UNIVERSIDAD EAFIT

NLDYNA - NONLINEAR DYNAMIC ANALYSIS

Julian David Parra Cardona

Advisor:
Prof. Juan David Gomez

A thesis submitted for the degree of Master of Engineering

July 2020

Introduction

The study of structural systems considering inelastic response was, until very recently, only possible through the use of commercial packages with subsequent limitations as research tools. Commercial packages are very rigid since not all of them allow the addition of independent elements and/or material models as often required in research activities. At the same time the used solvers in commercial codes are of the black-box type making its extension to general tasks an impossible goal. The recent development of high level languages (e.g., Python) facilitates the development of highly efficient in-house implementations. This project describes a general in-house finite element assembler and solver aimed at studying the non-linear response of dynamic systems. The code is intended, and has been developed, to be used in the testing of material models, complex kinematic formulations and/or novel structural systems commonly required in research activities. The code has been implemented on top of **SolidsPy**[1] and has been deployed in a GitHub repository[2] which allows portability, strict control version and facilitates it extension in future developments. The repository describes the main futures of the program together with several application problems in terms of Jupyter Notebooks which can be downloaded for execution in a local client or run directly in the repository using a remote virtual server. These Jupyter Notebooks are interactive computational environments which combine live code, equations, visualizations and narrative text. The documents provided as appendices are renderized versions of the Jupyter Notebooks but for proper visualization it is recommended that the reader uses the on-line version of this document.

NLDYNA is a generalized finite element program for the solution of time-dependent non-linear problems. The code is able to handle static and dynamic analysis problems assumed of hyperbolic nature. It is generalized as it can solve user defined problems in different physical contexts through the implementation of user elements and user constitutive responses.

In NLDYNA a dynamic problem is splitted into several time increments and each increment is solved by a Newton-Raphson algorithm. Time stepping is conducted by an implicit Wilson θ-method. The solution of linear static problems takes place in a single increment and a single iteration.

[1] SolidsPy: 2D-Finite Element Analysis with Python. Gómez, Juan and Guarín-Zapata, Nicolás (2018). `https://github.com/AppliedMechanics-EAFIT/SolidsPy`

[2] `https://github.com/jgomezc1/nldyna`

A model is defined in NLDYNA through 5 easy to write input data files containing:

- (i) Basic problem parameters.
- (ii) Nodal point data.
- (iii) Element data.
- (iv) Loads data.
- (v) Material data.

The model can use elements available in the code's own library, specific user defined elements or a combination of both. Similarly, a model can also use NLDYNA's available elements in combination with user defined material models.

The code has the following features:

It is multi-physics oriented: The code is a general dynamic Newton-Raphson solver where the physical context is provided by the user in terms of material and/or element models.

The implementation has been fully parametrized: It does not have an implicit space dimensionality and problems with an arbitrary number of degrees of freedom per node can be solved.

Python based user elements and material models: The implementation of user elements and user constitutive models is highly simplified in comparisson with commercial codes as it is conducted in a high level language.

Easily coupled with independent scripts: Since the code is fully open and written in a modular structure it can be coupled with external independent scripts required in specific analysis and design problems.

Table of Contents

NLDYNA-Nonlinear Dynamic Analysis.

Introduction

The study of structural systems considering inelastic response was, until very recently, only possible through the use of commercial packages with subsequent limitations as a research tool. For instance, commercial packages are very rigid since not all of them allow the addition of independent elements and/or material models as often required in research activities. In top of that restriction the used solvers in commercial codes are of the black-box type making its extension to general tasks an impossible goal. With the recent development of high level lenguages (like Python) it is now possible to develop very efficient in-house implementations. This project describes a general in-house finite element assembler and solver aimed at studying the non-linear response of dynamic systems. The code is intended to be used in the testing of material models and/or complex kinematic formulations commonly required in research activities. The code has the following advantages:

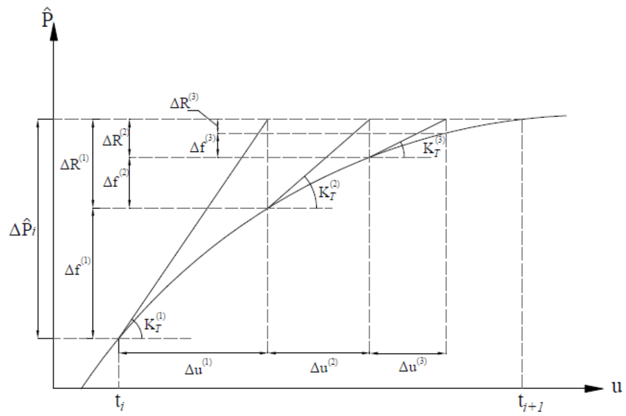
- **It is multiphysics oriented:** The code is just a general dynamic Newton-Raphson solver where the physical context is provided by the user in terms of material and/or element models.
- **The implementation has been fully parametrized:** It does not have an implicit space dimensionality and problems with an arbitrary number of degrees of freedom per node can be solved.
- **Python based user elements and material models:** The implementation of user elements and user constitutive models is highly simplified in comparisson with commercial codes as it is connducted in a high level language like Python.
- **Easily coupled with independent scripts:** Since the code is fully open and written in a modular structure it can be coupled with external independent scripts required in specific analysis and design problems.

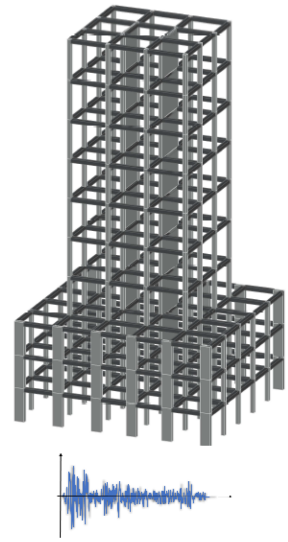
Nonlinear dynamic analysis of generalized finite element problems

NLDYNA is a generalized finite element program for the solution of time-dependent non-linear problems. The code is able to handle static and dynamic analysis problems assumed of hyperbolic nature. It is generalized as it can solve user defined problems in different physical contexts through the implementation of user elements and user constitutive responses.

In **NLDYNA** a dynamic problem is splitted into several time increments and each increment is solved by a Newton-Raphson algorithm. Time stepping is conducted by an implicit Wilson θ-method. The solution of linear static problems takes place in a single increment and a single iteration.

A model is defined in **NLDYNA** through 7 easy to write input data files containing basic problem parameters: nodal point data, element data, loads data and general constraints assignments data. The model can use elements available in the code's library, specific user defined elements or a combination of both. Similarly, a model can also use **NLDYNA's** available elements in combination with user defined material models.





The following sections describe aspects related to the implementation and general use:

- **General problem formulation:** Time stepping scheme (02_Formulation.ipynb) and overall program algorithms.
- **Nonlinear Dynamic Analysis program:** General description (03_NLDYNA.ipynb) of input parameters, output requests and general concepts.
- **User elements subroutines:** Example of implementation of a UEL (04_UEL_subroutine.ipynb) subroutine.
- **User material subroutines:** Example of implementation of a UMAT (05_UMAT_subroutine.ipynb) subroutine.

The following sections describe simple to follow applied examples:

> - **Example 01:** Linear static (06_Example01.ipynb) analysis of a 2D frame.
> - **Example 02:** Linear dynamic (07_Example02.ipynb) analysis of a linear 2D frame.
> - **Example 03:** Linear dynamic (08_Example03.ipynb) analysis of a linear 3D frame.
> - **Example 04:** Nonlinear static (09_Example04.ipynb) analysis of a simple 1D mass-spring system.
> - **Example 05:** Nonlinear dynamic (10_Example05.ipynb) analyisis of a 2D frame.
> - **Example 06:** Design of laterally loaded pile (11_Example06.ipynb).

Time integration

Step by step based methods are general approaches to obtain the system's response to dynamic loading. In these formulations, both the loading and the response history are divided into a sequence of time steps. Each step constitutes an independent analysis where the dynamic problem is solved based in the solution of the previous step. In the finite element method the time dependent system of equations is written like:

$$MA(t) + CV(t) + KU(t) = P(t)$$

where M, C and K are standard mass, damping and stiffness finite element-like matrices, while $A(t)$, $V(t)$ and $U(t)$ are generalized acceleration, velocity and displacement nodal vectors. These are termed generalized as they are not necessarily mechanical quantities. Based on this formulation, it's posible to consider the nonlinear behavior of the system simply by assumming that the assembled properties remain constant during each step and that the change of those properties only happens from one step to the next. Hence, nonlinear analysis becomes a sequence of linear analysis of a changing system (Clough & Penzien, 2003). Consequently, it is convenient to reformulate the system response in terms of the incremental equation of motion, due to the assumption that in nonlinear analysis the properties of the system remains constant only in short increments of time or deformation.

$$M\Delta A + C\Delta V + K\Delta U = \Delta P$$

In this development, the nonlinear behavior is considered in changes in the stiffness contribution. Time integration is conducted through a "θ Wilson" method.

Algorithm 1: The θ Wilson's method

Initial calculations:

1. Solve $\mathbf{M}\ddot{u}_0 = \mathbf{P}_0 - \mathbf{C}\dot{u}_0 \rightarrow \ddot{u}_0$
2. Select θ and Δt
3. $a = \frac{6}{\theta\Delta t}\mathbf{M} + 3\mathbf{C}$
4. $b = 3\mathbf{M} + \frac{\theta\Delta t}{2}\mathbf{C}$

Calculations for each time step, i

for $i \leftarrow 0$ **to** N_{times} **do**

$\delta\widehat{\mathbf{P}}_i = a\dot{u}_i + b\ddot{u}_i + \theta\Delta\mathbf{P}_i$;

Determine tangent stiffness matrix $\mathbf{K}_{T\,i}$;

$\widehat{\mathbf{K}}_{T_i} = \mathbf{K}_{T_i} + \frac{3}{\theta\Delta t}\mathbf{C} + \frac{6}{(\theta\Delta t)^2}\mathbf{M}$;

Solve δu_i from $\widehat{\mathbf{K}}_{T_i}$ and $\delta\widehat{\mathbf{P}}_i$ using **Algorithm 2**;

$\delta\ddot{u}_i = \frac{6}{(\theta\Delta t)^2}\delta u_i - \frac{6}{\theta\Delta t}\dot{u}_i - 3\ddot{u}_i$;

$\Delta\ddot{u}_i = \frac{1}{\theta}\delta\ddot{u}_i$;

$\Delta\dot{u}_i = (\Delta t)\ddot{u}_i + \frac{\Delta t}{2}\Delta\ddot{u}_i$;

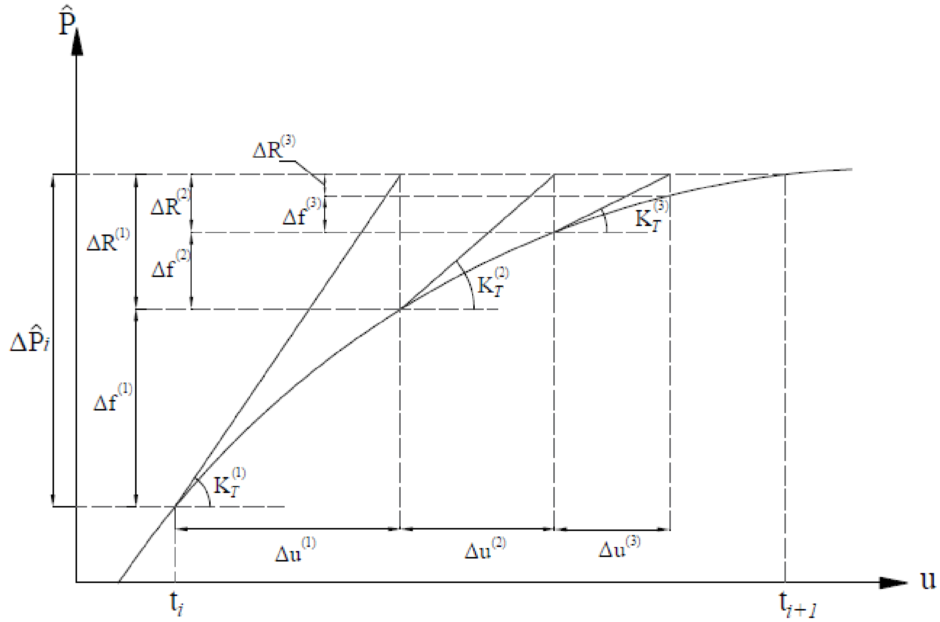
$\Delta u_i = (\Delta t)\dot{u}_i + \frac{(\Delta t)^2}{2}\ddot{u}_i + \frac{(\Delta t)^2}{6}\Delta\ddot{u}_i$;

$u_{i+1} = u_i + \Delta u$; $\dot{u}_{i+1} = \dot{u}_i + \Delta\dot{u}$ and $\ddot{u}_{i+1} = \ddot{u}_i + \Delta\ddot{u}$;

end

Repetition for the next time step (Repeat steps 2.1 to 2.11).

