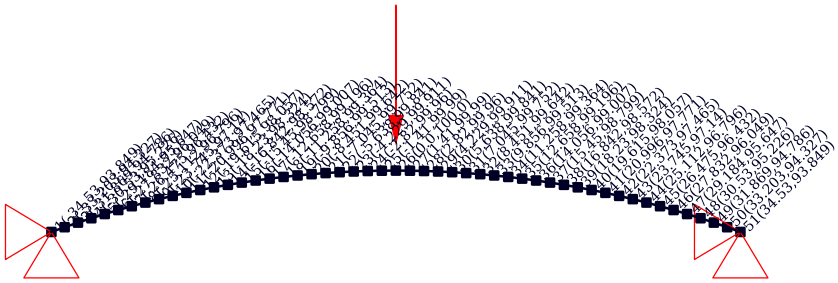

Figure 58: Pinned shallow arch main path: geometry, discretization, restraints and loading conditions

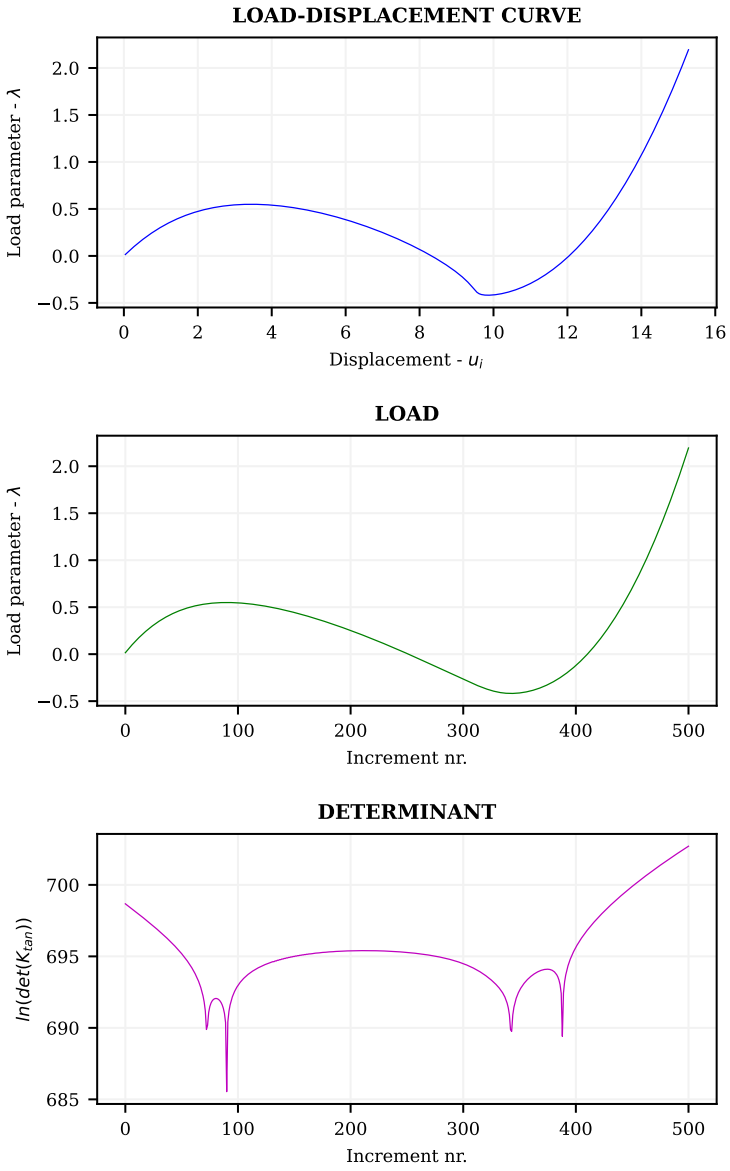

Figure 59: Pinned shallow arch main path: resulting charts

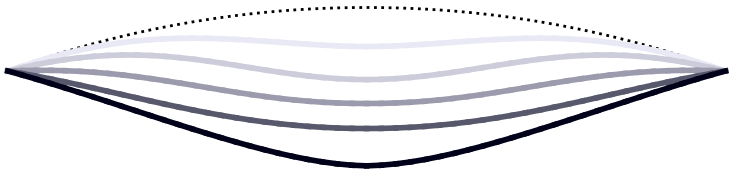

Figure 60: Pinned shallow arch main path: animation frames for different increments

6.6.2 Bifurcated path

The structure bifurcated path is triggered with a small point moment at the tip of the arch structure (Fig.61). Results are shown in Fig.62. The animation frames can be appreciated in Fig. 63.

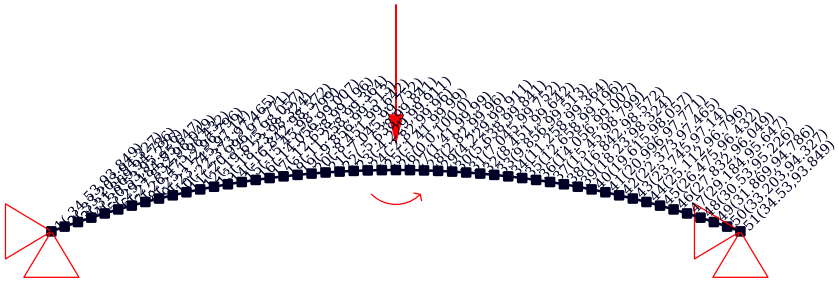

Figure 61: Pinned shallow arch bifurcated path: geometry, discretization, restraints and loading conditions

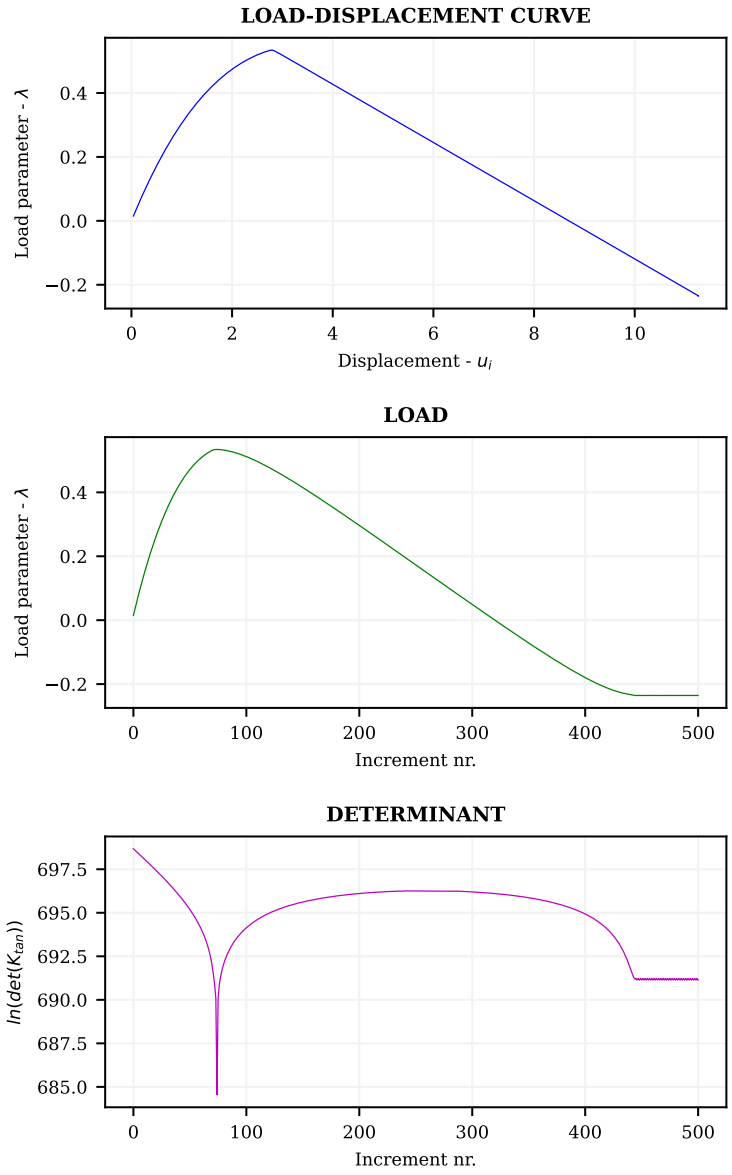

Figure 62: Pinned shallow arch bifurcated path: resulting charts

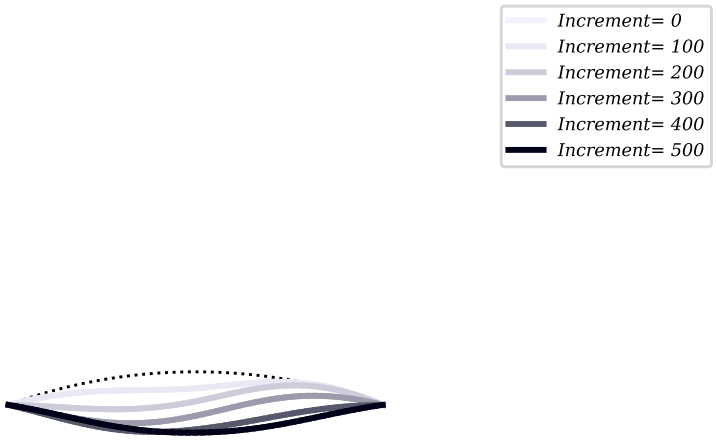

Figure 63: Pinned shallow arch bifurcated path: animation frames for different increments

6.7 Deep arch

A round arch structure is studied here. Reference to results contained in [7, 14] is made.

Properties	
$R = 127\ cm$	Radius
$\theta = 2 \cdot 90 = 180$	Arch angle
$J = 41.62\ cm^4$	Second moment of section
$A = 64.52\ cm^2$	Cross section area
$E = 0.1378\ N/cm^2$	Elastic modulus
$\nu = 0.5$	Poisson ratio

The structure (Fig.64) is discretized with 50 Timoshenko elements according with [7, 14]. The structure is solved with Arclength Riks method. The number of increments is 7500. Results are shown in Fig.65. The structure performs different loops in the load-deflection plane. The responce is in perfect accordance with [7] (Fig.67).

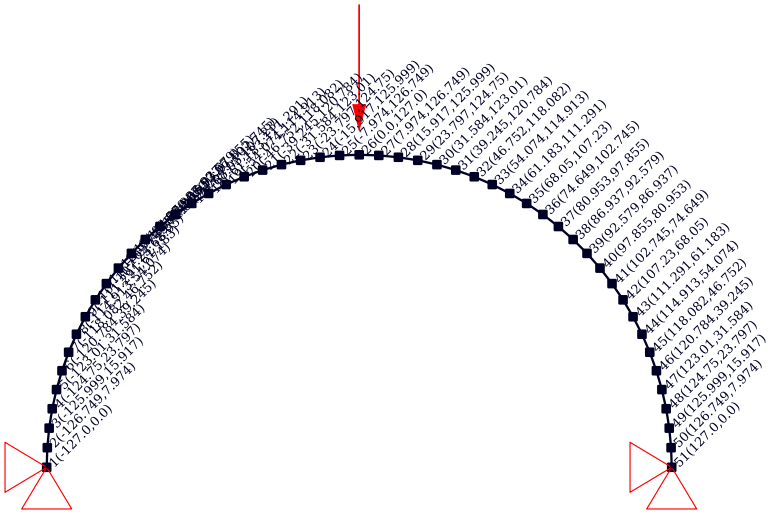

Figure 64: Pinned round arch: geometry, discretization, restraints and loading conditions