The system's nonlinear response is obtained considering a generalized Newton-Raphson iteration scheme, in which the effective tangent stiffness matrix K_T is calculated at the time i (the beginning of the time step), and that is used through each iteration of changing deformation Δu within that time step. The tangent stiffness matrix K_{T_i} is updated for each iteration until Δu_i it becomes small enough.



Algorithm 2: The Newton-Raphson iterative scheme

Result: $\Delta u_i, \widehat{\mathbf{K}}_{T_i}$
Initialize data:

1. Solve Δu_i from $\widehat{\mathbf{K}}_{T_i} = \widehat{\mathbf{K}}_T^{(0)}$ and $\delta \widehat{\mathbf{P}}_i$
2. Compute the system's internal forces, $\Delta \mathbf{f}^{(0)}$
3. $\Delta R^{(0)} = \delta \widehat{\mathbf{P}}_i - \Delta \mathbf{f}^{(0)}$

if $|\Delta R^{(0)}| > Tol$ **then**
  while $|\Delta R^{(j)}| > Tol$ **do**
    Solve $\Delta u^{(j)}$ from $\widehat{\mathbf{K}}_T^{(j)}$ and $\Delta R^{(j)}$;
    $\Delta u_i = \Delta u_i + \Delta u^{(j)}$;
    Update $\widehat{\mathbf{K}}_T^{(j)}$ from Δu_i;
    Compute $\Delta \mathbf{f}^{(j)}$;
    $\Delta R^{(j)} = \delta \widehat{\mathbf{P}}_i - \Delta \mathbf{f}^{(j)}$;
  end
end

In solving the nonlinear time step, two convergency criteria are considered. The iteration step is deemed to be completed as soon as both the residual forces and the residual deformations (Δf_i and Δu_i) are smaller than the tolerance value stablished by the user.

In the following, it will be exposed a comparison between analytical solution of the linear response of a single degree of freedom system due to an armonic loading excitation and the response that is obtained with "The θ Wilson" step by step procedure:

Input parameters, Output and General concepts.

3.1. Input files

A problem is defined in **NLDYNA** in terms of 7 input data files written in plain text format. Input files associated to different problems can be found in the examples section of this REPO.

3.1.1. General input parameters

The `parameters` file contains global data required for the problem solution.

```
                     INITIAL GLOBAL PARAMETERS FILE
-----------------------------------------------------------------------
Notes:
- 1 = compute, 0 = don´t compute
- If NLSTA = 1 and NLDYNA = 1, only computes NLDYNA.
-----------------------------------------------------------------------
0.02 ......... DeltaT
20.0 ......... Total-time
0.001 ........ Tolerance
100 .......... Max # iterations
0 ...........
0 ........... Compute incremental nonlinear static analysis (NLSTA)
1 ........... Compute time dependent nonlinear dynamic analysis (NLDYNA)
1 ........... Accel - UX
0 ........... Accel - UY
0 ........... Accel - UZ
0 ........... Accel - RX
0 ........... Accel - RY
0 ........... Accel - RZ
0 ...........
1 ........... Storage state variables history
1 ........... Storage internal forces history
0 ........... Data intentionally left as 0
```

The problem parameters are:

- **DeltaT**: Time step defining the input excitation when performing dynamic analysis (NLDYNA) or incremental pseudo-time step when conducting non-linear static analysis (NLSTA). This time step is different from the solution time step which is directly computed by the program.
- **Total-Time**: Total extension of the analysis time window. (This parameter may be different from the total duration of the input excitation).
- **Tolerance:** Precision employed to control the Newton-Raphson iterations (dimensionless).
- **Maximum number of iterations:** Maximum number of Newton-Raphson iterations within a time or deformation increment.
- **NLSTA:** Integer flag defining the analysis type (1 = Perform nonlinear static analysis).
- **NLDYNA:** Integer flag defining the analysis type(1 = Perform nonlinear dynamic analysis).

A value of 1 for any of the following set of parameters defines if inertial effects are active along the associated degree of freedom during a dynamic analysis step. These parameters are active only for `NLDYNA = 1` .

- **Accel-UX:** Activate inertial effects along the "X" direction.
- **Accel-UY:** Activate inertial effects along the "Y" direction.
- **Accel-UZ:** Activate inertial effects along the "Z" direction .
- **Accel-RX:** Activate inertial effects along the rotational mass along the "X" direction.
- **Accel-RY:** Activate inertial effects along the rotational mass along the "Y" direction.
- **Accel-RZ:** Activate inertial effects along the rotational mass along the "Z" direction.

This file also contains two general storage options.

- **Storage state variables history** (1 = Storage)
- **Storage internal forces history** (1 = Storage)

3.1.2. Material profiles

Material properties are defined in terms of material `profiles` . Since the code is not specific to any particular physical context the material properties are always user defined. In this context a `material profile` is a set of parameters defining material behavior and to be used by the different elements in the model. Geometric parameters, like those involved in the definition of structural elements are treated as material properties. The number of parameters may be different in each independent profile.

The sample profile shown in the figure below is defined by a 7-parameters set. This specific material profile is used by structural elements. (see Example 02 for reference)

```
                        MATERIAL/SECTION PARAMETERS INFORMATION FILE
-----------------------------------------------------------------------------------------------
Global material's properties definitions. Nonlinear properties should be specified acoording to
each particular elastoplastic model.

e.g: 7 parameters for 2D frame: A / Iz / E / Density / Alpha-Damp / Betha-Damp / Gravity Value
-----------------------------------------------------------------------------------------------

PARAMETERS INFORMATION
0.25  0.00521  2000000  2.4  0.0  0.0 9.806
0.16  0.00213  2000000  2.4  0.0  0.0 9.806
```

The set of parameters required in the definition of the material profile used in sample problem 02 (Element type = `2`) are

$$Profile = [A, Iz, E, \rho, \alpha_{damp}, \beta_{damp}, g]$$

and where

- **A:** Cross sectional area.
- **Iz:** Moment of inertia along `Z` axis (out of the window).
- **E:** Young's modulus.
- ρ: Material's density
- α_{damp}: Coefficient to calculate the element's damping matrix
- β_{damp}: Coefficient to calculate the element's damping matrix
- **g:** Value of gravity

Review the module `uelutil.py` for a definition of the different parameters that must be defined for the remaining elements currently available in `NLDYNA`

3.1.3. Nodal coordinates and boundary conditions

This files contains nodal coordinates (CX, CY and CZ). At each node the user must also define if the degrees of freedom (TrasX, TrasY, TrasZ, RotX, RotY and RotZ) are active or restrained. By default all the degrees of freedom are assumed active (a value of 0) while a value of -1 defines a restrained degree of freedom.

The user must restrain all those degrees of freedom associated to a non existing space dimension. For instance in a two-dimensional elasticity problem values of -1 must be assigned to those degrees of freedom associated to TrasZ, RotX, RotY and RotZ (see examples 02 for reference).

NODES INFORMATION FILE

ID	CX	CY	CZ	TrasX	TrasY	TrasZ	RotX	RotY	RotZ
0	0	0	0	-1	-1	-1	-1	-1	-1
1	4	0	0	-1	-1	-1	-1	-1	-1
2	4	4	0	-1	-1	-1	-1	-1	-1
3	0	4	0	-1	-1	-1	-1	-1	-1
4	0	0	3	0	0	0	0	0	0
5	4	0	3	0	0	0	0	0	0
6	4	4	3	0	0	0	0	0	0
7	0	4	3	0	0	0	0	0	0
8	0	0	6	0	0	0	0	0	0
9	4	0	6	0	0	0	0	0	0
10	4	4	6	0	0	0	0	0	0
11	0	4	6	0	0	0	0	0	0
12	0	0	9	0	0	0	0	0	0
13	4	0	9	0	0	0	0	0	0
14	4	4	9	0	0	0	0	0	0
15	0	4	9	0	0	0	0	0	0

3.1.4. Elements definition file

This file contains data required to define the different elements within the model. The number of parameters varies depending upon the element type. Each element is defined by a line with the following parameters:

> **3.1.4.1.** Element identifier: Integer defining the element ID.
>
> **3.1.4.2**. Element type: Integer defining the element type.
>
> The following elements are currently available in NLDYNA:
>
> > *- Type = 0. Linear - 1D Spring.*
> > *- Type = 1. Linear - 2D Simple frame.*
> > *- Type = 2. Linear - 2D Full frame.*
> > *- Type = 3. Linear - 2D Truss.*
> > *- Type = 4. Nonlinear - 4 noded plane strain.*
> > *- Type = 5. Nonlinear - 1D Spring.*
> > *- Type = 6. Linear - 2D Shear-Rotational spring.*
> > *- Type = 7. Linear - 1D Rotational spring.*
> > *- Type = 8. Nonlinear - 1D Rotational spring.*
> > *- Type = 9. Nonlinear - 1D soil spring with a bilinear P-Y curve.*
> > *- Type = 10. Linear - 3D Full frame considering shear effects*
>
> **3.1.4.3.** Material/Section profile for the current element.
>
> Material/Section profile identifier assigned to the current element.

```
                              ELEMENT INFORMATION FILE
------------------------------------------------------------------------------------
Element's connectivity information. Element type and connectivity should be specified.

e.g: 5 parameters for frame elements: ID / TYPE / MATERIAL / Ni / Nf
     7 parameters for 4-noded-solids: ID / TYPE / MATERIAL / N1 / N2 / N3 / N4
------------------------------------------------------------------------------------

ELEMENTS
0   10  0   0   4
1   10  0   1   5
2   10  0   2   6
3   10  0   3   7
4   10  0   4   5
5   10  0   5   6
6   10  0   6   7
7   10  0   7   4
8   10  0   4   8
9   10  0   5   9
10  10  0   6   10
```

3.1.5. Nodal (static) loads

Nodal loads are specified in this file. If `NLSTA = 1`, these loads will be treated as pseudo-static loads and are applied incrementally from 0 up to its maximum value; if `NLDYNA = 1`, nodal loads will be treated as permanent loads and will be added to each time interval. A static load is specified by its nodal `ID` and the load magnitude along a given degree of freedom.

```
                  NODAL STATIC LOADS INFORMATION FILE
----------------------------------------------------------------------------
NODE        Fx         Fy          Fz         Mx         My          Mz
5           100.0      235.0       -452.0     0.0        0.0         0.0
3           23.0      -12.5        245.30     0.0       -478.6       0.68
17          123.0      784.0       0.0        0.0        0.0         0.0
0           48.0      -7.5         287.30     0.0        58.6        0.0
```

3.1.6. Ground acceleration:

A value of `NLDYNA = 1` implies that the program is expected to conduct nonlinear dynamic analysis requiring as input excitation a ground motion time history. Only the acceleration history must be specified at this file. The format of the time history file is shown in the figure below.

```
                  SEISMO GROUND ACCELERATION SIGNAL
----------------------------------------------------------------------------------------------
Note:
Signal's time step should be the same that the specified in the initial global parameters file.
----------------------------------------------------------------------------------------------
0.0063
0.00364
0.00099
0.00428
0.00758
0.01087
0.00682
0.00277
-0.00128
0.00368
0.00864
0.0136
0.00727
0.00094
0.0042
0.00221
0.00021
0.00444
0.00867
```

3.1.7. General constraints:

As many automated structural analysis computer programs, NLDYNA admits to use master-slave general constraints options. Two general types of constraints are defined:

> - **Floor diaphragms:** This is activated when Type = 0 to n, and MSTND = 0. For this case, NLDYNA automatically calculates the location of the master node based on the center of mass of the constraint nodes.
> - **Rigid constraints:** This is activated when Type = -1 and MSTND = *Node ID*.

For general rigid constraints, master-slave degrees of freedom should be specified. For example, in rigid bodies, the most general three-dimensional constraint in which all degrees of freedom are related to the master node rigid body displacements, **TrasX, TrasY, TrasZ, RotX, RotY, RotZ** should be equal to 1.

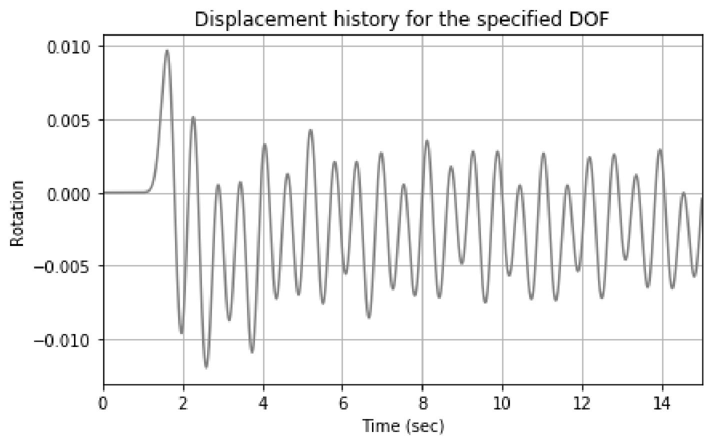
```
NODES GENERAL CONSTRAINTS INFORMATION FILE
------------------------------------------------------------------------------------------------------------
Note:
- Type  0 to n = Floor diaphragm constraints.
- Type -1 = Any other rigid constraint.
- If Type = -1, master node for analysis solution should be specified, e.g. MSTND = 1.
- If MSTND = 1 then it is the master node.
------------------------------------------------------------------------------------------------------------
NODE_ID     Type    MSTND   TrasX   TrasY   TrasZ   RotX    RotY    RotZ
4            0      0       1       1       0       0       0       1
5            0      0       1       1       0       0       0       1
6            0      0       1       1       0       0       0       1
7            0      0       1       1       0       0       0       1
8            1      0       1       1       0       0       0       1
9            1      0       1       1       0       0       0       1
10           1      0       1       1       0       0       0       1
11           1      0       1       1       0       0       0       1
12          -1      0       1       1       1       1       1       1
13          -1      13      1       1       1       1       1       1
14          -1      0       1       1       1       1       1       1
15          -1      0       1       1       1       1       1       1
```

3.2. Output

By default, the program returns the following results:

- The system's natural periods.
- Nodal displacements for each time step.
- Internal forces for each element at each time step.
- State variables at each time step. (This concept will be explained in section 3.3.1).

In the figure below, it is shown the displacement history obtained in the solution of Example 05, see notebook 10 for reference.



3.3. General concepts

To allow for a truly general multiphysics environment the nonlinear finite element code `NLDYNA` uses the concept of `User elements`. In a user element the physics per se is directly handle by the user which provides `NLDYNA` with the Jacobian matrix, residual vector and those variables which history are updated in each Newton iteration. Since the main program is unaware of the specific physics of the problem the variables are termed `State Variables`.

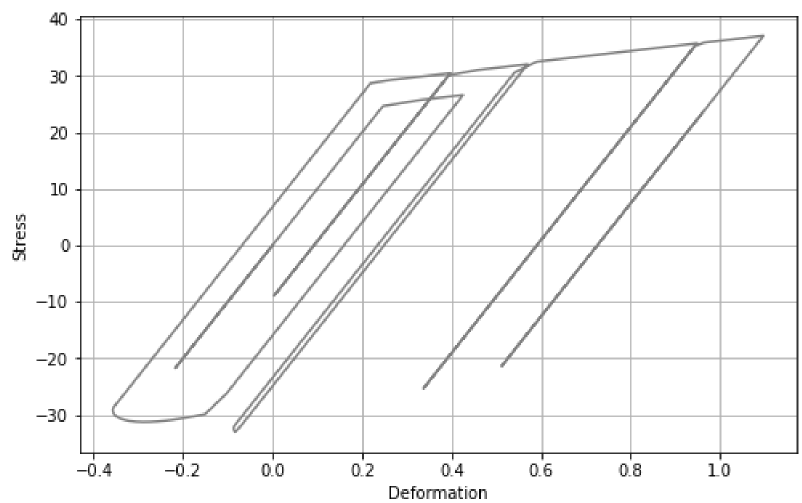
3.3.1. State variables

The state variables is the set of problem specific variables which are of interest in the physical problem being solved and those variables whose time history is required for the computation of the Jacobian matrix and residual vector. During program execution the last converged value of the state variables vector is passed to the element subroutine where it is updated depending upon the increment of the problem primary variable.

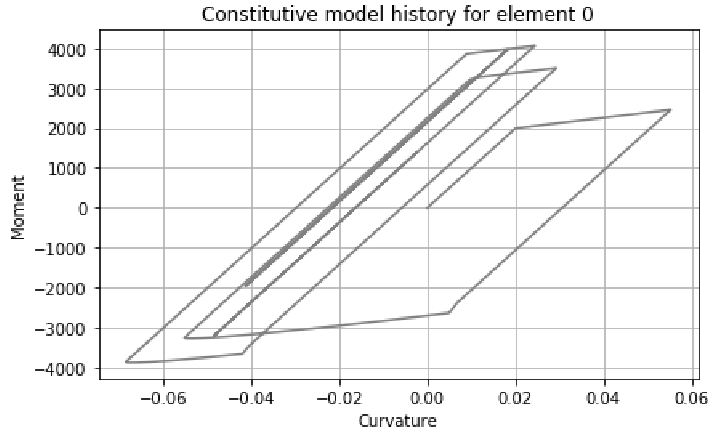
The concept of `State Variables` is briefly explained in terms of the simple example corresponding to a `type = 5` element corresponding to a Nonlinear - 1D Spring along the horizontal direction. The constitutive response of the spring is described by a bilinear isotropic hardening elasto-plastic model (Simo & Hughes, 1991.). In this problem the state variables are:

$$svar = \left[\sigma_i, \delta_i, \delta_{elasi}, \delta_{plas_i}, \alpha_i\right]$$

- σ_i : Stress at increment i
- δ_i : Spring deformation at increment i
- δ_{elasi} : Elastic deformation at increment i
- δ_{plas_i} : Plastic defomration at increment i
- α_i : Isotropic hardening variable at increment i



In the figure below, it is shown the Moment-curvature history for one of the rotational springs used in the solution of Example 05 (see notebook 10 for reference).



3.3.1. User elements

A `user element` subroutine is the set of functions required for the computation of the element contribution to the problem Jacobian matrix and the corresponding residual vector. Since this concept is strongly tied to the Newton-Raphson algorithm it is important that the user reviews **Notebook 04** *UEL (04_UEL_subroutine.ipynb)* **and Notebook 05** *UMAT (05_UMAT_subroutine.ipynb)*.

In [1]:
```
from IPython.core.display import HTML
def css_styling():
    styles = open('./nb_style.css', 'r').read()
    return HTML(styles)
css_styling()
```

Out[1]:

In [ ]:

User element subroutines

Introduction

In very simple terms a finite element code, regardless of its physical context, can be viewed as an assembler of element contributions to a global system of equations complemented with a solver. If the code is intended for the solution of non-linear problems the assembly and solution stages can be combined into a Newton-Raphson algorithm. In either case, the particular physical context of the problem, is enclosed in those subroutines in charge of computing the element contributions. Following the jargon of a popular multiphysics code like `ABAQUS` these specific subroutines in `NLDYNA` are also called `user subroutines` .

There are of two types of user subroutines, namely (i) user material subroutines `UMATS` and (ii) user element subroutines `UELS` . We will discuss both here.

In the particular case of stress analysis a finite element `UEL` subroutine computes the stiffness matrix and internal (or consistent) nodal loads. The computation of the stiffness matrix reduces to the numerical determination of integrals like:

$$K = \int_V B^T \frac{\partial\sigma}{\partial\varepsilon} B \,\mathrm{d}V$$

where the term:

$$C = \frac{\partial\sigma}{\partial\varepsilon}$$

is the Material Jacobian or constitutive tensor.

If the integration to compute K is conducted via Gauss quadrature, the material jacobian must be computed at every integration point. Moreover, in complex constitutive models the material jacobian is a function of a wide variety of physical parameters therefore the values of these parameters must also be available at every integration point. The process of computing C at the integration points is performed by a material user subroutine `UMAT.`

In this notebook we will discuss the implementation of a user element subroutine where it will be assumed that the material model is available. The specific implementation of the `UMAT` subroutine will be discussed elsewhere.

User Element Subroutine (UEL)

A so-called `user element subroutine` must be coded into `NLDYNA` to compute the contribution of the element to the full finite element model. In the most general context a user element is intended to be used in a non-linear dynamic analysis and it is expected that the user is familiar with time stepping procedures and with the general form of Newton-Raphson algorithms.

In the current non-linear dynamics context the user subroutine must define the contribution of the element to the residual vector; the contribution of the element to the jacobian (stiffness) matrix and must also form the mass and damping matrix (when needed), and update all the solution `state variables.`

At each call to the `UEL` subroutine from `NLDYNA` the main program provides the subroutine with values of the element nodal coordinates and all solution-dependent nodal variables (like displacements in elasticity), at all degrees of freedom associated with the element as well as values at the beginning of the current increment of the `solution dependent state variables` associated with the element. Tipically these `solution dependent state variables` are used in the computation of the material response.

Subroutine interface (input and output parameters)

The following set of parameters is passed from the main program to the user subroutine:

- `iele_disp:` (ndarray) Nodal displacements at time t for all the degrees of freedom of the element.
- `coord:` (ndarray) Nodal coordinates for the nodal points defining the element.
- `par:` (ndarray) Material parameters for the material profile associated to the element.
- `svar:` (ndarray) Solution dependent state variables at the beginning of the increment.

The following set of parameters is returned by the `UEL` subroutine to the main program:

- `kG:` (ndarray) Element contribution to the Jacobian (or stiffness) matrix.
- `mG:` (ndarray) Element contribution to the mass matrix (when needed).
- `cG:` (ndarray) Element contribution to the damping matrix (when needed).
- `svar:` (ndarray) Updated vector of solution dependent state variables at the end of the increment.
- `ilf:` (ndarray) Residual vector.

Bi-lineal plane strain quad element with non-linear constitutive response

The following example describes the implementation of a user element subroutine (`UEL`) for a bi-lineal 4-noded plane strain cuadrilateral element with a non-linear constitutive response. The material model (discussed elsewhere) is the plane strain elastoplastic model with combined non-linear isotropic/kinematic hardening formulated in Simo and Hughes (1998). The integration algorithm is a return mapping scheme.

The element data is divided in **(i) solution dependent nodal data** which in this case corresponds to the displacements associated to the degrees of freedom for the element and **(ii) solution dependent state variables**. This last category includes stresses and strains at the Gauss points as well as other variables required in a specific analysis or in the computation of history dependent constitutive responses.

In this particular analysis the `solution dependent state variables` is defined as follows:

- Stress tensor: This is one of the primary variables in the analysis.
- Total strain tensor: This is one of the primary variables in the analysis.
- Elastic strain tensor: Important per se and also required in the computation of the constitutive response.
- Plastic strain tensor: Required in the computation of the constitutive response.
- Back stress tensor: Required in the computation of the constitutive response in order to consider kinematic hardening.
- Equivalent plastic strain: Required in the computation of the constitutive response in order to consider isotropic hardening.
- Equivalent Misses stress: Required in the computation of the constitutive response in order to consider isotropic hardening.

In this 2D problem each tensorial variable contributes with scalar components. Since there are 5 tensorial quantities and defined at 4 Gauss integration points that will make a total of 80 state variables for the element. Also, in the computation of the isotropic hardening behavior we require the equivalent plastic strain and the corresponding equivalent stress (Misses stress). Thus, in total there are 88 state variables per element. Therefore the vector of state variables evolving back and forth bewteen the main program and the user subroutine has a total of 88 components. This vector is termed `svars` in the subroutine.

The subrotine parameters defined next.

```
Parameters
----------
coord    : ndarray
         Nodal coordinates.
props    : ndarray
         Material properties  for the element.
svars    : ndarray
         State variables array at all integration point
s.
du       : ndarray
         Nodal (incremental) displacements vector.
```

The subroutine is conformed by an external loop which conducts Gauss point computations. Let us focus in one of these computations which occur after the Gauss point coordinates and weighting factor have been retrieved.