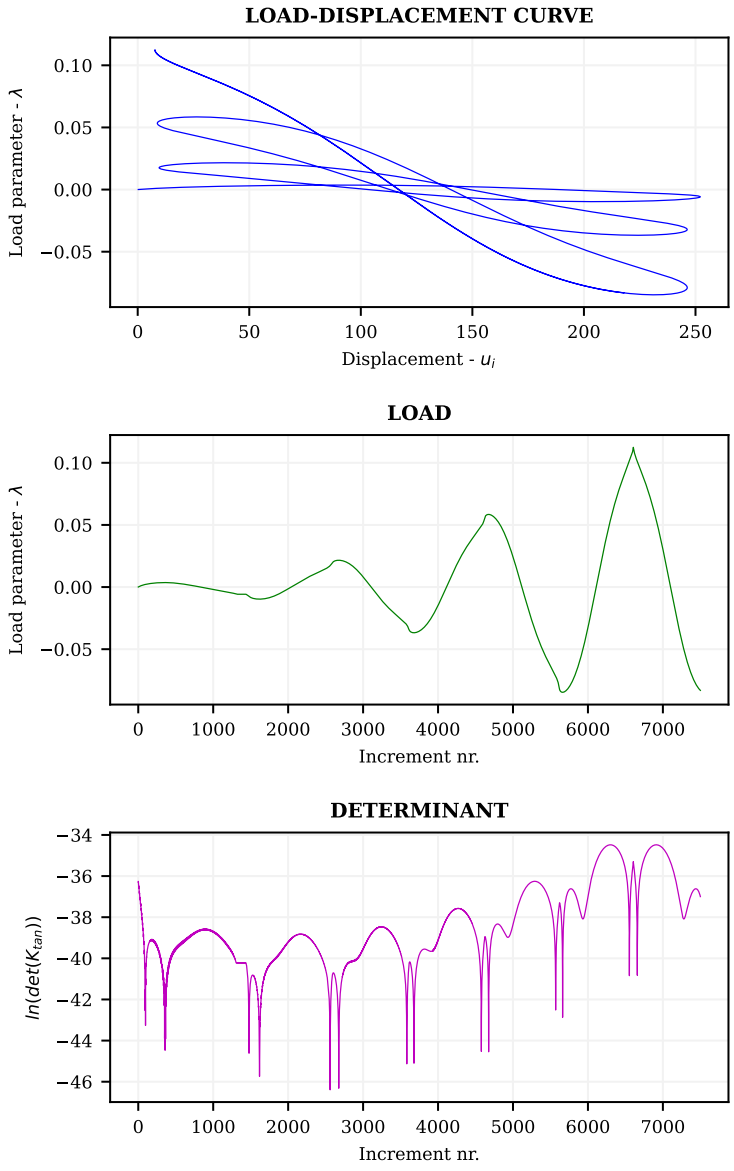

Figure 65: Pinned round arch: resulting charts

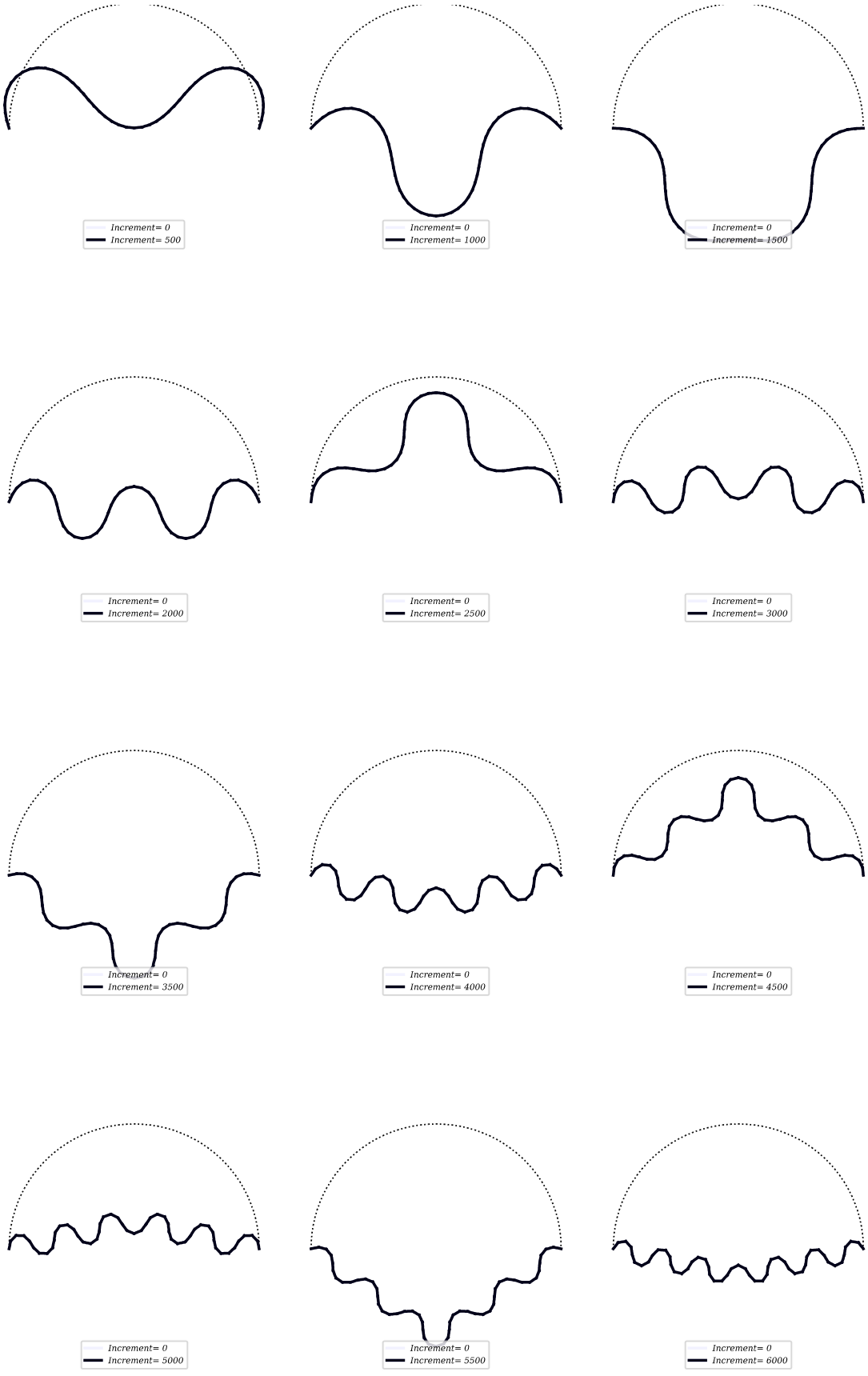

Figure 66: Pinned round arch: animation frames for different increments

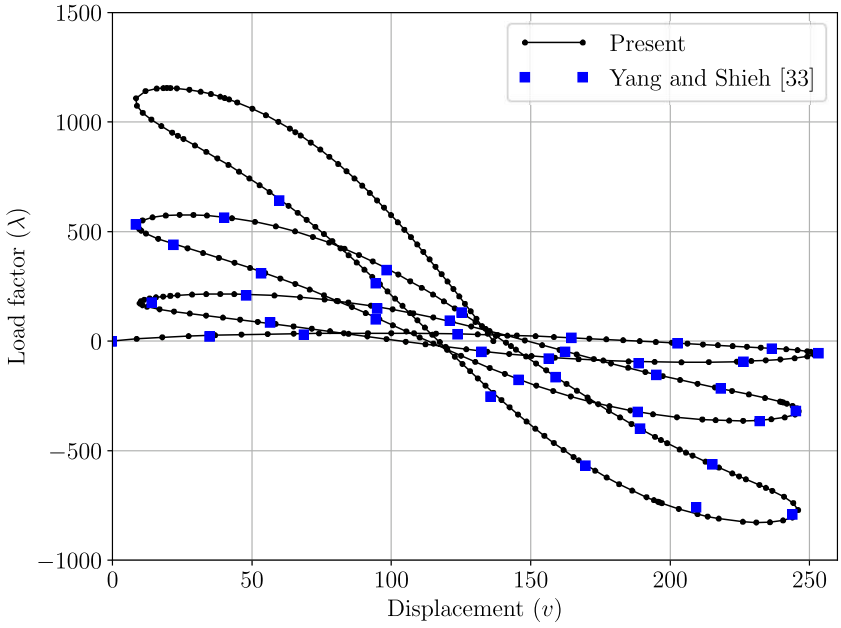

Figure 67: Pinned round arch: literature results from [7]

6.8 Elastic circle

The nonlinear analysis on the response of a vertically loaded elastic circle is shown. The structure is restrained at the bottom node with an encastre. At the top node a vertical point load is applied. In addition a small moment is applied in the same node in order to derive the response of the bifurcated load path. the properties are set as follows:

Properties	
$R = 10\,cm$	Radius
$A = 0.1\,cm^2$	Cross-sectional area
$J = 0.1\,cm^4$	Second moment of section
$E = 1000\,N/cm^2$	Elastic modulus
$\nu = 0.3$	Poisson ratio

The Riks arclength method is used. The arclength parameter Δs is set to 1.The circular structure is discretized into 100 Timoshenko elements.

6.8.1 Bifurcated path

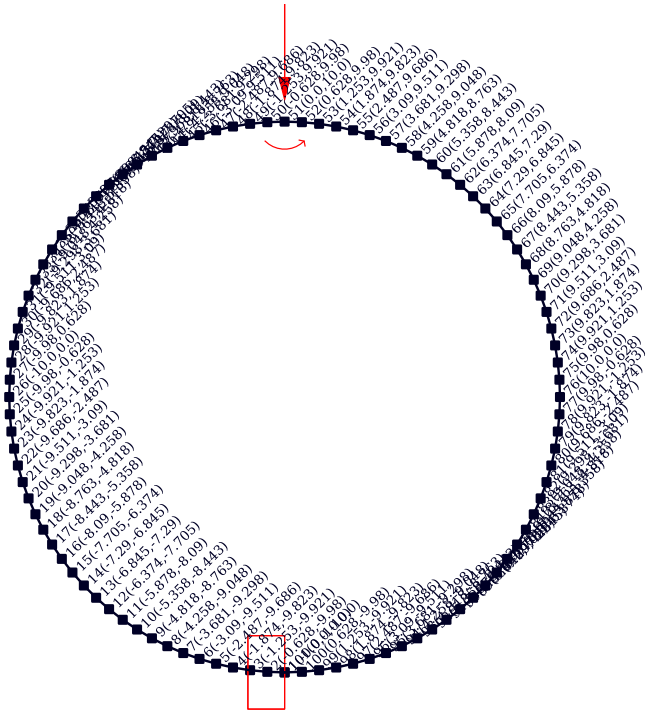

Figure 68: Circular structure main load path: geometry, discretization, restraints and loading conditions

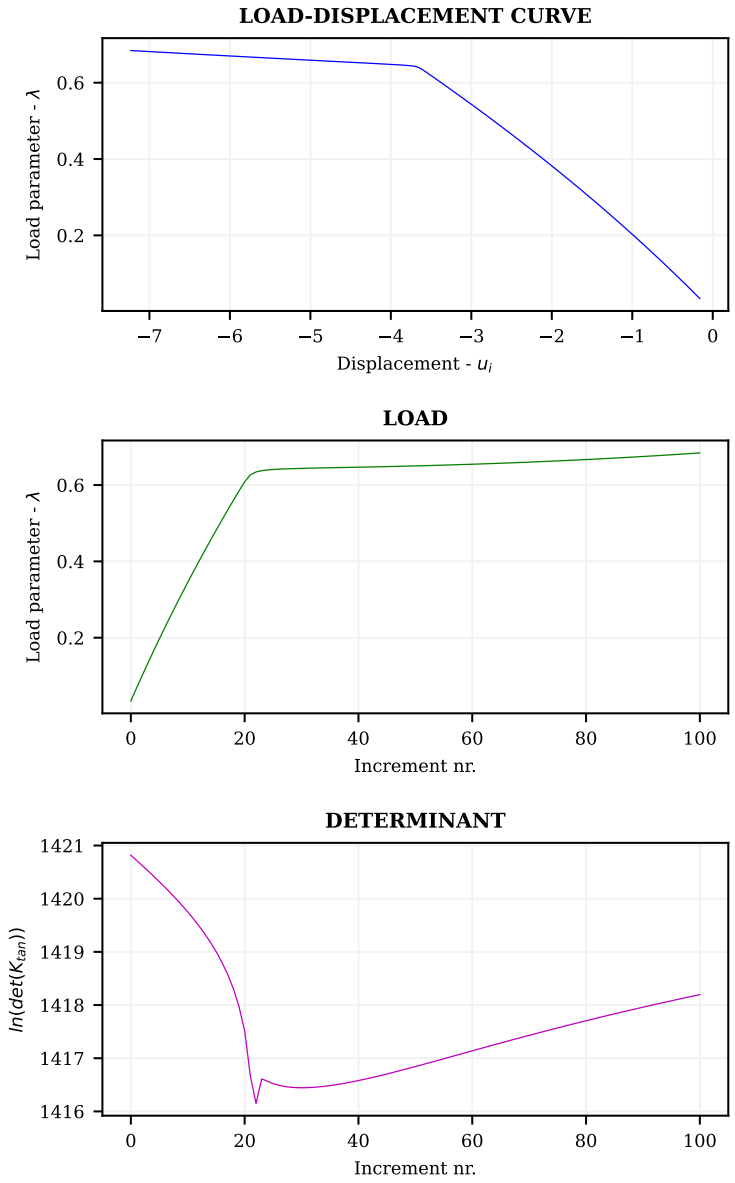

Figure 69: Circular structure main load path: resulting charts

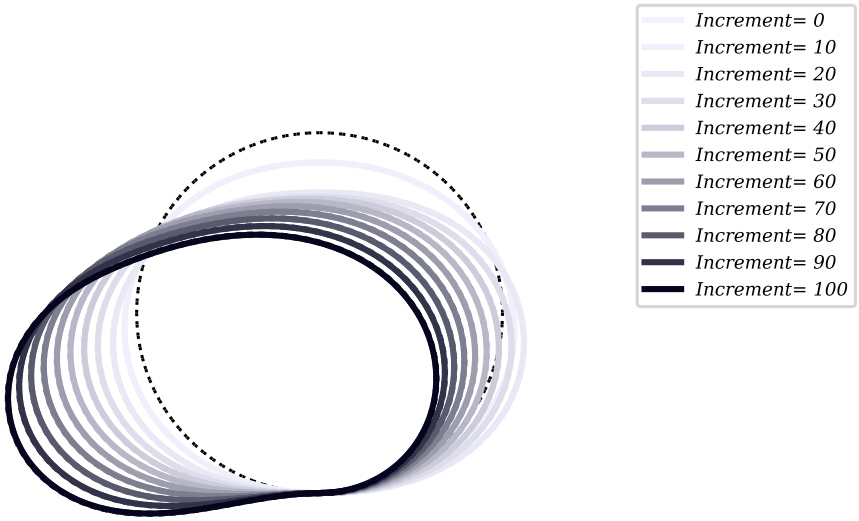

Figure 70: Circular structure main load path: animation frames for different increments

6.8.2 Main path

In order to follow the main path the small tip moment is removed. Just the vertical load is applied to the top structure node. The load-vertical displacement response is almost linear. A sharp decrease of the determinant value occurs in the region where the bifurcated path joints the main path.