To start these computations the state variables associated to the current Gauss point must be retrieved from the 88-positions vector `svars` . This is achieved by the `svars` handling subroutine `svarshandl` . The subroutine returns the stress and the total strain tensors at the Gauss point, while the remaining state variables at the Gauss point are stored in a local vector named `statev` . In this problem this vector will store the elastic and plastic strain tensors, the back stress tensor and the equivalent plastic strain and Misses stress. The called to the subroutine is shown below.

```
svars , statev , stress , strann = svarshandl(0 , svars , statev , stress , strann , igp)
```

The second relevant aspect of the subroutine is the computation of the material jacobian or constitutive tensor. This tensor is computed by the `UMAT` subroutine like

```
C , stress , statev = umat_PCLK(stress , strann , dstran , statev , props , ntens)
```

Note that the subroutine receives as input the local state variables at the Gauss point `statev` in addition to the stress and strain tensors and the material properties associated to the material profile. The `UMAT` subroutine returns the constitutive tensor `C` and updated values of stress, strain and state variables.

After accumulation of the stiffness matrix (and others) at the Gauss point is completed the last step is to call once again the `svars` handling subroutine `svarshandl` but now in the reverse direction, i.e., local updated state variables are stored into the global state variables vector. This is shown in the following call:

```
svars , statev , stress , strann = svarshandl(1 , svars , statev , stress , strann , igp)
```

In [1]:
```python
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import sympy as sym
from os import sys
sys.path.append("../source/")
from STRUCTURE import Struct_DYN
from postprocesor import *
from uel_solid import *
```

In [2]:
```
def uel4nquad_PLK(coord, props , svars , du):

    rho = props[6]
    nstatev = 14
    ntens  = 4
    stress = np.zeros([ntens])
    strann = np.zeros([ntens])
    dstran = np.zeros([ntens])
    statev = np.zeros([nstatev])

    kl     = np.zeros([8, 8])
    ml     = np.zeros([8, 8])
    cl     = np.zeros([8, 8])
    rhsl   = np.zeros([8])
    XW, XP = gpoints2x2()

    ngpts = 4
    for i in range(0, ngpts):
        ri  = XP[i, 0]
        si  = XP[i, 1]
        alf = XW[i]
        igp = i*22
        svars , statev , stress , strann = svarshandl(0 , svars , statev , stress , st
nn , igp)

        ddet, B = stdm4NQ(ri, si, coord)
        dstran  = np.dot(B,du)
        strann  = strann + dstran
        C , stress , statev = umat_PCLK(stress , strann , dstran , statev , props , nt
s)
        rhsl = rhsl + np.dot(B.T,stress)*alf*ddet
        kl = kl + np.dot(np.dot(B.T,C), B)*alf*ddet
        N  = sha4(ri , si )
        ml = ml + rho*np.dot(N.T , N)*alf*ddet

        svars , statev , stress , strann = svarshandl(1 , svars , statev , stress , st
nn , igp)

    return kl, ml, cl, svars, rhsl
```

Upon execution the subroutine returns the following parameters

```
kl       : ndarray
           Element stiffness matrix
ml       : ndarray
           Element mass matrix
svars    : ndarray
           Updated state variables array for the element
rhsl     : ndarray
           Consistent internal loads vector
```

To test and execute the subroutine the following block of code emulates the required entries from the main program to the element subroutine.

In [3]:
```
nprops = 7
nsvars = 88
ntens  = 4
props   = np.zeros([nprops])
svars   = np.zeros([nsvars])
du = np.zeros([8])
coord =([0.0 , 0.0], [1.0 , 0.0], [1.0 , 1.0], [0.0 , 1.0])
props[0]= 52.0e3
props[1]= 0.33
props[2]= 60.0
props[3]= 37.0
props[4]= 383.3
props[5]= 2040.0
props[6]= 1000.0
kl, ml, cl, svars, rhsl = uel4nquad_PLK(coord, props , svars , du)
```

Results

In [4]:
```
print(kl)
print(rhsl)
```

```
[[ 32198.14241486  14374.17072092 -22423.70632464   4599.73463069
  -16099.07120743 -14374.17072092   6324.6351172   -4599.73463069]
 [ 14374.17072092  32198.14241486  -4599.73463069   6324.6351172
  -14374.17072092 -16099.07120743   4599.73463069 -22423.70632464]
 [-22423.70632464  -4599.73463069  32198.14241486 -14374.17072092
    6324.6351172    4599.73463069 -16099.07120743  14374.17072092]
 [  4599.73463069   6324.6351172  -14374.17072092  32198.14241486
   -4599.73463069 -22423.70632464  14374.17072092 -16099.07120743]
 [-16099.07120743 -14374.17072092   6324.6351172   -4599.73463069
   32198.14241486  14374.17072092 -22423.70632464   4599.73463069]
 [-14374.17072092 -16099.07120743   4599.73463069 -22423.70632464
   14374.17072092  32198.14241486  -4599.73463069   6324.6351172 ]
 [  6324.6351172    4599.73463069 -16099.07120743  14374.17072092
  -22423.70632464  -4599.73463069  32198.14241486 -14374.17072092]
 [ -4599.73463069 -22423.70632464  14374.17072092 -16099.07120743
    4599.73463069   6324.6351172  -14374.17072092  32198.14241486]]
[0. 0. 0. 0. 0. 0. 0. 0.]
```

References

Simo, Juan C., and Thomas JR Hughes. Computational inelasticity. Vol. 7. Springer Science & Business Media, 2006

In [5]:
```
from IPython.core.display import HTML
def css_styling():
    styles = open('./nb_style.css', 'r').read()
    return HTML(styles)
css_styling()
```

Out[5]:

In [ ]:

User material subroutines

Introduction

This notebook discusses the implementation of user material subroutine `UMAT` . The particular subroutine corresponds to the classical metal plasticity model in a $J\text{-}2$ - theory formulation. The subroutine is for plane strain idealizations and it considers non-linear combined isotropic/kinematic hardening.

The formulation of the model is detailed in Simo and Hughes (1998). The user subroutine corresponds to the return mapping integration algorithm. An extended version of the model (considering damage and thermal effects) together with the integration algorithm is formulated in Gomez and Basaran (2004).

Subroutine interface (input and output parameters)

The following set of parameters is passed from the main program to the user subroutine:

- `stress:` (ndarray) Stress tensor at the current integration point at the beginning of the increment.
- `strann:` (ndarray) Total strains tensor at the current integration point.
- `dstran:` (ndarray) Incremental strains at the current integration point
- `statev:` (ndarray) Sate variables array at the current integration point at the beginning of the increment.
- `props:` (ndarray) SMaterial properties for the element
- `ntens:` (int) Number of components for the stress and strain tensors..

The following set of parameters is returned by the `UEL` subroutine to the main program:

- `C:` (ndarray) Constitutive tensor.
- `stress:` (ndarray) Updated stress tensor at the current integration point.
- `statev:` (ndarray) Sate variables array at the current integration point at the end of the increment.

In [1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import sympy as sym
from os import sys
sys.path.append("../source/")
from STRUCTURE import Struct_DYN
from postprocesor import *
from uel_solid import *
```

In [2]:
```
def umat_PCLK(stress , strann , dstran , statev , props , ntens):
    """Return mapping integration algorithm for J2-flow theory
       rate independent plasticity under plane strain conditions
       combined isotropic kinematic hardenig (see Simo, Juan C., and
       Thomas JR Hughes. Computational inelasticity. Vol. 7.
       Springer Science & Business Media, 2006).

    statev     : nd array
               Sate variables array at the current integration point
               at the beginning of the increment.
               4 components of the elastic strain tensor
               4 components of the plastic strain tensor
               4 components of the back stress tensor
               1 equivalent plastic strain
               1 equivalent stress
    """

    toler = 1.0e-7
    istop = 1

    eelas   = np.zeros([4])
    eplas   = np.zeros([4])
    xback   = np.zeros([4])
    flow    = np.zeros([4])
    eqplas = 0.0
    smises = 0.0

    statev , eelas , eplas , xback , eqplas , smises = stv_handl( 0 , statev , eelas ,
plas , xback , eqplas , smises , ntens)

    Emod       = props[0]
    enu        = props[1]
    eg2=Emod/(1.0+enu)
    eg = eg2/2.0
    sig0    = props[2]
    sigsat  = props[3]
    hrdrate = props[4]
    hmod    = props[5]

    C = elas_tensor(enu , Emod)
    stress = stress + np.dot(C , dstran)
    shydro =(stress[0]+stress[1]+stress[2])/3.0
    eelas  = eelas + dstran
    sdev = deviator(stress)
    stsrel = sdev - xback
    smises = vmises(stsrel)

    fbar = np.sqrt(2.0/3.0)*smises
    syiel0 , syieldk , ehardi , ehardk = uhardnlin(sig0 , sigsat , hrdrate , hmod , eq
as)
    syield   = syiel0
    syieldk0 = syieldk

    if fbar > (1.0+toler)*syiel0:
        flow = stsrel/fbar
        gam_par , eqplas , syieldk , istop = local_NR(syield, syieldk0 , ehardi , ehar
, hmod , sig0 , sigsat , hrdrate , eqplas , fbar , eg)
        for k in range(3):
            xback[k]  = xback[k] + np.sqrt(2.0/3.0)*(syieldk-syieldk0)*flow[k]
            eplas[k]  = eplas[k] + gam_par*flow[k]
```

```
            eelas[k]  = eelas[k] - eplas[k]
            stress[k] = flow[k]*syield + xback[k] + shydro
        xback[3] = xback[3] + np.sqrt(2.0/3.0)*(syieldk-syieldk0)*flow[3]
        eplas[3]  = eplas[3] + 2.0*gam_par*flow[3]
        eelas[3]  = eelas[3] - eplas[3]
        stress[3] = flow[3]*syield + xback[3]
        eqplas = eqplas+np.sqrt(2.0/3.0)*gam_par
        C = np.zeros((4 , 4))
        C = plas_tensor(gam_par , fbar , flow , ehardk , ehardi , enu , Emod)
#
# Store updated values of state variables
#
    statev , eelas , eplas , xback , eqplas , smises = stv_handl( 1 , statev , eelas ,
plas , xback , eqplas , smises , ntens)

    if istop == 0:
        print('local plasticty algorithm did not converged')
        print('After' , iter , 'iterations')
        print('Last value of the consistency parameter' , gam_par)

    return C , stress , statev
```

To test and execute the subroutine the following block of code emulates the required entries from the element subroutine to the material subroutine.

In [3]:

```
nstatev = 14
ntens   = 4
statev = np.zeros([nstatev])
stress = np.zeros([ntens])
strann = np.zeros([ntens])
dstran = np.zeros([ntens])
nprops = 7
props   = np.zeros([nprops])

dstran[0] =-0.0001/3.0
dstran[1] = 0.0001

props[0]= 52.0e3
props[1]= 0.33
props[2]= 60.0
props[3]= 37.0
props[4]= 383.3
props[5]= 2040.0
props[6]= 1000.0

C , stress , statev = umat_PCLK(stress , strann , dstran , statev , props , ntens)
```

Results

In [4]:

```
print(C)
print(stress)
print(statev)
```

```
[[77045.55506413 37947.81070323 37947.81070323     0.        ]
 [37947.81070323 77045.55506413 37947.81070323     0.        ]
 [37947.81070323 37947.81070323 77045.55506413     0.        ]
 [    0.             0.             0.         19548.87218045]]
[1.2265959  6.43962848 2.52985405 0.        ]
[-3.33333333e-05  1.00000000e-04  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  4.69896407e+00]
```

References

Simo, Juan C., and Thomas JR Hughes. Computational inelasticity. Vol. 7. Springer Science & Business Media, 1998.

Gomez, J and Basaran, C(2006) Damage mechanics constitutive model for Pb/Sn solder joints incorporating nonlinear kinematic hardening and rate dependent effects using a return mapping integration algorithm. mechanics of Materials. (38), 585-598.

In [5]:

```
from IPython.core.display import HTML
def css_styling():
    styles = open('./nb_style.css', 'r').read()
    return HTML(styles)
css_styling()
```

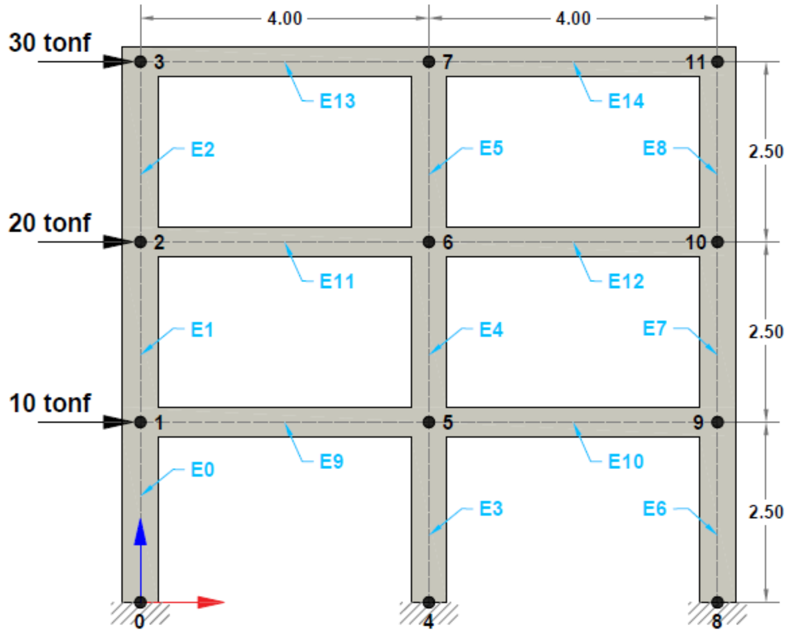
Out[5]:

In [ ]:

Linear 2D frame under horizontal static loads (NLSTA).

This problem describes the static analysis of a linear two-dimensional frame under point loads. The analysis is performed in a single step.

Input and output files for this problem are available in the examples folder of this REPO (notebooks\Examples).



- Element type for columns and beams: 2
- Columns cross section *0.50 m x 0.50 m*
- Beams cross section *0.40 m x 0.40 m*
- Material profile for all elements is concrete with elastic modulus of *2000000 tonf/m²* and specific weight of *2.4 tonf/m³*

Internal forces, together with a simple vertification of static global equilibrium are available in the file:

notebooks\Examples\Ex_01\Output.xls

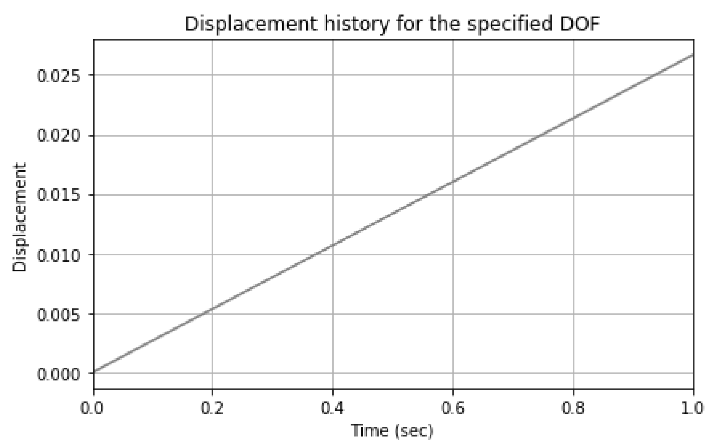
In [1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import sympy as sym
from os import sys
sys.path.append("../source/")
from STRUCTURE import Struct_DYN
from postprocesor import *

# Execute analysis
displacement,folder,IBC,nodes,elements,ninc,T,MvarsGen,ILFGen = Struct_DYN("Examples/E
01/01_INPUT/")
```

```
----------------------
Number of nodes: 12
Number of elements: 15
Number of equations: 27
Number of equations after constraints: 27
----------------------
Natural periods of the system :  Not computed,static system solution
----------------------
Time step for solution: 0.002 sec
Number of time increments: 500
----------------------
Duration for system solution: 0:00:00.931524
Duration for the system's solution: 0:00:00.932524
Duration for post processing: 0:00:00
----------------------
Analysis terminated successfully!
----------------------
```

Results

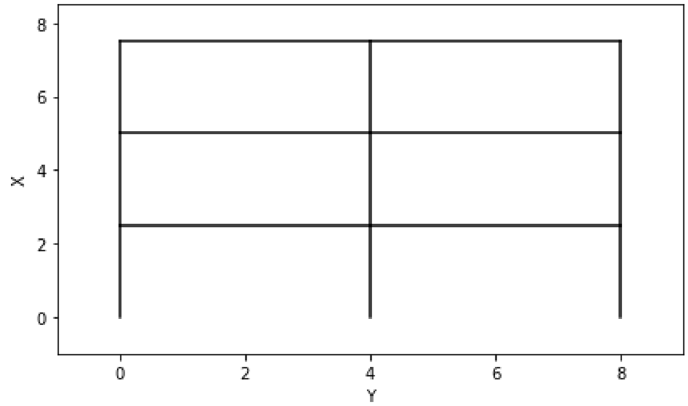
The displacement response along the horizontal direction for node 3 is shown below.

In [2]:
```
fig    = NodalDispPLT(displacement[6,:], T, ninc, ylabel = "Displacement")
```



The code can also display the structure under study for verification purposes.

In [3]:
```
model = GrafModel(elements, nodes)
```



In [4]:
```
from IPython.core.display import HTML
def css_styling():
    styles = open('./nb_style.css', 'r').read()
    return HTML(styles)
css_styling()
```

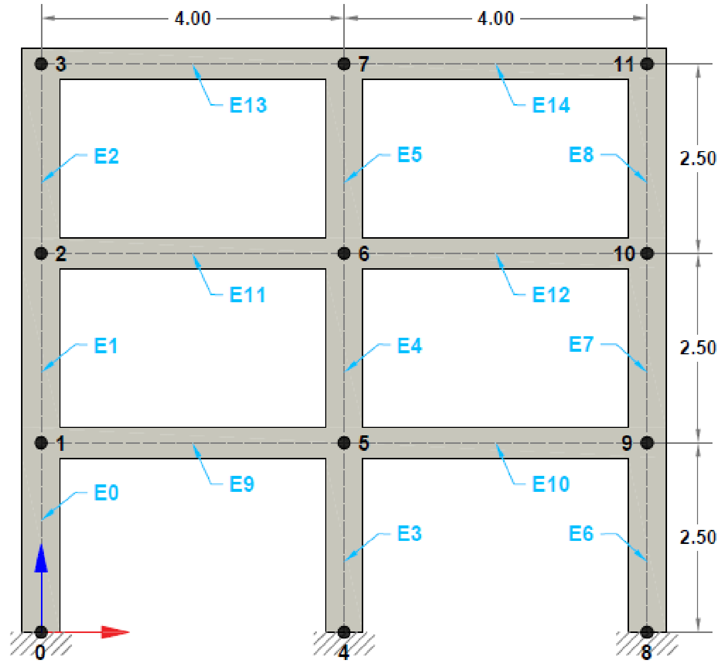
Out[4]:

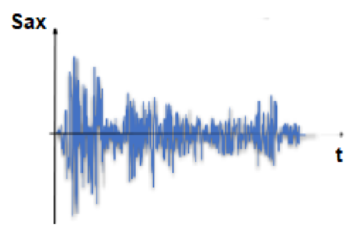
In [ ]:

Linear dynamic analysis of a 2D-frame.

The two-dimensional frame shown in the figure is subjected to a base acceleration corresponding to a record from El Centro earthquake. The base acceleration is imposed along the horizontal direction. The analysis is intended to obtain the natural frequencies of the system and to find the time history of the response. The analysis conisders mass contribution only along the horizontal direction.

Input and output files for this problem are available in the examples folder of this REPO (notebooks\Examples).





- Ground acceleration signal: *El Centro earthquake record (California, 1940)*.
- Time step for the excitation: *0.02 sec*
- Total time for the solution: *15.0 sec*
- Value of gravity: *9.806 m/s²*
- Element type for columns and beams: 2
- Columns cross sections *0.50 m x 0.50 m*
- Beams cross sections *0.40 m x 0.40 m*
- Material profile for all elements is concrete with an elastic modulus of *2000000 tonf/m²* and specific weight of *2.4 tonf/m³*

-Diagrams for axial and shear forces and bending moments for a selected time increment are written to the following file:
notebooks\Examples\Ex_02\Output.xls

In [1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import sympy as sym
from os import sys
sys.path.append("../source/")
from STRUCTURE import Struct_DYN
from postprocesor import *