6.8.3 Bifurcated path

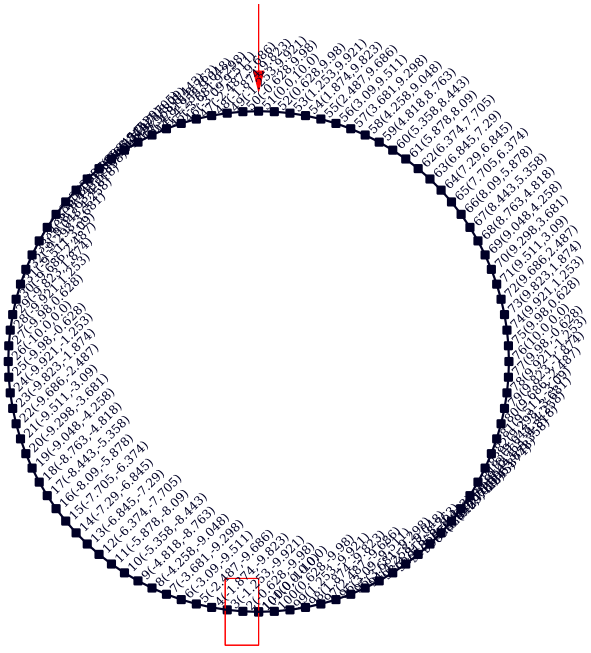

Figure 71: Circular structure bifurcated path: geometry, discretization, restraints and loading conditions

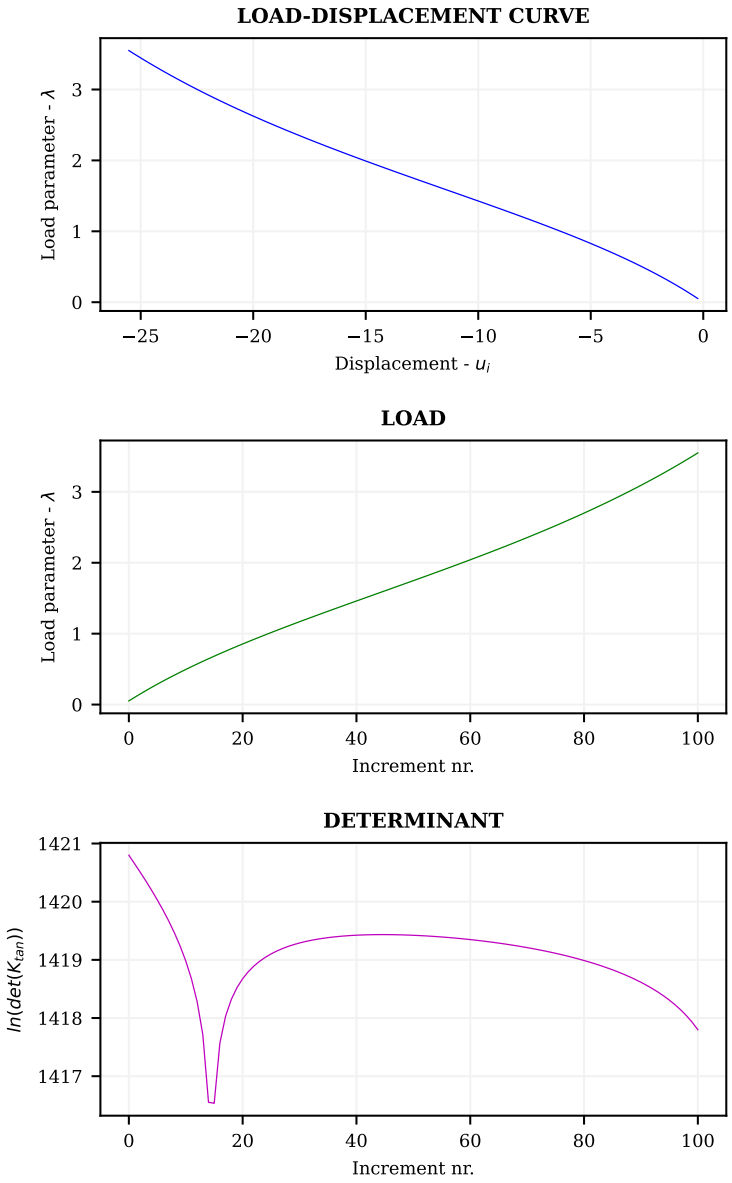

Figure 72: Circular structure bifurcated path: resulting charts

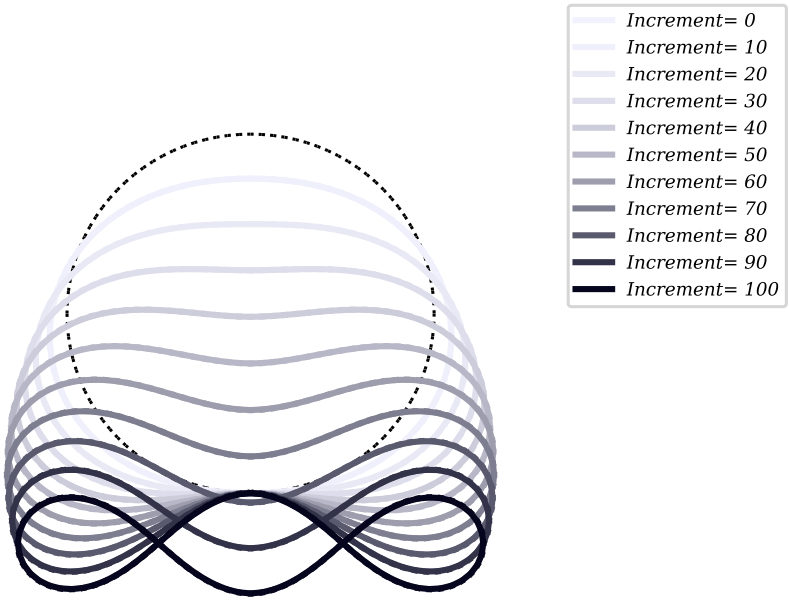

Figure 73: Circular structure bifurcated path: animation frames for different increments

6.9 Cellular material sample

The mechanical behaviour of a cellular material structure is tested. The shape is inspired by the TAVR valves in biomedical applications. The structure consists in a 6×5 rows per columns matrix of cells. The arranged cells have sinusoidal shaped walls and each cell consists in $6\mu m \times 2\mu m$ dimensions. Cellular walls are modelled with Timoshenko beam type elements. The structure is clamped ad one side and increasing forces are applied at the other side end-nodes. Both solicitations of tensile loading and compression will be tested. The following properties are used:

Properties	
$A = 0.1\ \mu m^2$	Cross sectional area
$J = 0.1\ \mu m^4$	Second moment of section
$E = 1000\ N/\mu m^2$	Elastic modulus
$\nu = 0.5$	Poisson ratio

The structure is discretized with 432 nodes and 540 Timoshenko elements. A Riks arclength method is used with Δs and a total number of increments of 100 is used.

6.9.1 Tensile loading

Tensile forces are applied at end nodes. The structure is shown in Fig.74. Resulting charts can be appreciated in Fig.75. The animation frames for different time increments are illustrated in Fig.76. The structure has an hardening behaviour as the fibers align to the load axis.

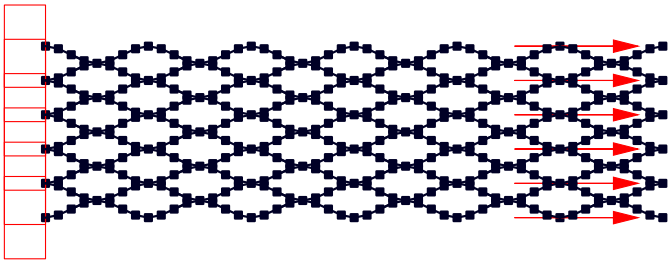

Figure 74: Cellular material sample structure: geometry, discretization, restraints and loading conditions

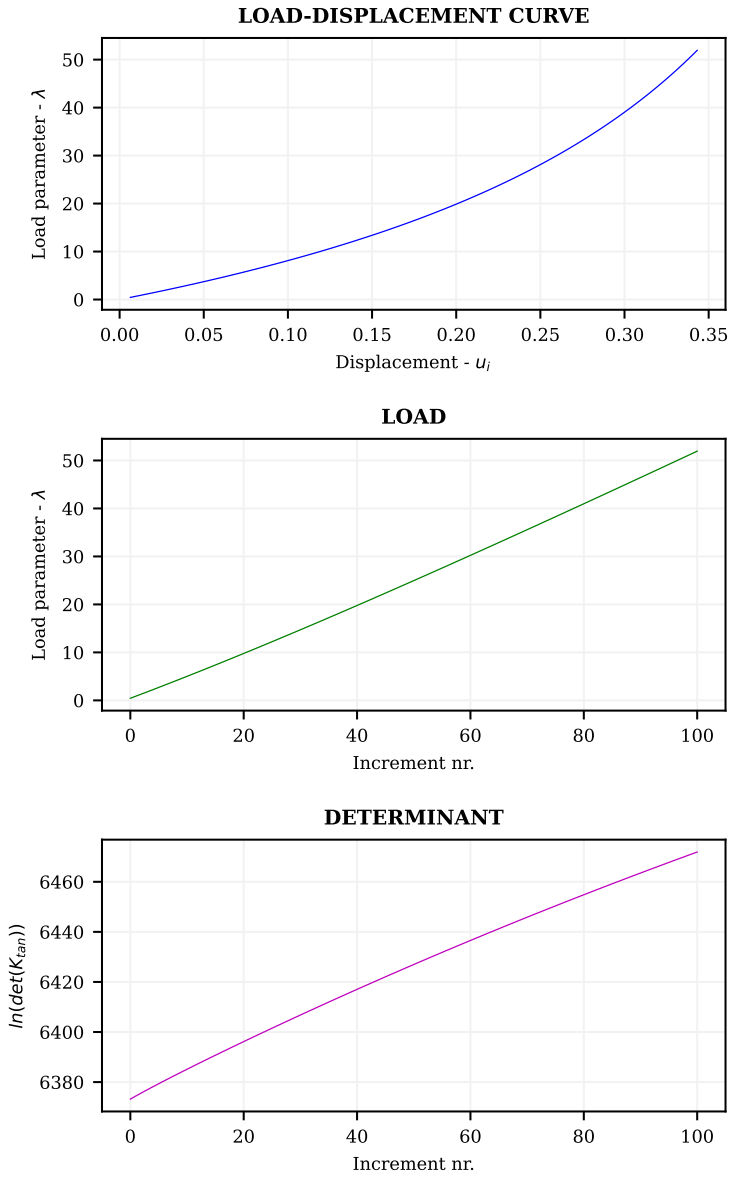

Figure 75: Cellular material sample structure: resulting charts

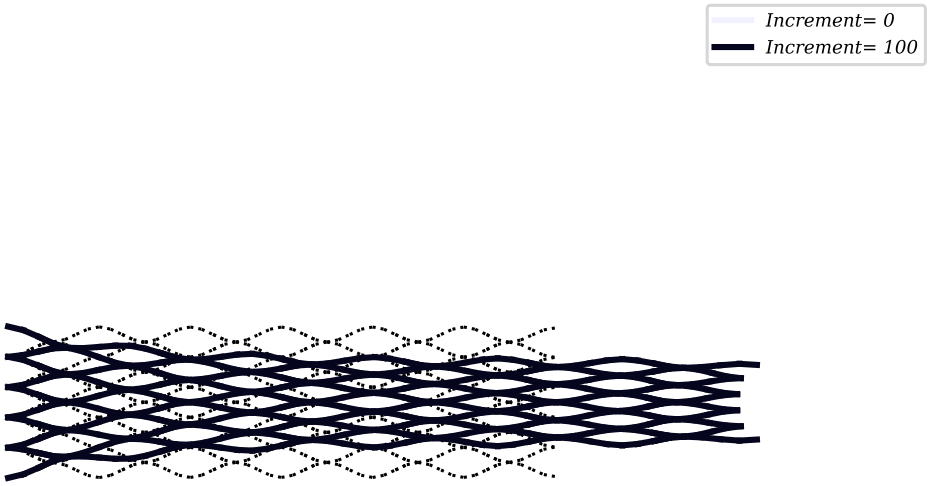

Figure 76: Cellular material sample structure: animation frames

6.9.2 Compression loading: main path

Compression forces are applied at end nodes. The structure is shown in Fig.77. An additional constraint is added on the vertical displacement of end nodes in order to prevent local instabilities in the areas of force introduction. A more correct approach in order to apply the force on one side of the sample would involve a definition of a tie constraint (rigid body constraint) on all the end nodes. Resulting charts can be appreciated in Fig.78. The animation frames for different time increments are illustrated in Fig.79. The structure has an hardening behaviour. A significant reduction of the tangent stiffness matrix determinant occur at a certain spot. The reduction occurs due to a bifurcation of the load path. As shown by the main load path determinant, the buckling mode is expected for a value of about $\lambda = 22$ of the load multiplier.

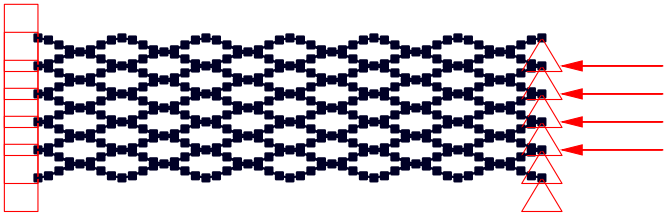

Figure 77: Cellular material sample structure: geometry, discretization, restraints and loading conditions

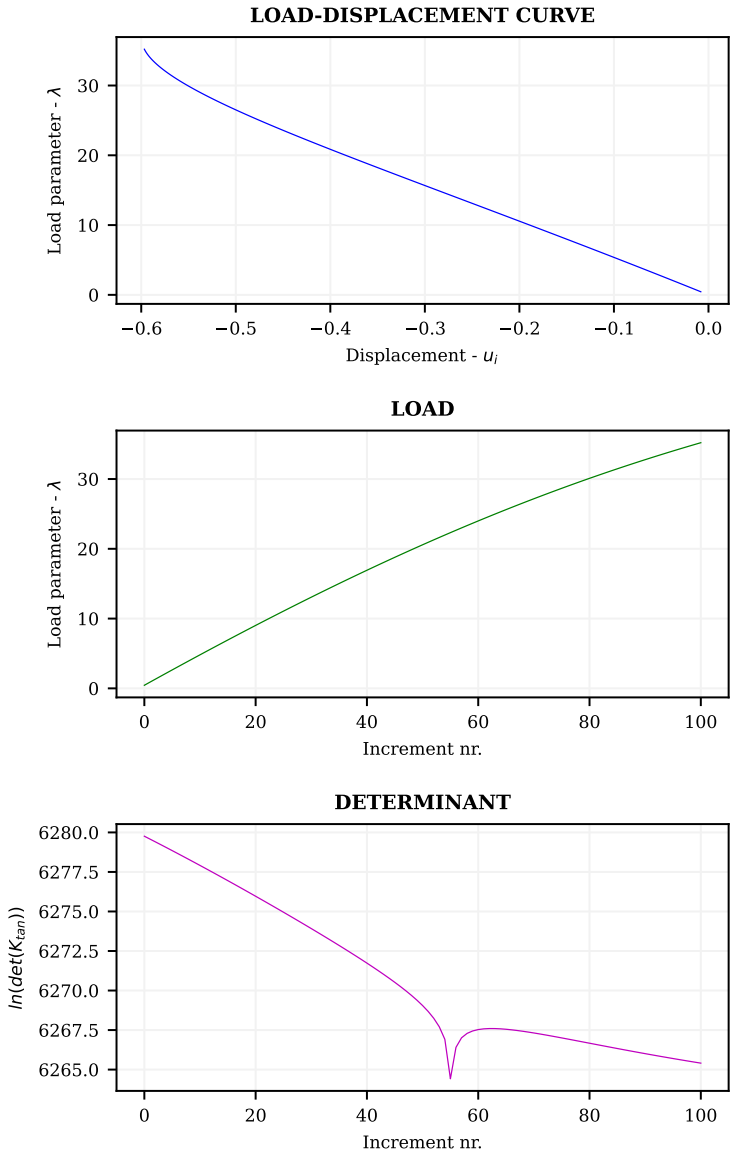

Figure 78: Cellular material sample structure: resulting charts

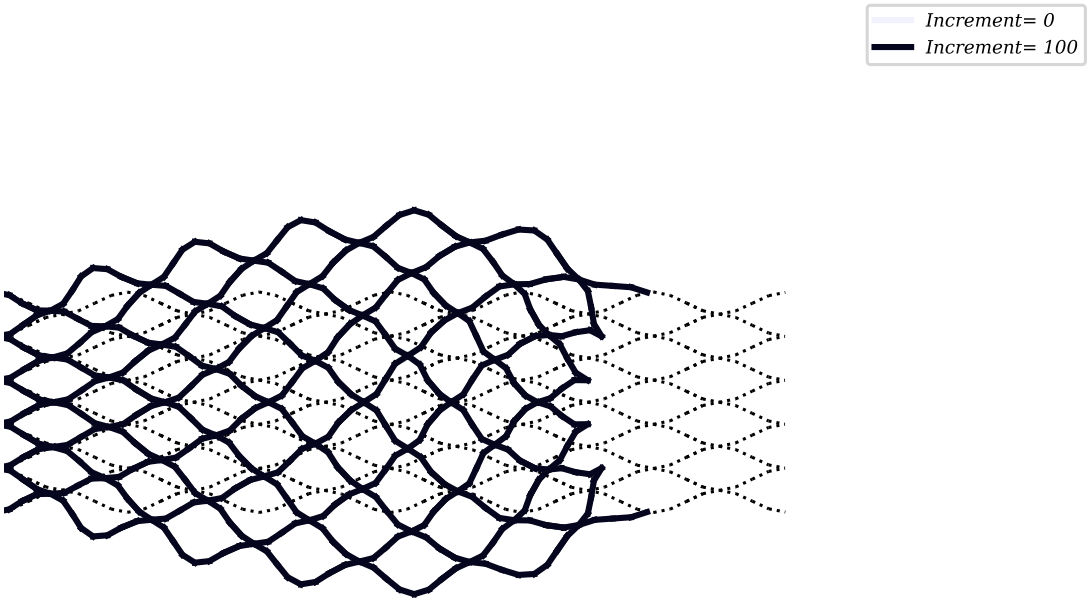

Figure 79: Cellular material sample structure: animation frames

6.9.3 Compression loading: buckling mode

A small non-simmetry is introduced here in the forces values to trig the buckling mode. As shown by the main load path determinant, the second buckling mode is expected for a value of about $\lambda = 22$ of the load multiplier. Compression forces are applied at end nodes. The structure is shown in Fig.80. Resulting charts can be appreciated in Fig.81. The animation frames for different time increments are illustrated in Fig.82. The structure has an hardening behaviour. A significant reduction of the tangent stiffness matrix determinant occur at a certain spot.

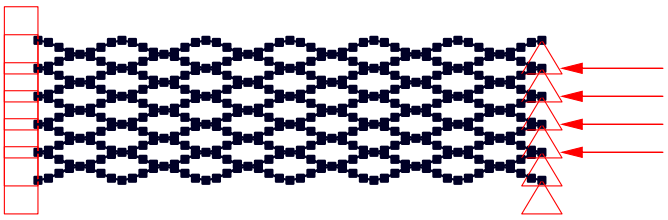

Figure 80: Cellular material sample structure: geometry, discretization, restraints and loading conditions

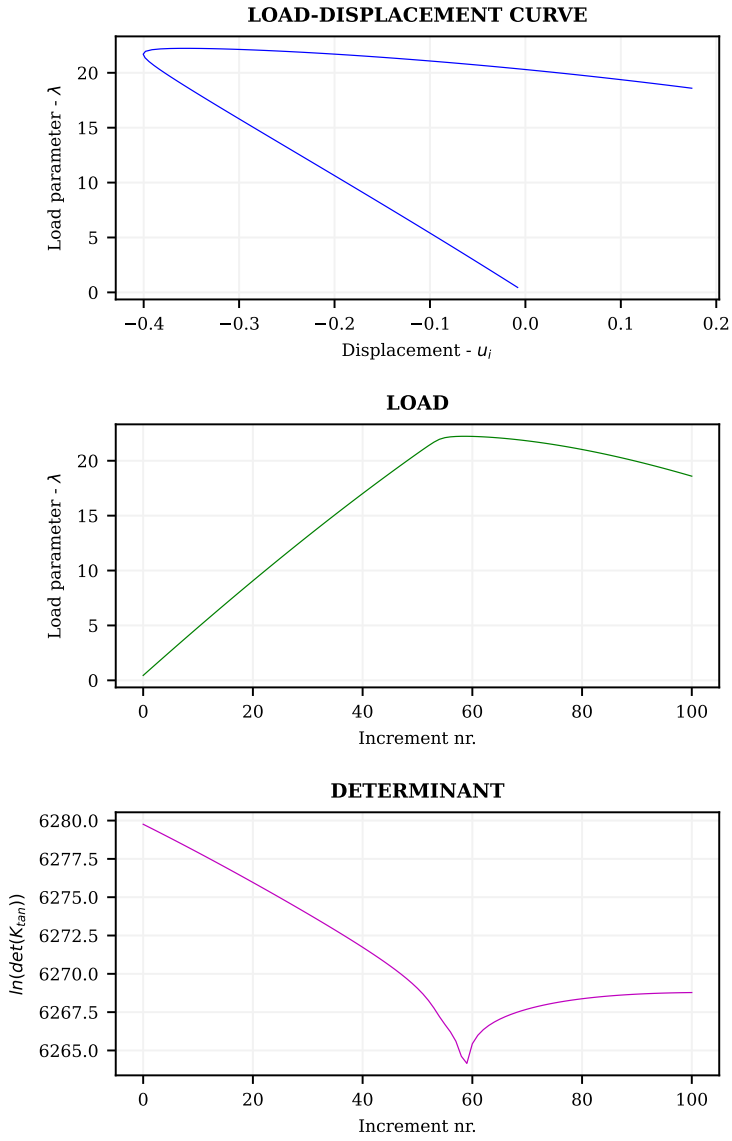

Figure 81: Cellular material sample structure: resulting charts

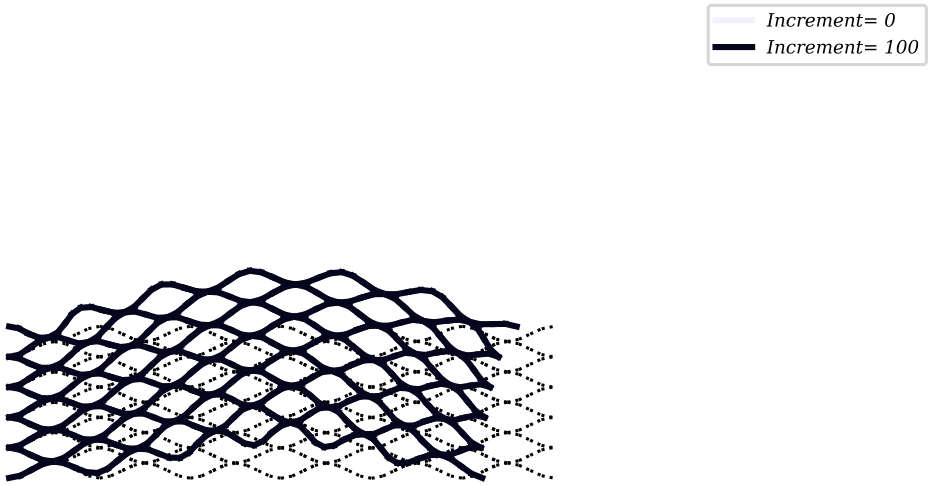

Figure 82: Cellular material sample structure: animation frames

7 Conclusions

A structural analysis software for planar structure analysis in large deflection has been successfully developed in this work. Results have been proven to be in agreement with other literature works. Multiple case studies of engineering interest have been analysed showing both the deformation evolution and the load-displacement path.