# Execute analysis
displacement,folder,IBC,nodes,elements,ninc,T,MvarsGen,ILFGen = Struct_DYN("Examples/E
02/01_INPUT/")
```

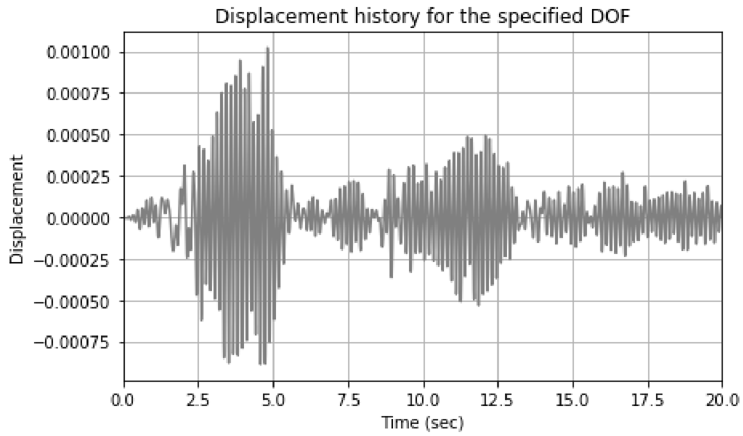
```
 ***** Time step and seismo signal has been updated *****
 ***** Warning: Total time of seismo signal is greater than solution total time *****
----------------------
Number of nodes: 12
Number of elements: 15
Number of equations: 27
Number of equations after constraints: 27
----------------------
Natural periods of the system :  [0.14946052 0.04736539 0.02562026 0.02327047 0.020859
2 0.01848281
 0.01794195 0.01734948 0.01609275 0.01542371 0.01420094 0.01410994
 0.01373219 0.01351975 0.01271046 0.01013722 0.00932614 0.00811226
 0.00664932 0.00644066 0.0059378  0.00561157 0.00502667 0.00502665
 0.00424963 0.00363439 0.00363439]
----------------------
Time step for solution: 0.0033333333333333335 sec
Number of time increments: 6000
----------------------
Finished initial conditions....: 0
Duration for system solution: 0:00:13.663503
Duration for the system's solution: 0:00:13.665530
Duration for post processing: 0:00:00
----------------------
Analysis terminated successfully!
----------------------
```

Results

Notice that in this analysis the time step initially specified for the analysis $\Delta t = 0.02s$ is too large for convergence thus forcing nldyna to adjust the time step.

The horizontal displacement time history at nodal point 3 is shown below:

In [2]:
```
fig = NodalDispPLT(displacement[6 ,:], T, ninc, ylabel = "Displacement")
```



In [3]:
```
from IPython.core.display import HTML
def css_styling():
    styles = open('./nb_style.css', 'r').read()
    return HTML(styles)
css_styling()
```

Out[3]:

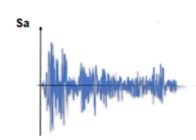
In [ ]:

Dynamic analysis of a three-dimensional frame with rigid diaphragm

Multiple point constrainsts ase those tipically required in the simulation of rigid diaphragms are allowed in `NLDYNA`. Its implementation to conduct dynamic analysis with rigid floors and other general constraints are explained in Notebook 03. These constraints are imposed using slaves degrees of freedom connected to a master nodel (corresponding to the center of gravity in rigid diaphragms). The implementation used here is that propossed by Wilson (1995).

This notebook describes the dynamic analysis (time history) of a three dimensional building model. The building is a fixed-base 3-story frame under ground acceleration. Rigid diaphragms are used to represent stiff concrete slabs at each floor.

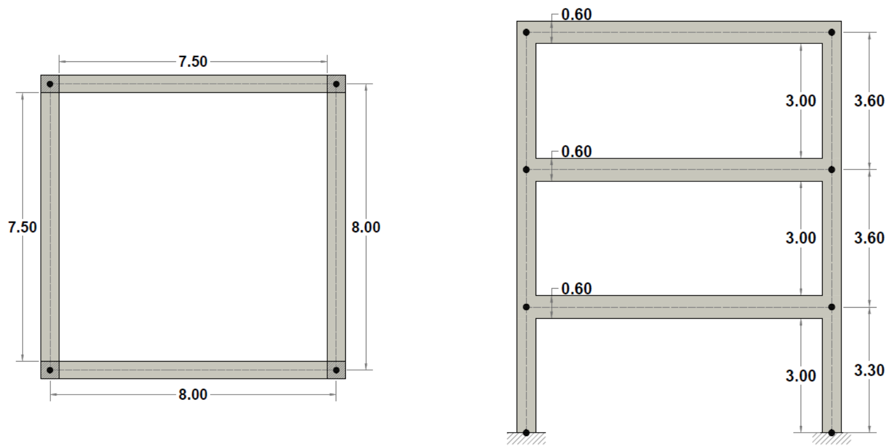
The 3-D frame elements used in the model correspond to type `10` in the `NLDYNA` library. These elements are based on the Timoshenko beam theory considering shear strain effects. Results are provided in terms of the displacements time history at point located in the third floor of the building.



The general parameters used in the analysis are defined next:

- Ground motion: acceleration time history corresponding to the *California 1940 El Centro record*.
- Time step of the input excitation: $\Delta t = 0.02s$.
- Total analysis tie: $T = 20.0s$.
- Element type as defined in NLDYNA: Timoshenko beams eltype = `10`.
- Columns cross sections *0.50 m x 0.50 m*
- Beams cross sections *0.50 m x 0.60 m*.
- Material profile for all elements is concrete with $E = 2000000$ tonf/m² ad $\gamma = 2.4$ tonf/m³.

The building together wit a plan view of the typical floor are shown below.



Units for this example are [tonf-m]**.

Input and output files for this problem are available in the examples folder of this REPO (notebooks\Examples).

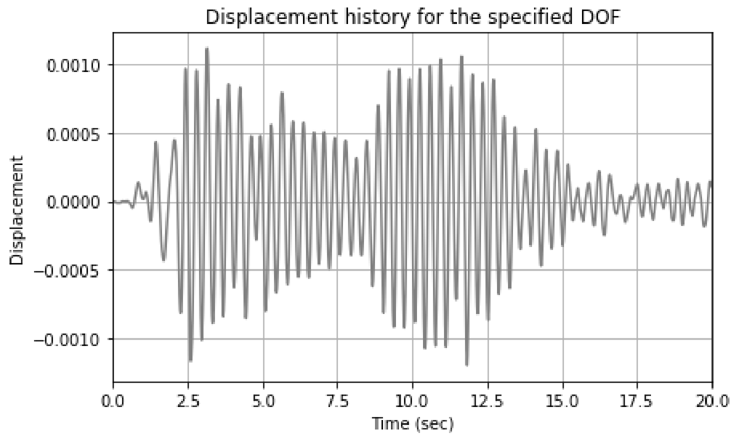
In [1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import sympy as sym
from os import sys
sys.path.append("../source/")
from STRUCTURE import Struct_DYN
from postprocesor import *

# Execute analysis
displacement,folder,IBC,nodes,elements,ninc,T,MvarsGen,ILFGen = Struct_DYN("Examples/E
03/01_INPUT/")
```

```
 ***** Time step and seismo signal has been updated *****
 ***** Warning: Total time of seismo signal is greater than solution total time *****
----------------------
Number of nodes: 16
Number of elements: 24
Number of equations: 72
Number of equations after constraints: 54
----------------------
Natural periods of the system :  [0.35223163+0.00000000e+00j 0.35223163+0.00000000e+00
 0.2204523 +0.00000000e+00j 0.1163864 +4.27099245e-17j
 0.1163864 -4.27099245e-17j 0.08686012+0.00000000e+00j
 0.08650117+0.00000000e+00j 0.07220895+0.00000000e+00j
 0.07220895+0.00000000e+00j 0.07155991+0.00000000e+00j
 0.06157273+0.00000000e+00j 0.05716461+0.00000000e+00j
 0.04636163+0.00000000e+00j 0.04405392+0.00000000e+00j
 0.04405392+0.00000000e+00j 0.0388679 +0.00000000e+00j
 0.0388679 +0.00000000e+00j 0.0370321 +0.00000000e+00j
 0.0370321 +0.00000000e+00j 0.03609697+0.00000000e+00j
 0.03609697+0.00000000e+00j 0.03583761+0.00000000e+00j
 0.03583761+0.00000000e+00j 0.0352326 +0.00000000e+00j
 0.03516515+0.00000000e+00j 0.03391587+0.00000000e+00j
 0.03391587+0.00000000e+00j 0.03240249+0.00000000e+00j
 0.03240249+0.00000000e+00j 0.031811  +0.00000000e+00j
 0.031811  +0.00000000e+00j 0.0303654 +0.00000000e+00j
 0.0303654 +0.00000000e+00j 0.02997751+0.00000000e+00j
 0.0298653 +0.00000000e+00j 0.02871485+0.00000000e+00j
 0.02869704+0.00000000e+00j 0.02869704+0.00000000e+00j
 0.02850893+0.00000000e+00j 0.02740397+0.00000000e+00j
 0.02709122+0.00000000e+00j 0.02554989+0.00000000e+00j
 0.02544222+0.00000000e+00j 0.02544222+0.00000000e+00j
 0.02508893+0.00000000e+00j 0.02508893+0.00000000e+00j
 0.0231168 +0.00000000e+00j 0.02261329+0.00000000e+00j
 0.02219773+0.00000000e+00j 0.02074471+0.00000000e+00j
 0.02074471+0.00000000e+00j 0.01929411+0.00000000e+00j
 0.01286052+0.00000000e+00j 0.01265509+0.00000000e+00j
 0.01265509+0.00000000e+00j 0.01245955+0.00000000e+00j
 0.01154935+0.00000000e+00j 0.01154935+0.00000000e+00j
 0.01145907+0.00000000e+00j 0.01145907+0.00000000e+00j
 0.01139858+0.00000000e+00j 0.01139858+0.00000000e+00j
 0.01131002+0.00000000e+00j 0.01131002+0.00000000e+00j
 0.01026239+0.00000000e+00j 0.01026239+0.00000000e+00j
 0.01018211+0.00000000e+00j 0.01018211+0.00000000e+00j
 0.00927072+0.00000000e+00j 0.0092024 +0.00000000e+00j
 0.0092024 +0.00000000e+00j 0.00913557+0.00000000e+00j]
----------------------
Time step for solution: 0.006666666666666667 sec
Number of time increments: 3000
----------------------
Finished initial conditions....: 0
Duration for system solution: 0:00:31.860763
Duration for the system's solution: 0:00:31.862759
Duration for post processing: 0:00:00.387988
----------------------
Analysis terminated successfully!
----------------------
```

Results

The time history of displacements at a nodal point located in the roof can be obtained as follows.

In [2]:
```
fig = NodalDispPLT(displacement[0,:], T, ninc, ylabel = "Displacement")
```



References

Wilson, Edward L. Three Dimensional Static and Dynamic Analysis Of Structures. Computers & Structures Inc, 1995.

The following cell just changes the Notebook format

In [3]:
```
from IPython.core.display import HTML
def css_styling():
    styles = open('./nb_style.css', 'r').read()
    return HTML(styles)
css_styling()
```

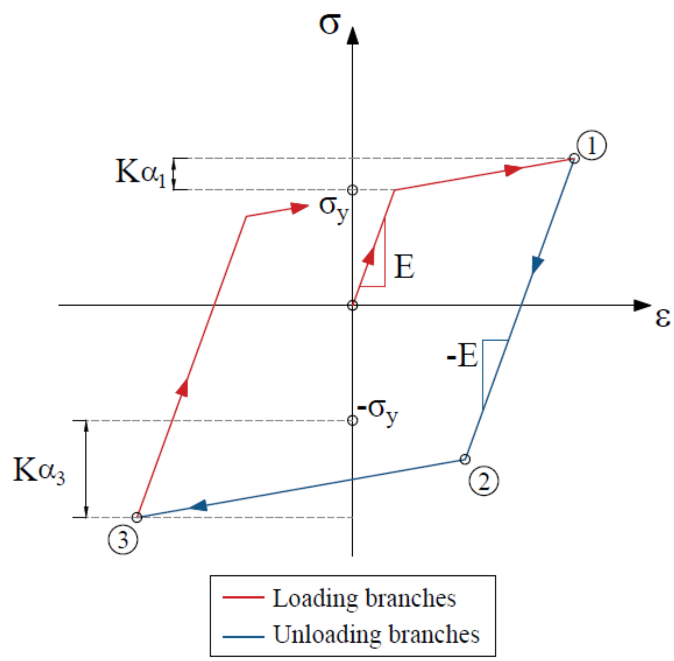
Out[3]:

In [ ]:

One-dimensional non-linear spring elements.

Simple spring elements are commonly used for the assemblage of complex structural systems like in the modelling of drilled shafts, plastic hinges in framed structures and base isolated buildings. This example describes the non-linear static analysis of a simple assemblage of spring elements. A pseudo-static load of total magnitude $100kgf$ is applied to the center node.

The constitutive model for the non-linear spring is the one formulated in Simo and Hughes (2006) corresponding to a rate independent plasticity model with linear isotropic hardening. The model is shown in the figure below.

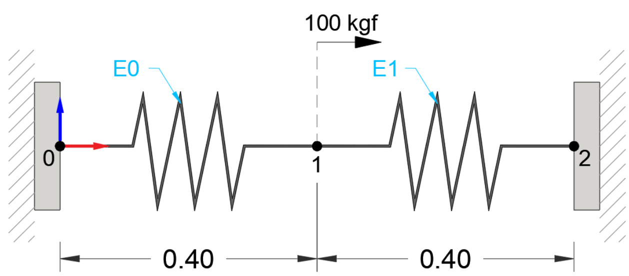


The stiffness coefficient for the spring is given in terms of a cross-section, Young's modulus and length.

$$K_{loc} = \begin{bmatrix} \frac{AE}{L} & \frac{-AE}{L} \\ \frac{-AE}{L} & \frac{AE}{L} \end{bmatrix}$$

Input and output files for this problem are available in the examples folder of this REPO (notebooks\Examples).

The following 1D spring assembly was analyzed. Static nodal force was applied at nodes 1 as it is shown at the figure. For this example it was used **[kgf-m]** as consistent units for the analysis.



- Element type: 5
- Cross sectioal area, A: *0.25 m²*
- Young modulus, E: *100000 kgf/m²*
- Yield stress, σ_y: *150 kgf/m²*
- Strain hardening parameter, K: *10000 kgf/m²*

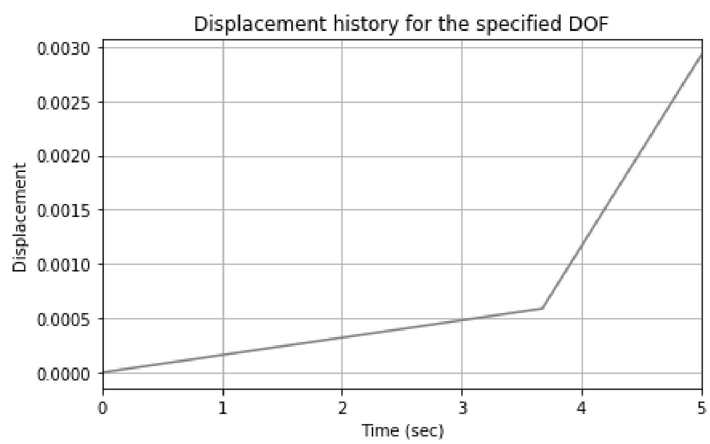
In [5]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import sympy as sym
from os import sys
sys.path.append("../source/")
from STRUCTURE import Struct_DYN
from postprocesor import *

# Execute analysis
displacement,folder,IBC,nodes,elements,ninc,T,MvarsGen,ILFGen = Struct_DYN("Examples/E
04/01_INPUT/")
```