Further future implementation and improvement of the code might be:

- further control methods developement (spherical, hyperspherical, hyperelliptical, local hyperelliptical controls);
- other elements formulations;
- implementing buckling analysis (eigen problem resolution);
- program enhancement of tools (example: importing structure geometry and data from excel or an improving in exporting the outputs);
- ...

References

[1] Eurocode 3: Design of steel structures - part 1-5: General rules - plated structural elements.

[2] *Geometrically Non-linear Analysis*, chapter 3, pages 63–111. John Wiley Sons, Ltd, 2012.

[3] Piccolroaz A. Slides of computational mechanics ii course - university of trento - italy.

[4] Devi Chandra, Eka Satria, and Nukman Yusoff. Buckling analysis of curve stiffened fuselage panel of very light jet aircraft. *IOP Conference Series: Materials Science and Engineering*, 1062(1):012049, feb 2021.

[5] Carlos Felippa. The tl plane beam element formulation. *Handouts*.

[6] DalCorso Francesco. Mathematica notebooks of instability of structure course.

[7] Chennakesava Kadapa. A simple extrapolated predictor for overcoming the starting and tracking issues in the arc-length method for nonlinear structural mechanics. 5 2020.

[8] Kjell Mattiasson. Numerical results from large deflection beam and frame problems analysed by means of elliptic integrals. *International Journal for Numerical Methods in Engineering*, 17(1):145–153, 1981.

[9] P. Nanakorn and L. N. Vu. A 2d field-consistent beam element for large displacement analysis using the total lagrangian formulation. *Finite Elements in Analysis and Design*, 42:1240–1247, 10 2006.

[10] Francesco Nappi, Laura Mazzocchi, Irina Timofeeva, Laurent Macron, Simone Morganti, Sanjeet Singh Avtaar Singh, Avtaar Singh, David Attias, Antonio Congedo, and Ferdinando Auricchio. A finite element analysis study from 3d ct to predict transcatheter heart valve thrombosis. *Diagnostics*, 10, 03 2020.

[11] Dinh Kien Nguyen and Thi Thom Tran. A corotational formulation for large displacement analysis of functionally graded sandwich beam and frame structures. *Mathematical Problems in Engineering*, 2016, 2016.

[12] K.H. Schweizerhof and P. Wriggers. Consistent linearization for path following methods in nonlinear fe analysis. *Computer Methods in Applied Mechanics and Engineering*, 59(3):261–279, 1986.

[13] P. Wriggers. *Nonlinear Finite Element Methods*. Springer Berlin Heidelberg, 2008.

[14] Yeong-Bin Yang and Ming-Shan Shieh. Solution method for nonlinear problems with multiple critical points. *AIAA journal*, 28(12):2110–2116, 1990.

A Appendix: Code lines and program structure

As complementary content, the Python code constituting the program is here shown and commented. An overview of the code structure is given and the raw code lines are introduced and briefly described. In the code the analytical defined quantities are recognisable. The code tries to be as much self-explanatory as possible exploiting the Python coding style. The important functions for computational mechanics purposes are illustrated and explained. Auxiliary functions (such as result plotting, animation plotting, reset, user input, interface building functions) are omitted.

A.1 Python language brief introduction

Python is an interpreted, object-oriented, high level programming language. A python code can be written exploiting integrated development environments (IDEs) such as PyDev, PyCharm, Visual Studio Code (...). For this work PyCharm was exploited but other environments are valid choices as well. Python presents valid alternatives to other computational, analytical and engineering tools such as Matlab or Mathematica. An important feature of python language is the availability of already implemented libraries that can be found. The libraries constitute extensions to the raw python languafige capabilities. Important libraries that have been exploited in this work are the numpy library that enables to perform linear algebra calculations, the matplotlib library that consent to plot graphics such as charts, the kivy library that allowst to structure the user interface.

The python code is structured in Class objects: a Class is an object constructor (a "blueprint" for constructing structured objects). A Class contains properties and methods. The properties are variables that are associated with the defined object. Methods are basically functions which are strictly related to the class. Other important tools in a python code are functions.

A.2 Joint class

A joint object is constructed according to the Joint class (Fig.83). The joint class contains general attributes like the number of joints defined, the dof number of the Joint class, the dofs labels. a Joint object has a name, an x1 coordinate a x2 coordinate and a "Restraint" and a "JointLoad" object. These last two are constructed according to different classes described in Fig.85. The Joint class has different methods. Most important methods are shown in Fig.84. The "nodal load vector" method returns a 3-vector containing the nodal force components. The connectivity matrix is $3 \times N$ matrix that allows to shifting to the structural dofs, that has N total dofs, to the joint object dofs, that has 3 dofs, with a simple matrix multiplication operation. The vice-versa operation is of course possible by transposing this matrix.

File - C:\Users\Francesco\PycharmProjects\Tachyon\old\nonlinear_frame\joint\joint.py

```
 3 class Joint: #class
 4     num_of_joints=0
 5     dofs_number = 3
 6     dofs_labels = ['u', 'v', 'theta']
 7 
 8     def __init__(self, name, x1, x2, restraints, jointLoads):
 9         self.name=name
10         self.x1=x1
11         self.x2=x2
12         self.restraints=restraints
13         self.jointLoads = jointLoads
14         self.dofs_list=[]
15         Joint.num_of_joints+=1
```

Figure 83: Joint class definition and object initialization procedure

File - C:\Users\Francesco\PycharmProjects\Tachyon\old\nonlinear_frame\joint\joint.py

```
21     def nodal_load_vector(self):
22         ret=np.transpose(np.array([[self.jointLoads.P1,self.jointLoads.P2,self.jointLoads.P3]]))
23         return ret
24 
25     def connectivity_matrix(self, dofs_number):
26         a = np.zeros((Joint.dofs_number, dofs_number))
27         for i in range(Joint.dofs_number):
28             a[i, self.restraints.globalDOF[i]-1] = 1
29 
30         return np.array(a)
```

Figure 84: Joint class: nodal load vector and connectivity matrix methods

File - C:\Users\Francesco\PycharmProjects\Tachyon\old\nonlinear_frame\joint\joint.py

```
46 class JointLoad:
47     def __init__(self, P1=None, P2=None, P3=None):
48         if P1==None:
49             P1=0
50         if P2==None:
51             P2=0
52         if P2==None:
53             P3=0
54         self.P1=P1
55         self.P2=P2
56         self.P3 = P3
57 
58 class Restraint:
59     def __init__(self, flags=None, globalDOF=None):
60         if globalDOF is None:
61             globalDOF = [-1, -1, -1]
62         if flags is None:
63             flags = [0, 0, 0]
64 
65         self.flags = flags
66         self.globalDOF=globalDOF #(-1= unassigned; 0=unactive; >0=active and assigned to corresp.
   number)
67 
68     def reset(self):
69         self.globalDOF= [-1, -1, -1]
70 
71     def flag_u1(self):
72         return self.flags[0]
73 
74     def flag_u2(self):
75         return self.flags[1]
76 
77     def flag_u3(self):
78         return self.flags[2]
```

Figure 85: Other classes: the "JointLoad" class and the "Restraint" class

A.3 TrussElement class

The "TrussElement" class (Fig.86) provides the construction of the Truss element type object. The definition needs a name, two "Joint" objects, a cross section area in the reference configuration and the Young's modulus.

File - C:\Users\Francesco\PycharmProjects\Tachyon\old\nonlinear_truss\finite_elements\truss_element.py

```
 3 class TrussElement:
 4     num_of_frameElements = 0
 5
 6     def __init__(self, name, joint1, joint2, E, A, t0):
 7         self.name=name
 8         self.joint1=joint1
 9         self.joint2=joint2
10         self.E = E
11         self.A = A
12         self.t0 = t0
13         self.restraints_joint1=Restraint(flags=[1, 1])
14         self.restraints_joint2=Restraint(flags=[1, 1])
15         self.animation = None
16
17         self.displacements=np.array([0,0,0,0])
18
19         TrussElement.num_of_frameElements+=1
```

Figure 86: "TrussElement" class definition and initialization procedure

In Fig.87 the connectivity matrix and the rotation matrix are provided.

File - C:\Users\Francesco\PycharmProjects\Tachyon\old\nonlinear_truss\finite_elements\truss_element.py

```
49     def connectivity_matrix(self, dofs_number):
50         a=np.zeros((4,dofs_number))
51         for i in range(2):
52             for j in range(dofs_number):
53                 if self.restraints_joint1.globalDOF[i]==j+1:
54                    a[i,j]=1
55                 if self.restraints_joint2.globalDOF[i]==j+1:
56                    a[i+2,j]=1
57         return np.array(a)
58
59     def rotation_matrix(self):
60         phi=self.angle()
61         cos = np.cos(phi)
62         sin = np.sin(phi)
63         rotationMatrix=np.array([[cos, sin, 0, 0], [-sin, cos, 0, 0], [0, 0, cos, sin],
64                                  [0, 0, -sin, cos]])
65         return np.array(rotationMatrix)
```

Figure 87: Truss element class: connectivity matrix and rotation matrix (reference configuration) methods

The green lagrange strain measure e, the conjugate PK stress t, the matrix $\boldsymbol{M}$, the vector $\boldsymbol{B}_0$ and the vector $\boldsymbol{B}$ according to what defined in the analytical description are shown in Fig.88.

File - C:\Users\Francesco\PycharmProjects\Tachyon\old\nonlinear_truss\finite_elements\truss_element.py

```
67      def strainGL_e(self):
68          B0=self.vector_B0()
69          u=self.displacements
70          M=self.matrix_M()
71
72          e=np.dot(B0,u)+0.5*np.dot(np.transpose(u),np.dot(M, u))
73          return e
74
75      def stress_t(self):
76          return self.t0+self.E*self.strainGL_e()
77
78      def matrix_M(self):
79          length=self.length()
80          L0_quad=length**2
81          return 1/L0_quad*np.array([[1,0,-1,0],[0,1,0,-1],[-1,0,1,0],[0,-1,0,1]])
82
83      def vector_B0(self):
84          L0 = self.length()
85          c0x=np.cos(self.angle())
86          c0y = np.sin(self.angle())
87          B0=1/L0*np.array([[-c0x, -c0y, c0x, c0y]])
88          return B0
89
90      def vector_B(self):
91          u=self.displacements
92          B=np.add(self.vector_B0(),np.dot(np.transpose(u), self.matrix_M()))
```

Figure 88: Truss eleemnt class: methods returning the green lagrange strain measure, the Piola-Kirchhoff stress, and the analytically defined vectors and matrices $\boldsymbol{M}$, $\boldsymbol{B}$, $\boldsymbol{B}_0$

The element tangent stiffness matrix and the internal forces vector are finally computed as in Fig.89.

File - C:\Users\Francesco\PycharmProjects\Tachyon\old\nonlinear_truss\finite_elements\truss_element.py

```
95      def nodal_loads_p_internal_forces(self):
96          L0=self.length()
97          A0=self.A
98          t=self.stress_t()
99          B=self.vector_B()
100         p=np.transpose(L0*A0*t*B)
101         return p
102
103     def material_stiffness_matrix_KM(self):
104         E=self.E
105         A0=self.A
106         L0=self.length()
107         B=self.vector_B()
108         return E*A0*L0*np.outer(B,B)
109
110     def axial_force_f(self):
111         return self.A*self.stress_t()
112
113     def geometric_stiffness_matrix_KG(self):
114         F=self.axial_force_f()
115         L0=self.length()
116         M=self.matrix_M()
117         return F*L0*M
118
119     def nl_stiffness_matrix_TL(self):
120         kM=self.material_stiffness_matrix_KM()
121         kG= self.geometric_stiffness_matrix_KG()
122         return np.add(kG,kM)
```

Figure 89: Truss element class: internal forces vector and tangent stiffness matrix methods

A.4 TimoshenkoBeam class

The "TimoshenkoBeam" class (Fig.90) allows to define Timoshenko Beam type elements. The "TimoshenkoBeam" is a 6-dofs finite element. The object is defined providing a name, two "Joint" type

objects, two restraint type objects, the vearious material modulus, sectional area and inertia second moment of the section.

File - C:\Users\Francesco\PycharmProjects\Tachyon\old\nonlinear_frame\finite_elements\timoshenko_beam.py

```
 4 class TimoshenkoBeam():
 5     dofs_number=6
 6     dofs_labels=['u1','v1','theta1', 'u2', 'v2', 'theta2']
 7
 8     def __init__(self,name, joint1, joint2, restraints_joint1, restraints_joint2, E, G, A0, I0, N0, V0
   , M0):
 9         self.name=name
10         self.joint1=joint1
11         self.joint2=joint2
12         self.restraints_joint1=restraints_joint1
13         self.restraints_joint2 = restraints_joint2
14         self.E=E
15         self.G=G
16         self.A0=A0
17         self.I0=I0
18         self.N0 = N0
19         self.V0 = V0
20         self.M0 = M0
21         self.displacements=None
22
23         self.EA0=self.E*self.A0
24         self.GA0=self.G*self.A0
25         self.EI0=self.E*self.I0
26
27         self.animation = None
```

Figure 90: "TimoshenkoBeam" class definition and initialization procedure

First important methods of the "TimoshenkoBeam" class are the rotation matrix and the connectivity matrix (Fig.91).

File - C:\Users\Francesco\PycharmProjects\Tachyon\old\nonlinear_frame\finite_elements\timoshenko_beam.py

```
32     def connectivity_matrix(self, global_dofs_number):
33         a=np.zeros((TimoshenkoBeam.dofs_number,global_dofs_number))
34         for i in range(3):
35             for j in range(global_dofs_number):
36                 if self.restraints_joint1.globalDOF[i]==j+1:
37                     a[i,j]=1
38                 if self.restraints_joint2.globalDOF[i]==j+1:
39                     a[i+3,j]=1
40         return np.array(a)
41
42     def rotation_matrix(self):
43         phi = self.undeformed_phi()
44         cos = np.cos(phi)
45         sin = np.sin(phi)
46         rotationMatrix = np.array(
47             [[cos, sin, 0, 0, 0, 0], [-sin, cos, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0], [0, 0, 0, cos, sin
   , 0],
48              [0, 0, 0, -sin, cos, 0], [0, 0, 0, 0, 0, 1]])
49         return np.array(rotationMatrix)
```

Figure 91: Timoshenko beam element connectivity and rotation matrices methods

The current and reference configuration length and angles can be derived with the functions of Fig.92.

File - C:\Users\Francesco\PycharmProjects\Tachyon\old\nonlinear_frame\finite_elements\timoshenko_beam.py

```
56      def undeformed_length(self):
57          return np.sqrt((self.joint2.x1-self.joint1.x1)**2+(self.joint2.x2-self.joint1.x2)**2)
58
59      def current_length(self):
60          x21=self.joint2.x1-self.joint1.x1+self.displacements[3,0]-self.displacements[0,0]
61          y21 = self.joint2.x2 - self.joint1.x2 + self.displacements[4,0] - self.displacements[1,0]
62          return np.sqrt((x21)**2+(y21)**2)
63
64      def undeformed_phi(self):
65          return np.arctan2((self.joint2.x2-self.joint1.x2),(self.joint2.x1-self.joint1.x1))
66
67      def current_phi(self):
68          x21 = self.joint2.x1 - self.joint1.x1 + self.displacements[3,0] - self.displacements[0,0]
69          y21 = self.joint2.x2 - self.joint1.x2 + self.displacements[4,0] - self.displacements[1,0]
70          return np.arctan2((y21), (x21))
71
72      def psi(self):
73          return self.current_phi()-self.undeformed_phi()
```

Figure 92: Timoshenko beam element current and reference configuration useful methods

The useful defined vectors also defined in the analytical description of the formulation are defined according to the methods of Fig.93.

File - C:\Users\Francesco\PycharmProjects\Tachyon\old\nonlinear_frame\finite_elements\timoshenko_beam.py

```
75      def Bm(self):
76          L_0=self.undeformed_length()
77          L=self.current_length()
78          theta_m = (self.displacements[2,0] + self.displacements[5,0]) / 2
79          phi = self.undeformed_phi()
80          omega_m = theta_m + phi
81          c_m = np.cos(omega_m)
82          s_m = np.sin(omega_m)
83          psi=self.psi()
84          e_m = L / L_0 * np.cos(theta_m - psi)-1
85          gamma_m = L / L_0 * np.sin(psi - theta_m)
86
87          Bm_matrix=np.transpose(1/L_0*np.array([[-c_m, -s_m, 0.5*L_0*gamma_m, c_m, s_m, 0.5*L_0*
    gamma_m],
88                                  [s_m, -c_m, -0.5*L_0*(1+e_m), -s_m, c_m, -0.5*L_0*(1+e_m)],
89                                  [0,0,-1,0,0,1]]))
90          return Bm_matrix
91
92      def h_vector(self):
93          L_0 = self.undeformed_length()
94          L = self.current_length()
95          theta_m = (self.displacements[2,0] + self.displacements[5,0]) / 2
96          psi = self.psi()
97          e_m = L / L_0 * np.cos(theta_m - psi)-1
98          gamma_m = L / L_0 * np.sin(psi - theta_m)
99          curvature=1/L_0*(self.displacements[5,0]-self.displacements[2,0])
100
101         return np.transpose(np.array([e_m, gamma_m, curvature]))
102
103     def z_vector(self):
104         N0=self.N0
105         V0=self.V0
106         M0=self.M0
107
108         EA0=self.EA0
109         GA0=self.GA0
110         EI0=self.EI0
111
112         h=self.h_vector()
113
114         N=N0+EA0*h[0]
115         V=V0+GA0*h[1]
116         M=M0+EI0*h[2]
117
118         return np.transpose(np.array([[N, V, M]]))
```

Figure 93: Timoshenko beam element: methods returning the analytically defined vectors

The "internal nodal forces", "material stiffness matrix", "geometric stiffness matrix", total tangent "non linear stiffness matrix", are derived according to the methods shown in Fig.94 and Fig.95.

File - C:\Users\Francesco\PycharmProjects\Tachyon\old\nonlinear_frame\finite_elements\timoshenko_beam.py

```
120     def internal_nodal_forces(self):
121         z=self.z_vector()
122         Bm_matrix=self.Bm()
123         L_0=self.undeformed_length()
124 
125         return L_0*np.dot(Bm_matrix, z)
126 
127     def material_stiffness_matrix(self):
128         EA0=self.EA0
129         EI0=self.EI0
130         GA0=self.GA0
131         L_0 = self.undeformed_length()
132         L = self.current_length()
133         theta_m = (self.displacements[2,0] + self.displacements[5,0]) / 2
134         phi = self.undeformed_phi()
135         omega_m = theta_m + phi
136         c_m = np.cos(omega_m)
137         s_m = np.sin(omega_m)
138         psi = self.psi()
139         e_m = L / L_0 * np.cos(theta_m - psi)-1
140         gamma_m = L / L_0 * np.sin(psi - theta_m)
141         a_1=1+e_m
142 
143         k_ma=EA0/L_0*np.matrix([[c_m**2, c_m*s_m, -c_m*gamma_m*L_0/2, -c_m**2, -c_m*s_m, -c_m*gamma_m
    *L_0/2],
144                         [c_m*s_m, s_m**2, -gamma_m*L_0*s_m/2, -c_m*s_m, -s_m**2, -gamma_m*L_0*s_m/2],
145                         [-c_m*gamma_m*L_0/2, -gamma_m*L_0*s_m/2, gamma_m**2*L_0**2/4, c_m*gamma_m*L_0/
    2, gamma_m*L_0*s_m/2, gamma_m**2*L_0**2/4],
146                         [-c_m**2, -c_m*s_m, c_m*gamma_m*L_0/2, c_m**2, c_m*s_m, c_m*gamma_m*L_0/2],
147                         [-c_m*s_m, -s_m**2, gamma_m*L_0*s_m/2, c_m*s_m, s_m**2, gamma_m*L_0*s_m/2],
148                         [-c_m*gamma_m*L_0/2, -gamma_m*L_0*s_m/2, gamma_m**2*L_0**2/4, c_m*gamma_m*L_0/
    2, gamma_m*L_0*s_m/2, gamma_m**2*L_0**2/4]])
149 
150         k_mb=EI0/L_0*np.matrix([[0,0,0,0,0,0],
151                                 [0,0,0,0,0,0],
152                                 [0,0,1,0,0,-1],
153                                 [0,0,0,0,0,0],
154                                 [0,0,0,0,0,0],
155                                 [0,0,-1,0,0,1]])
156 
157         k_ms=GA0/L_0*np.matrix([[s_m**2, -c_m*s_m, -a_1*L_0*s_m/2, -s_m**2, c_m*s_m, -a_1*L_0*s_m/2],
158                                 [-c_m*s_m, c_m**2, c_m*a_1*L_0/2, c_m*s_m, -c_m**2, c_m*a_1*L_0/2],
159                                 [-a_1*L_0*s_m/2, c_m*a_1*L_0/2, a_1**2*L_0**2/4, a_1*L_0*s_m/2, -c_m*
    a_1*L_0/2, a_1**2*L_0**2/4],
160                                 [-s_m**2, c_m*s_m, a_1*L_0*s_m/2, s_m**2, -c_m*s_m, a_1*L_0*s_m/2],
161                                 [c_m*s_m, -c_m**2, -c_m*a_1*L_0/2, -c_m*s_m, c_m**2, -c_m*a_1*L_0/2],
162                                 [-a_1*L_0*s_m/2, c_m*a_1*L_0/2, a_1**2*L_0**2/4, a_1*L_0*s_m/2, -c_m*
    a_1*L_0/2, a_1**2*L_0**2/4]])
163 
164         k=k_ma+ k_mb+k_ms
165 
166         return k
167 
168     def geom_stiffness_matrix(self):
169         L_0=self.undeformed_length()
170         L_0h=L_0/2
171         L_0 = self.undeformed_length()
172         L = self.current_length()
173         theta_m = (self.displacements[2,0] + self.displacements[5,0]) / 2
174         phi = self.undeformed_phi()
175         omega_m = theta_m + phi
176         c_m = np.cos(omega_m)
177         s_m = np.sin(omega_m)
178         psi = self.psi()
179         e_m = L / L_0 * np.cos(theta_m - psi) - 1
180         gamma_m = L / L_0 * np.sin(psi - theta_m)
181         z = self.z_vector()
182         N_m=z[0,0]
183         V_m=z[1,0]
184 
185         k_g1=N_m/2*np.matrix([[0,0,s_m,0,0,s_m],
186                               [0,0,-c_m,0,0,-c_m],
187                               [s_m, -c_m, -L_0h*(1+e_m), -s_m, c_m, -L_0h*(1+e_m)],
188                               [0,0,-s_m,0,0,-s_m],
189                               [0,0,c_m,0,0,c_m],
190                               [s_m,-c_m,-L_0h*(1+e_m), -s_m, c_m, -L_0h*(1+e_m)]])
191 
192         k_g2=V_m/2*np.matrix([[0,0,c_m,0,0,c_m],
193                               [0,0,s_m, 0,0, s_m],
194                               [c_m, s_m, -L_0h*gamma_m, -c_m, -s_m, -L_0h*gamma_m],
195                               [0,0,-c_m, 0,0,-c_m],
```

Figure 94: Timoshenko element: internal forces vector, material stiffness matrix, geometric stiffness matrix and tangent stiffness matrix methods

File - C:\Users\Francesco\PycharmProjects\Tachyon\old\nonlinear_frame\finite_elements\timoshenko_beam.py

```
196                             [0,0,-s_m, 0,0,-s_m],
197                             [c_m, s_m, -L_0h*gamma_m, -c_m, -s_m, -L_0h*gamma_m]])
198
199         return k_g1+k_g2
200
201     def nl_stiffness_matrix(self):
202         kM=self.material_stiffness_matrix()
203         kG=self.geom_stiffness_matrix()
204         return kM+kG
```

Figure 95: Timoshenko element: total tangent stiffness matrix method definition

A.5 Structure class

The "Structure" class (Fig.96) is a Python class object that has a list of "TimoshenkoElement" or "TrussElement" objects and a list of "Joint" objects. Moreover the "Structure" class contains a "displacement" vector in that the current configuration displacement components of the structure are collected. Another important variable associated with the Structure Class is the ".dofs" and ".active dofs" variable that contain the list of the structural dofs and the active dofs.

- C:\Users\Francesco\PycharmProjects\Tachyon\old\nonlinear_frame\structure\structure.py

```
class Structure():
    def __init__(self, frameElementArray, jointsArray):
        self.jointsArray=jointsArray
        self.frameElementArray=frameElementArray
        self.displacements=None
        self.restrained_dofs = None
        self.dofs=0
        self.active_dofs=0
        self.initialized=False
        self.lambda_evolution=[]
```

Figure 96: Structure class definition and object initialization procedure

When running a generic analysis (elastic or nonlinear) the first thing is done is an initialization process. Having a list of joints and elements object the "assign dofs" (Fig.97) method initializes the structure by assigning the degree of freedoms.

File - C:\Users\Francesco\PycharmProjects\Tachyon\old\nonlinear_frame\structure\structure.py

```
def assign_dofs(self):
        count_assigned_DOFs = 0
        count_active_dofs = 0
        restrained_dofs = []
        rotational_dofs=[]

        def add_dof_to_joint(k, joint, restr):
            if k == 0:
                new_dof = Dof(name=count_assigned_DOFs, direction=0, restrained=restr)
            elif k == 1:
                new_dof = Dof(name=count_assigned_DOFs, direction=1, restrained=restr)
            elif k == 2:
                new_dof = Dof(name=count_assigned_DOFs, direction=2,restrained=restr)
            joint.dofs_list.append(new_dof)

        for i in self.frameElementArray:
            for j in range(2):
                if j==0:
                    joint=i.joint1
                    elem_restraints=i.restraints_joint1
                elif j==1:
                    joint = i.joint2
                    elem_restraints = i.restraints_joint2

                for k in range(type(joint).dofs_number):  # nodo 1 elemento i-esimo
                    restraintStatus = joint.restraints.flags[k] * elem_restraints.flags[k]
                    if restraintStatus == 0:
                        if joint.restraints.flags[k] == 0 and elem_restraints.flags[k] == 1:
                            if joint.restraints.globalDOF[k] > 0:
                                elem_restraints.globalDOF[k] = joint.restraints.globalDOF[k]
                            if joint.restraints.globalDOF[k] < 0:
                                count_assigned_DOFs += 1
                                add_dof_to_joint(k, joint,False)

                                if k==2:
                                    rotational_dofs.append(1)
                                else:
                                    rotational_dofs.append(0)
                                count_active_dofs += 1
                                restrained_dofs.append(0)
                                joint.restraints.globalDOF[k] = count_assigned_DOFs
                                elem_restraints.globalDOF[k] = count_assigned_DOFs
                        else:
                            count_assigned_DOFs += 1
                            add_dof_to_joint(k, joint, False)

                            if k == 2:
                                rotational_dofs.append(1)
                            else:
                                rotational_dofs.append(0)
                            count_active_dofs += 1
                            restrained_dofs.append(0)
                            elem_restraints.globalDOF[k] = count_assigned_DOFs
                    if restraintStatus == 1:
                        if joint.restraints.globalDOF[k] > 0:
                            elem_restraints.globalDOF[k] = joint.restraints.globalDOF[k]
                        else:
                            count_assigned_DOFs += 1
                            add_dof_to_joint(k, joint, True)
                            if k == 2:
                                rotational_dofs.append(1)
                            else:
                                rotational_dofs.append(0)
                            restrained_dofs.append(1)
                            joint.restraints.globalDOF[k] = count_assigned_DOFs
                            elem_restraints.globalDOF[k] = count_assigned_DOFs

        print('Restrained DOFs: {}'.format(restrained_dofs))
        print('Number of DOFs: {}'.format(count_assigned_DOFs))
        print('Number of active DOFs: {}'.format(count_active_dofs))
        print('Rotational DOFs: {}'.format(rotational_dofs))

        self.active_dofs=count_active_dofs
        self.bounding_box=self.get_bounding_box()
        self.dofs = count_assigned_DOFs
```

Figure 97: Structure class: DOFs assignment method

The "reduction matrix" method (Fig. 98) of the "Structure" class allows to compute a matrix composed of zeros and ones components that consent to get the reduced vector and matrices when

multiplied by the correspondent global vector and matrices.

File - C:\Users\Francesco\PycharmProjects\Tachyon\old\nonlinear_frame\structure\structure.py

```
171 def reduction_matrix(self):
172         r_matrix=np.zeros((self.dofs, self.active_dofs))
173         j=0
174         for i in range(self.dofs):
175             if self.restrained_dofs[i]==0:
176                 r_matrix[i,j]=1
177                 j+=1
178 
179         return np.array(r_matrix)
```

Figure 98: Structure class: reduction matrix method

The external forces vector in global and reduced formats can be computed with the functions of Fig.99.

File - C:\Users\Francesco\PycharmProjects\Tachyon\old\nonlinear_frame\structure\structure.py

```
206     def reduced_external_forces(self):
207         r_matrix = self.reduction_matrix()
208         return np.dot(np.transpose(r_matrix), self.global_external_forces())
209 
210     def global_external_forces(self):
211         p_glob=np.zeros((self.dofs,1))
212         for i in self.jointsArray:
213             p_1=i.nodal_load_vector()
214             a = i.connectivity_matrix(self.dofs)
215             p_glob=np.add(p_glob,np.dot(np.transpose(a),p_1))
216         return p_glob
```

Figure 99: Structure class: reduced and global external forces vectors methods

The internal forces vector in global and reduced formats can be computed with the functions of Fig.100.

File - C:\Users\Francesco\PycharmProjects\Tachyon\old\nonlinear_frame\structure\structure.py

```
194     def reduced_internal_forces(self):
195         r_matrix = self.reduction_matrix()
196         return np.dot(np.transpose(r_matrix), self.global_internal_forces())
197 
198     def global_internal_forces(self):
199         p_glob = np.zeros(shape=(self.dofs, 1))
200         for i in self.frameElementArray:
201             a = i.connectivity_matrix(self.dofs)
202             p_internal = i.internal_nodal_forces()
203             p_glob = np.add(p_glob, np.dot(np.transpose(a), p_internal))
204         return p_glob
```

Figure 100: Structure class: reduced and global internal forces vectors methods

The reduced and global tangent stiffness matrices can be computed according with the code lines of Fig.101.

File - C:\Users\Francesco\PycharmProjects\Tachyon\old\nonlinear_frame\structure\structure.py

```
181     def global_geom_nl_stiffness_matrix(self):
182         k_glob=np.zeros((self.dofs, self.dofs))
183 
184         for i in self.frameElementArray:
185             a = i.connectivity_matrix(self.dofs)
186             Knl = i.nl_stiffness_matrix()
187             k_glob=np.add(k_glob,np.dot(np.transpose(a),np.dot(Knl,a)))
188         return np.array(k_glob)
189 
190     def reduced_geom_nl_stiffness_matrix(self):
191         red=self.reduction_matrix()
192         return np.dot(np.transpose(red), np.dot(self.global_geom_nl_stiffness_matrix(), red))
```

Figure 101: Structure class: reduced and global tangent stiffness matrix methods

The solver method (Fig.102, Fig.103, Fig.104, Fig.105) is a function strictly associated with the Structure class that allows to perform the incremental nonlinear analysis. The number of increments, the maximum iterations number for the correction step and the residual tolerance are set.

The strcuture can be solved by different solver methods. First by exploiying an internal python solver exploited from the "scipy" library that contains various rootfinding methods. In the shown case a Sequential Least Squares Programming (SLSQP) is for example exploited in order to minimize the residual. As second option the "my nr" case provides an incremental predictor-corrector method that can rely on different control strategies such as the load control, displacement control and arclength control method. As last option the "my nr arclength" option provides a Riks arclength method implemented according to [5].

File - C:\Users\Francesco\PycharmProjects\Tachyon\old\nonlinear_frame\structure\structure.py

```
230 def solve_structure_nl_geom_analysis(self, u_max, controlled_dof):
231         solver='my_nr_arclength'
232         control_strategy = 'load'
233         number_of_steps = 500
234         max_num_iterations=100
235         eps_r=0.0001
236 
237         if solver == 'internal_python':
238             self.imposed_u = 0.0
239             self.controlled_dof = controlled_dof
240             u = np.dot(np.transpose(self.reduction_matrix()), self.displacements)
241             multiplier_lambda = 0
242             unknowns = np.insert(np.delete(u, controlled_dof), controlled_dof, multiplier_lambda)
243 
244             for step in range(number_of_steps):
245                 print('>>> Step: {}'.format(step))
246                 self.imposed_u = step / (number_of_steps - 1) * u_max
247                 solution = minimize(self.residual, unknowns, method='SLSQP')
248                 unknowns = solution.x
249                 self.store_animation_frame()
250 
251                 print('Incognite: {}'.format(unknowns))
252 
253                 new_row = np.array([[self.imposed_u, self.displacements[5,0], unknowns[controlled_dof
    ]]])
254                 if step == 0:
255                     load_displacement_curve = new_row
256                 else:
257                     load_displacement_curve = np.append(load_displacement_curve, new_row, axis=0)
258 
259                 # self.print_status()
260                 print('[{}, {}, {}]'.format(self.frameElementArray[-1].displacements[3,0],
261                                 self.frameElementArray[-1].displacements[4,0],
262                                 self.frameElementArray[-1].displacements[5,0]))
263 
264         if solver=='my_nr':
265             self.controlled_dof=controlled_dof
266             val_lambda=0
267             red_displacements=np.dot(np.transpose(self.reduction_matrix()), self.displacements)
268             x=np.append(red_displacements, [[val_lambda]], axis=0)
269             for step in range(number_of_steps+1):
270                 p_ref = self.reduced_external_forces()
271                 if control_strategy == 'displacement':
272                     applied_u=(step/(number_of_steps))**1*u_max
273                 elif control_strategy=='load':
274                     applied_lambda=(step/(number_of_steps))*10
275                 elif control_strategy=='arclength':
276                     del_s=0.56
277                 u = x[:-1]
278                 self.displacements = np.dot(self.reduction_matrix(), u)
279                 self.store_displaced_coordinates()
280                 self.store_frame_elements_displacements()
281 
282                 residual_forces = self.reduced_internal_forces() - x[-1,0] * p_ref
283                 if control_strategy == 'displacement':
284                     residual_constraint_equation = [x[controlled_dof] - applied_u]
285                 elif control_strategy=='load':
286                     residual_constraint_equation=[x[-1] - applied_lambda]
287                 elif control_strategy == 'arclength':
288                     u_n=u
289                     lambda_n=x[-1]
290                     k_tan_0 = self.reduced_geom_nl_stiffness_matrix()
291                     v_n = np.linalg.solve(k_tan_0, p_ref)
292                     f_n = np.sqrt(1 + np.dot(np.transpose(v_n), v_n))[0,0]
293                     del_u = v_n/f_n*del_s
294                     del_lambda=1/f_n*del_s
295                     residual_constraint_equation = 1/f_n*((np.dot(np.transpose(v_n), del_u)+
    del_lambda))-del_s
296                 residual = np.append(residual_forces, residual_constraint_equation, axis=0)
297 
298                 exit_iteration = False
299                 iteration = 0
300                 while exit_iteration is False:
301                     iteration+=1
302                     k_tan = self.reduced_geom_nl_stiffness_matrix()
303                     if control_strategy == 'displacement':
304                         a = np.zeros((1, self.active_dofs))
305                         a[0, controlled_dof] = 1
306                         g = 0
307                     elif control_strategy == 'load':
308                         a = np.zeros((1, self.active_dofs))
```