```
----------------------
Number of nodes: 3
Number of elements: 2
Number of equations: 1
Number of equations after constraints: 1
----------------------
Natural periods of the system :  Not computed,static system solution
----------------------
Time step for solution: 0.1 sec
Number of time increments: 50
----------------------
Convergency reached after  1  iterations at increment  36  ( 3.6 sec)
Duration for system solution: 0:00:00.024943
Duration for the system's solution: 0:00:00.024943
Duration for post processing: 0:00:00
----------------------
Analysis terminated successfully!
----------------------
```

Results

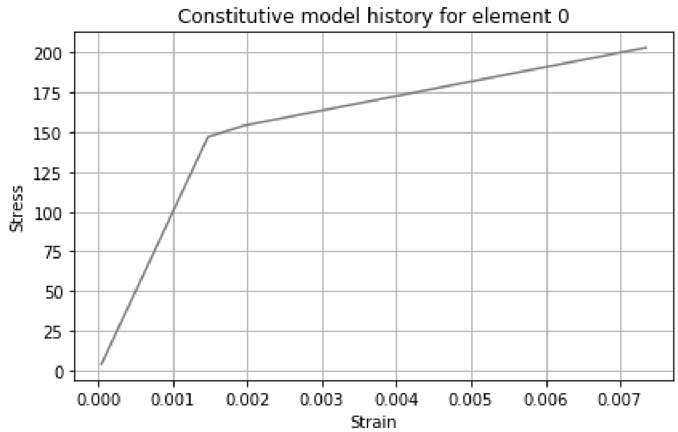
The following figure shows the displacement time history for the loaded node.

In [6]:
```
fig = NodalDispPLT(displacement[0,:], T, ninc, ylabel = "Displacement")
```



The observed slope change occurs at the time increment where the inelastic behavior takes place (3.6 s). The particular bi-lineal shape of the dispalcement response is controlled by the bi-lineal constitutive behavior of the nonlinear sprinmgs. The associated stress-strain curve for element `0` is shown next.

In [7]:
```
histe = PlasModel(MvarsGen, Element = 0, xlabel = "Strain", ylabel = "Stress")
```



It is observed that inelastic response appears when the stress in the spring reaches the prescribed value of 150 kgf/m².

References

Simo, Juan C., and Thomas JR Hughes. Computational inelasticity. Vol. 7. Springer Science & Business Media, 2006

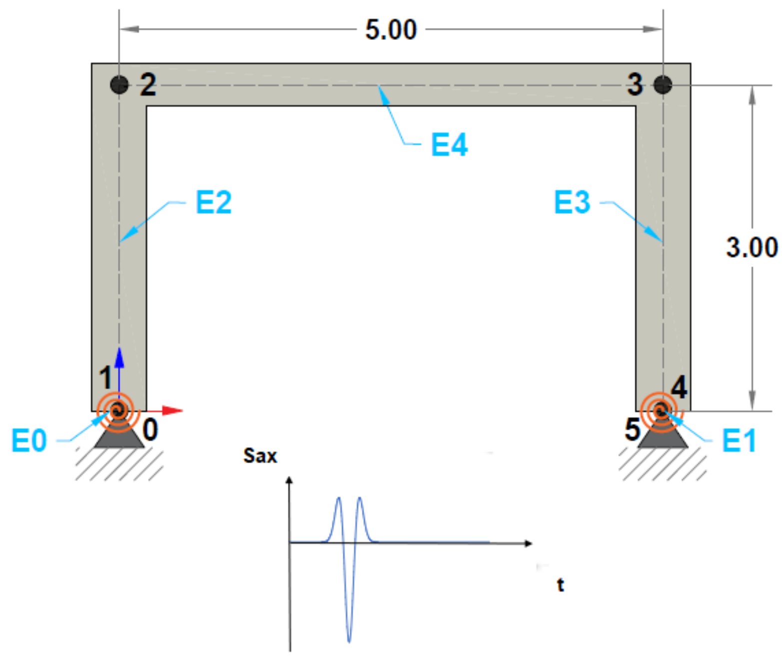
In [8]:
```
from IPython.core.display import HTML
def css_styling():
    styles = open('./nb_style.css', 'r').read()
    return HTML(styles)
css_styling()
```

Out[8]:

In [ ]:

Two-dimensional base isolated frame.

This example discusses a simple base isolated two-dimensional frame under ground motion acceleration. The base isolation system is represented by non-linear rotational springs located between the column edges and the foundation system. The base acceleration is a Ricker pulse (see figure) applied along the horizontal direction.



The constitutive model for the non-linear spring is the one formulated in Simo and Hughes (2006) corresponding to a rate independent plasticity model with linear isotropic hardening. The non-inear springs at the base in this particular case provide rotational stiffness at the base of the first floor columns. This rotational stiffness is defined like:

$$K_{loc} = \begin{bmatrix} \frac{4EI}{L} & \frac{-4EI}{L} \\ \frac{-4EI}{L} & \frac{4EI}{L} \end{bmatrix}$$

where:

- E = Material's young modulus.
- I = Column's base moment of inertia along "Z" axis.
- L = Column's lenght.

Input and output files for this problem are available in the examples folder of this REPO (notebooks\Examples).

Units for this example are **[kgf-m]**.

The main problem parameters are described next.

- Ground acceleration signal: *Ricker pulse*
- Ricker's central time, *Tc = 1.5 sec*
- Ricker's central frequency, *fc = 1.5 Hz*
- Maximum acceleration value: *1.0 g*
- Time step for the input excitation: *0.002 sec*
- Size if the analysis time window: *15.0 sec*
- Element type for frame elements: 2
- Element type for nonlinear 1D rotational springs: 8
- Cross section for all the elements: *0.60 m x 0.60 m*
- Material profile for the frame elements withelastic modulus of *500000 kgf/m²* and specific weight of *2000 kgf/m³*
- Material profile for the rotational springs with an elastic modulus, E = *100000 kgf/m²*, yield strees, σ_y = *2000 kgf/m²* and isotropic hardening parameter, K = *15000 tonf/m²*.

-Diagrams for axial and shear forces and bending moments for a selected time increment are written to the following file:
notebooks\Examples\Ex_02\Output.xls

In [1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import sympy as sym
from os import sys
sys.path.append("../source/")
from STRUCTURE import Struct_DYN
from postprocesor import *

# Execute analysis
displacement,folder,IBC,nodes,elements,ninc,T,MvarsGen,ILFGen = Struct_DYN("Examples/E
05/01_INPUT/")
```

```
----------------------
Number of nodes: 6
Number of elements: 5
Number of equations: 8
Number of equations after constraints: 8
----------------------
Natural periods of the system :  [3.95578795 1.21821887 0.98266225 0.60780604 0.582759
6 0.43858917
 0.43434444 0.39161916]
----------------------
Time step for solution: 0.002 sec
Number of time increments: 7500
----------------------
Finished initial conditions....: 0
Convergency reached after  1  iterations at increment  699  ( 1.398 sec)
Convergency reached after  1  iterations at increment  890  ( 1.78 sec)
Convergency reached after  1  iterations at increment  1085  ( 2.17 sec)
Convergency reached after  1  iterations at increment  1235  ( 2.47 sec)
Convergency reached after  1  iterations at increment  1991  ( 3.982 sec)
Convergency reached after  1  iterations at increment  2571  ( 5.142 sec)
Duration for system solution: 0:00:06.188326
Duration for the system's solution: 0:00:06.189336
Duration for post processing: 0:00:00
----------------------
Analysis terminated successfully!
----------------------
```

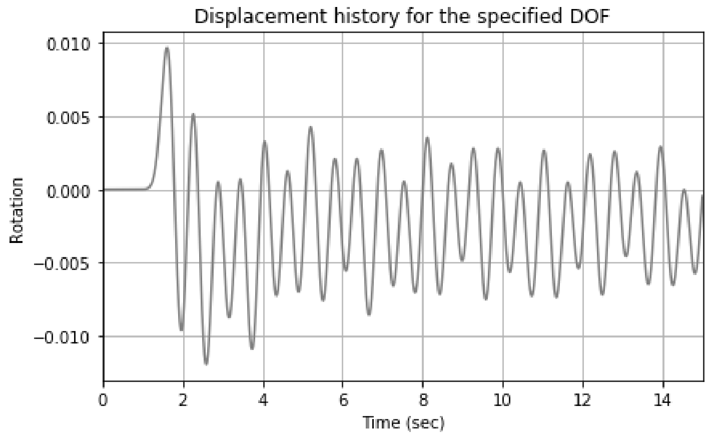
Results

In a previous analysis the first 3 natural modes of the non-isolated building were found to be: **2.942 sec, 1.125 sec, 0.891 sec**.

The natural periods of the isolated building are found to be: **3.956 sec, 1.218 sec, 0.923 sec**.

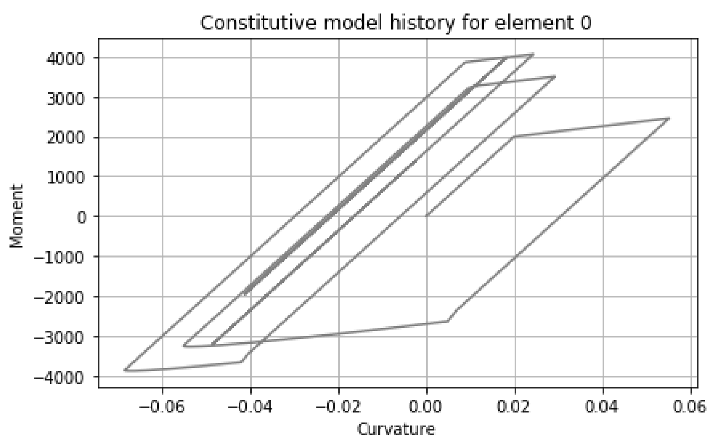
On the other hand, the analysis shows that the rotational spring (element `0`) exhibits plastic behavior during the dynamic excitation.

In [2]:
```
fig = NodalDispPLT(displacement[0,:], T, ninc, ylabel = "Rotation")
```



Due to the inelastic response of the rotational spring the system now oscillates around a permanent deformed configuration. The Moment-curvature history for the rotational spring is shown below.

In [3]:
```
histe = PlasModel(MvarsGen, Element = 0, xlabel = "Curvature", ylabel = "Moment")
```



References

Simo, Juan C., and Thomas JR Hughes. Computational inelasticity. Vol. 7. Springer Science & Business Media, 2006

In [4]:
```
from IPython.core.display import HTML
def css_styling():
    styles = open('./nb_style.css', 'r').read()
    return HTML(styles)
css_styling()
```

Out[4]:

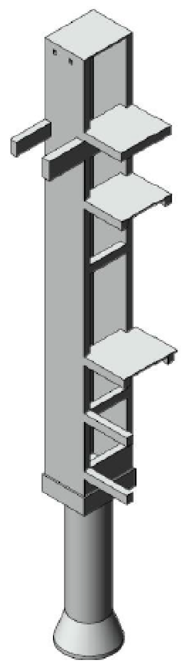
In [ ]:

Response of a laterally-loaded pile foundation.

The following example discusses the design of a laterally-loaded pile foundation. The fundamental problem is that of finding the required length of a 2.20m diameter pile under an externally applied lateral load and bending moment. The mechanical properties of the soil deposit are given in table 01. The analysis is conducted for the maximum lateral load and bending moment under service conditions.

Units for this example are **[tonf-m]**.

Input and output files for this problem are available in the examples folder of this REPO (notebooks\Examples).



The main problem parameters are described next.

Mechanical properties.

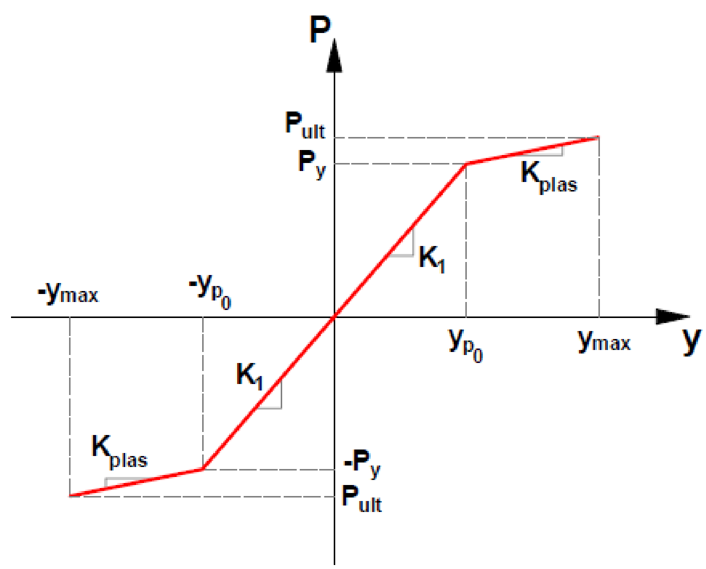
The following set of parameters defines the soil deposit at the site:

- Y : Depth.
- K_h : Lateral soil stiffness at a depth Y.
- σ_{ult} : Maximum stress in the soil at a depth Y.