Figure 102: Structure class: solver method - part 1

File - C:\Users\Francesco\PycharmProjects\Tachyon\old\nonlinear_frame\structure\structure.py

```
309                         g = 1
310                     elif control_strategy == 'arclength':
311                         a=1/f_n*np.transpose(v_n)
312                         g=1/f_n
313                     jacobian=np.block([[k_tan,-p_ref],[a, g]])
314                     x=x-np.linalg.solve(jacobian, residual)
315                     u=x[:-1]
316                     self.displacements=np.dot(self.reduction_matrix(), u)
317                     self.store_displaced_coordinates()
318                     self.store_frame_elements_displacements()
319                     residual_forces=self.reduced_internal_forces()-x[-1,0]*p_ref
320                     if control_strategy == 'displacement':
321                         residual_constraint_equation = [x[controlled_dof] - applied_u]
322                     elif control_strategy=='load':
323                         residual_constraint_equation = [x[-1] - applied_lambda]
324                     elif control_strategy == 'arclength':
325                         del_u = x[:-1]-u_n
326                         del_lambda=(x[-1]-lambda_n)
327                         residual_constraint_equation = 1 / f_n * (np.dot(np.transpose(v_n), del_u) +
    del_lambda) - del_s
328                     residual=np.append(residual_forces, residual_constraint_equation, axis=0)
329                     abs_r=np.linalg.norm(residual)
330                     if abs_r<=eps_r:
331                         print("Convergency reached in {} iterations for step nr. {}".format(iteration
    , step))
332                         exit_iteration=True
333                     elif iteration>=max_num_iterations:
334                         print("Maximum number of iterations ({}) reached for step nr.{}".format(
    max_num_iterations, step))
335                         exit_iteration=True
336 
337                 self.store_animation_frame()
338                 val_lambda=x[-1,0]
339 
340                 det = np.linalg.slogdet(k_tan)[1]
341                 new_row = np.array([[step, -u[controlled_dof], val_lambda, det, -u[controlled_dof-1
    ]]])
342                 if step == 0:
343                     load_displacement_curve = new_row
344                 else:
345                     load_displacement_curve = np.append(load_displacement_curve, new_row, axis=0)
346 
347         if solver=='my_nr_arclength':
348             del_s=0.15
349             current_lambda=0
350             u=np.zeros((self.active_dofs, 1))
351             last_correct_u = u
352             u_changing_ratio_sign=1
353             self.displacements=np.dot(self.reduction_matrix(), u)
354             self.store_displaced_coordinates()
355             self.store_frame_elements_displacements()
356 
357             p_ref=self.reduced_external_forces()
358             for step in range(number_of_steps+1):
359                 #predictor step
360                 k_tan = self.reduced_geom_nl_stiffness_matrix()
361                 del_u=np.linalg.solve(k_tan, p_ref)
362                 k_i=np.dot(np.transpose(p_ref),del_u)/(np.dot(np.transpose(del_u),del_u))
363 
364                 if step==0:
365                     k_0=k_i
366 
367                 sign=np.sign(k_i/k_0)
368                 del_lambda=sign*del_s/(np.sqrt(np.dot(np.transpose(del_u),del_u)))
369                 predicted_del_u=del_lambda*del_u
370                 if step==0:
371                     previous_del_u=predicted_del_u
372                     previous_del_lambda=del_lambda
373 
374 
375                 vec_1=np.array([[predicted_del_u[controlled_dof,0], del_lambda[0,0]]])
376                 vec_2=np.array([[previous_del_u[controlled_dof,0], previous_del_lambda[0,0]]])
377 
378                 cos_angle=1/np.linalg.norm(vec_1)*1/np.linalg.norm(vec_2)*np.dot(vec_1,np.transpose(
    vec_2))
379                 if cos_angle<-0.9:
380                     invert_sign=-1
381                 else:
382                     invert_sign=1
383                 # invert_sign=1
384 
```


Figure 103: Structure class: solver method - part 2

File - C:\Users\Francesco\PycharmProjects\Tachyon\old\nonlinear_frame\structure\structure.py

```
385                 last_correct_u = u
386                 last_correct_lambda=current_lambda
387                 current_lambda = current_lambda + del_lambda*invert_sign
388                 lambda_0 = current_lambda
389                 u=u+predicted_del_u*invert_sign
390                 u_0=u
391 
392 
393                 self.displacements = np.dot(self.reduction_matrix(), u)
394                 self.store_displaced_coordinates()
395                 self.store_frame_elements_displacements()
396 
397                 g = lambda_0-last_correct_lambda
398                 a = np.transpose(np.subtract(u_0,last_correct_u))
399 
400                 residual_forces = self.reduced_internal_forces() - current_lambda * self.
    reduced_external_forces()
401                 residual_constraint_equation = np.dot(np.transpose(u_0-last_correct_u),(u-u_0))+(
    lambda_0-last_correct_lambda)*(current_lambda-lambda_0)
402 
403                 #correction steps
404                 iteration=0
405                 r = np.linalg.norm(residual_forces)
406                 if r <= eps_r:
407                     exit_iteration=True
408                     print("Convergency reached in {} iterations for step nr. {}".format(iteration,
    step))
409                 else:
410                     exit_iteration = False
411                 while exit_iteration is False:
412                     iteration+=1
413 
414                     k_tan = self.reduced_geom_nl_stiffness_matrix()
415                     del_uP_next=np.linalg.solve(k_tan, p_ref)
416                     del_uG_next=-np.linalg.solve(k_tan, residual_forces)
417                     del_lambda=-(residual_constraint_equation+np.dot(a,del_uG_next))/(g+np.dot(a,
    del_uP_next))
418                     del_u=del_lambda*del_uP_next+del_uG_next
419                     current_lambda=current_lambda+del_lambda
420                     u=u+del_u
421                     self.displacements = np.dot(self.reduction_matrix(), u)
422                     self.store_displaced_coordinates()
423                     self.store_frame_elements_displacements()
424 
425                     residual_forces = self.reduced_internal_forces() - current_lambda * self.
    reduced_external_forces()
426                     residual_constraint_equation = np.dot(np.transpose(u_0-last_correct_u),(u-u_0))+(
    lambda_0-last_correct_lambda)*(current_lambda-lambda_0)
427                     # residual=np.append(residual_forces, residual_constraint_equation,axis=0)
428                     r=np.linalg.norm(residual_forces)
429                     if r<=eps_r:
430                         print("Convergency reached in {} iterations for step nr. {}".format(iteration
    , step))
431                         exit_iteration=True
432                     elif iteration>=max_num_iterations:
433                         print("Maximum number of iterations ({}) reached for step nr.{}".format(
    max_num_iterations, step))
434                         exit_iteration=True
435 
436                 previous_del_lambda=current_lambda-last_correct_lambda
437                 previous_del_u=u-last_correct_u
438 
439                 self.store_animation_frame()
440                 self.lambda_evolution.append(current_lambda)
441 
442                 det=np.linalg.slogdet(k_tan)[1]
443                 new_row = np.array([[step, -u[controlled_dof],current_lambda, det, -u[controlled_dof-
    1]]])
444                 if step == 0:
445                     load_displacement_curve = new_row
446                 else:
447                     load_displacement_curve = np.append(load_displacement_curve, new_row, axis=0)
448 
449         cm=1/2.54
450 
451         font = {'family': 'serif',
452                 'weight': 'normal',
453                 'size': 7}
454 
455         font_title = {'family': 'serif',
456                       'weight': 'bold',
```


Figure 104: Structure class: solver method - part 3

File - C:\Users\Francesco\PycharmProjects\Tachyon\old\nonlinear_frame\structure\structure.py

```
457                         'size': 8}
458
459         matplotlib.rc('font', **font)
460
461         fig, [ax_load_displacement,ax_load_steps,ax_determinant] = plt.subplots(3, 1, figsize=(12*cm
    , 18*cm))
462         fig.tight_layout(pad=5)
463         ax_load_displacement.plot(load_displacement_curve[:, 1], load_displacement_curve[:, 2], color
    ='b', linewidth=0.5, label='Vertical displacement')
464         # ax_load_displacement.plot(load_displacement_curve[:, 4], load_displacement_curve[:, 2],
    color='r', linewidth=0.5, label='Horizontal displacement')
465         ax_determinant.plot(load_displacement_curve[:,0],load_displacement_curve[:,3], color='m',
    linewidth=0.5)
466         ax_load_steps.plot(load_displacement_curve[:, 0], load_displacement_curve[:, 2], color='g',
    linewidth=0.5)
467         # ax_load_displacement.legend()
468
469         ax_determinant.set_xlabel('Increment nr.')
470         ax_determinant.set_ylabel(r'$ln(det(K_{tan}))$')
471         ax_determinant.set_title('DETERMINANT',**font_title)
472         ax_determinant.grid(True,color=[0.95,0.95,0.95])
473
474         ax_load_steps.set_xlabel('Increment nr.')
475         ax_load_steps.set_ylabel(r'Load parameter - '+r'$\lambda$')
476         ax_load_steps.set_title('LOAD',**font_title)
477         ax_load_steps.grid(True,color=[0.95,0.95,0.95])
478
479         ax_load_displacement.set_xlabel(r'Displacement - '+r'$u_i$')
480         ax_load_displacement.set_ylabel(r'Load parameter - '+r'$\lambda$')
481         ax_load_displacement.set_title('LOAD-DISPLACEMENT CURVE', **font_title)
482         ax_load_displacement.grid(True,color=[0.95,0.95,0.95])
483
484         path_directory = r'C://Users/Francesco/Documents/0.0 - DOCUMENTI/0.2 - STUDIO -Università/1.2
     - LM/2.5 - MECCANICA COMPUTAZIONALE 2 (6CFU)/Progetto/CaseStudies/'
485         fig.savefig(path_directory + str(self.description) + "_charts.pdf", bbox_inches='tight')
486
487         plt.show()
488
489         return self.displacements
```

Figure 105: Structure class: solver method - part 4

A.6 Running analysis and user interface

The user interface is built with the python library "Kivy". The user interface allows the user to insert joints and elements and to perform the analysis. While defining the joints and elements in the background the running program defines a "Structure" object and collects the "TimoshencoBeam" or "TrussElement" objects and the "Joint" objects in the "Structure" object correspondent lists. The function that runs the analysis is shown in Fig.106.

File - C:\Users\Francesco\PycharmProjects\Tachyon\old\nonlinear_frame\interface\kv_interface.py

```
353     def run_nonlinear_analysis(self):
354
355         u_max=float(self.ids.u_max.text)
356         controlled_dof=int(self.ids.controlled_dof.text)
357
358         self.structure.print_structure_informations()
359         self.structure.reset()
360         self.structure.assign_dofs()
361         self.structure.solve_structure_nl_geom_analysis(u_max,controlled_dof)
362         self.update_plot(save_pdf=1)
363         self.structure.plot_animation(mode=1)
364         # self.update_details_box()
```

Figure 106: Analysis running procedure

A capture of the user interface is shown in Fig.107.

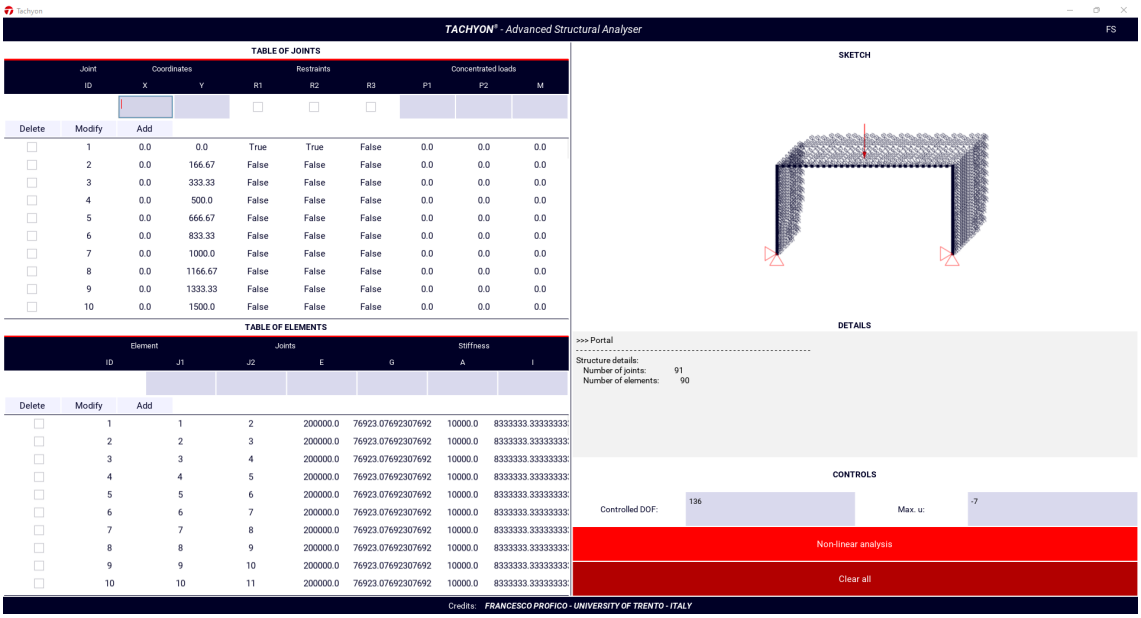

Figure 107: User interface