Depth	Kh	σ_{ult}
0.0 m	917 tonf/m³	21.6 tonf/m²
5.5 m	4280 tonf/m³	21.6 tonf/m²
7.5 m	9070 tonf/m³	54.0 tonf/m²
12.5 m	24470 tonf/m³	90.0 tonf/m²

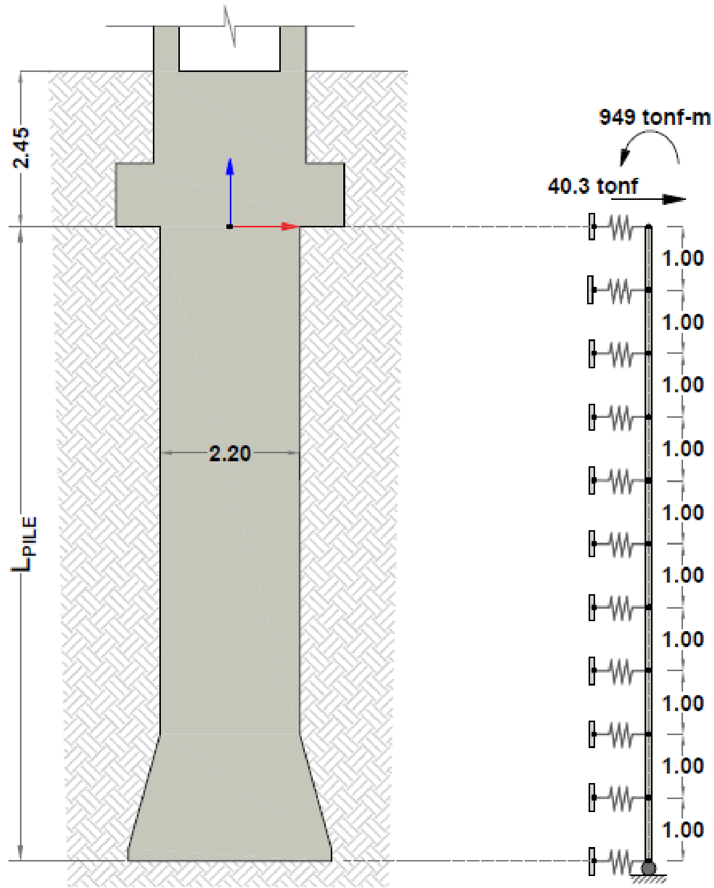
Table 01. Soil properties

- Soil stiffness is represented by a bilinear P-Y curve according to the variation with depth of K_h.



Structural idealization for the pile

Concrete drilled shaft



- Element type: 2
- Pile cross sectional diameter: *2.20 m*
- Material profile for the pile is concrete with an elastic modulus of *2527000 tonf/m²* and unit weight of *2.4 tonf/m³*

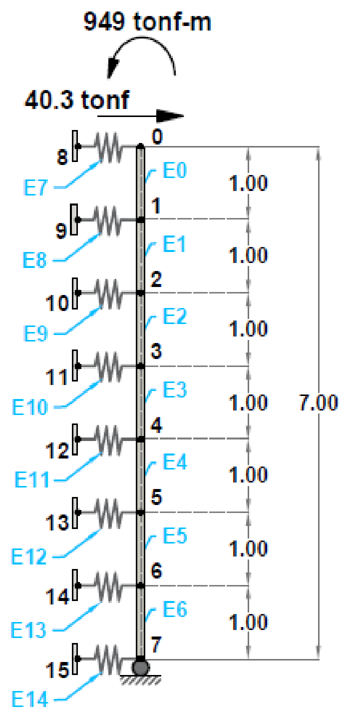
Soil springs

- Element type: 9
- Material profile for the springs according to the variation with depth of K_h:

Depth	K_1	Yp_0	K_{plas}	P_{ult}
0.0 m	2018 tonf/m	0.024 m	0 tonf/m	95 tonf
1.0 m	3363 tonf/m	0.014 m	0 tonf/m	95 tonf
2.0 m	4708 tonf/m	0.010 m	0 tonf/m	95 tonf
3.0 m	6053 tonf/m	0.008 m	0 tonf/m	95 tonf
4.0 m	7398 tonf/m	0.006 m	0 tonf/m	95 tonf
5.0 m	8743 tonf/m	0.005 m	0 tonf/m	95 tonf

The pile response was computed for a pile length of `7.0m` .

Identifiers for nodal points, elements and other problem dimensions are shown below:



The springs located at the `0.0m` and `7.0m` level employ `50%` of the total stiffness as per table 02.

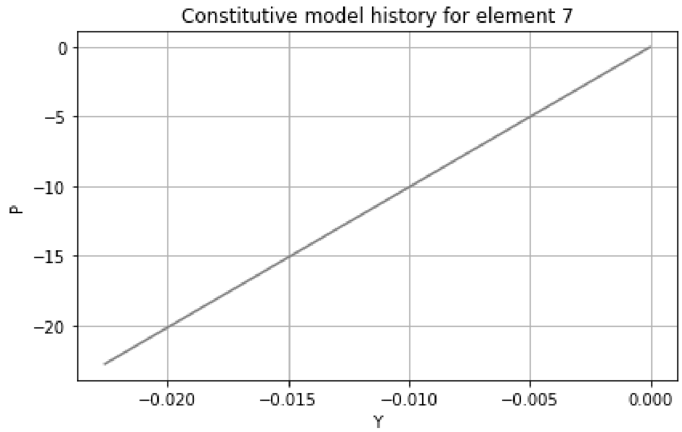
In [1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import sympy as sym
from os import sys
sys.path.append("../source/")
from STRUCTURE import Struct_DYN
from postprocesor import *

# Execute analysis
displacement,folder,IBC,nodes,elements,ninc,T,MvarsGen,ILFGen = Struct_DYN("Examples/E
06/01_INPUT/")
```

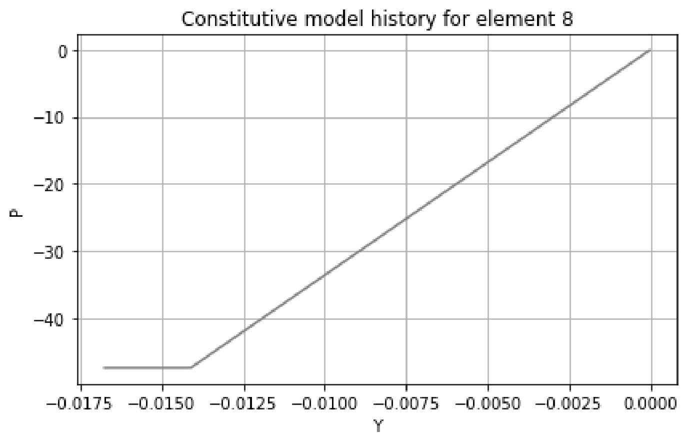
```
----------------------
Number of nodes: 16
Number of elements: 15
Number of equations: 23
Number of equations after constraints: 23
----------------------
Natural periods of the system :  Not computed,static system solution
----------------------
Time step for solution: 0.01 sec
Number of time increments: 500
----------------------
Convergency reached after  1  iterations at increment  334  ( 3.34 sec)
Convergency reached after  1  iterations at increment  462  ( 4.62 sec)
Convergency reached after  1  iterations at increment  481  ( 4.81 sec)
Duration for system solution: 0:00:00.767947
Duration for the system's solution: 0:00:00.768939
Duration for post processing: 0:00:00
----------------------
Analysis terminated successfully!
----------------------
```

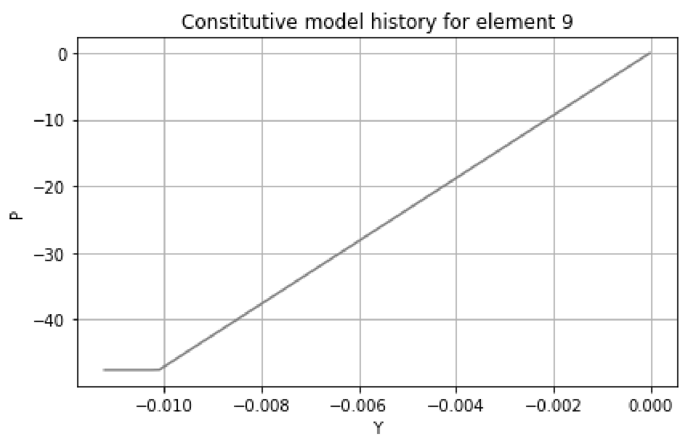
Results

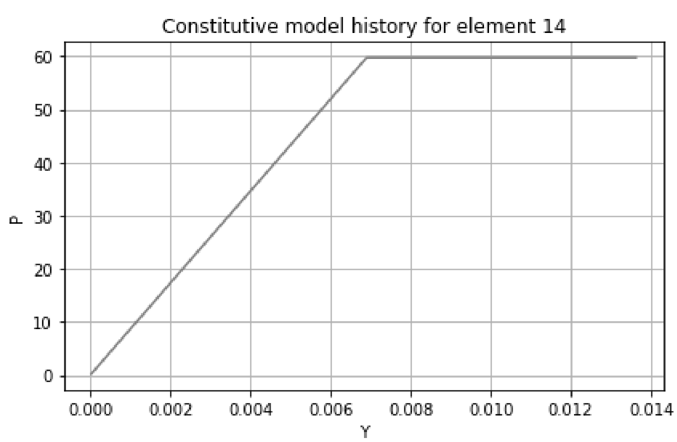
The evolution of the soil response in terms of the P-Y curves for the springs undergoing plastic behavior is shown next. The plastified springs are identified by elements `8`, `9` and `14`.

In [2]:
```
histe = PlasModel(MvarsGen, Element = 7, xlabel = "Y", ylabel = "P")
histe = PlasModel(MvarsGen, Element = 8, xlabel = "Y", ylabel = "P")
histe = PlasModel(MvarsGen, Element = 9, xlabel = "Y", ylabel = "P")
histe = PlasModel(MvarsGen, Element = 14, xlabel = "Y", ylabel = "P")
```


Constitutive model history for element 7
0
−5
−10
−15
−20
P
−0.020
−0.015
−0.010
−0.005
0.000
Y


Constitutive model history for element 8
0
−10
−20
−30
−40
P
−0.0175
−0.0150
−0.0125
−0.0100
−0.0075
−0.0050
−0.0025
0.0000
Y


Constitutive model history for element 9
0
−10
−20
−30
−40
P
−0.010
−0.008
−0.006
−0.004
−0.002
0.000
Y

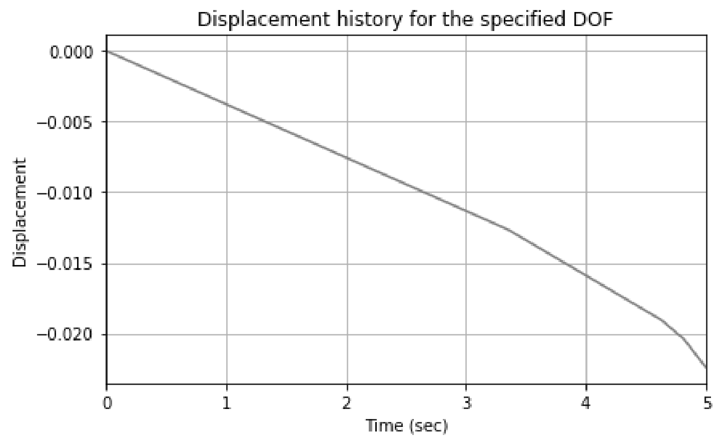

Constitutive model history for element 14
60
50
40
30
20
10
0
P
0.000
0.002
0.004
0.006
0.008
0.010
0.012
0.014
Y

As expected the spring elements with low rigidity exhibit plastic response (elements 7, 8 and 9) since these elements are associated to materials with low capacity to sustain lateral loads. Typically the soil at large depths exhibits a larger stiffness. Although element 7 did not experience inelastic response its maximum strain was close the plastic limit of the material Y_{p_0}.

On the other hand, element 14, corresponding to the spring associated to the soil layer with larger stiffness along the pile length undergoes plastic behavior since the found strains are larger than the plastic limit.

The predicted displacement time history along the horizontal direction for the node at the top of the pile is shown in the figure below:

In [3]:
```
fig   = NodalDispPLT(displacement[0,:], T, ninc, ylabel = "Displacement")
```



Internal forces, together with a simple displacement diagram are available in the file: *...\01_NoteBooks\Examples\Ex_05\Output.xls*

In [4]:
```
from IPython.core.display import HTML
def css_styling():
    styles = open('./nb_style.css', 'r').read()
    return HTML(styles)
css_styling()
```

Out[4]:

In [ ]: