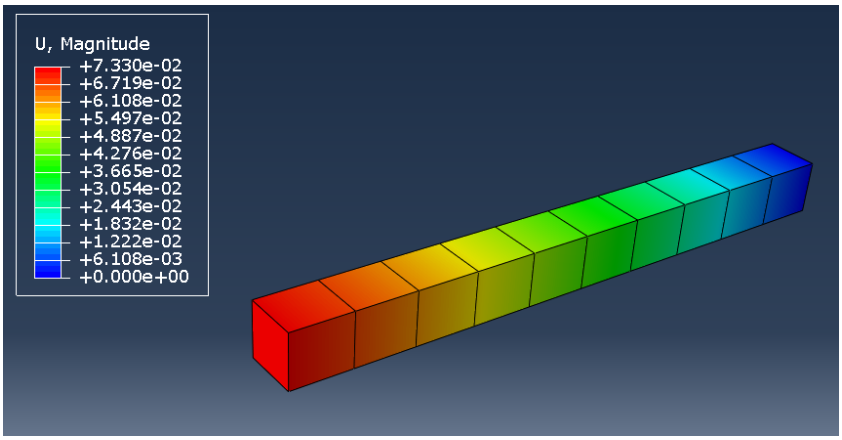
Métodos Computacionales en Ingeniería Civil

Session #3, Topic 1.
Finite element models for an elastic 1D bar



José M.ª Goicolea, Pedro Navas, Juan José Arribas, A. Cámara

Grupo de Mecánica Computacional, UPM

16-17 de febrero de 2022

Índice

1. Objetives

This session focuses on the development of a Finite Element (FE) code in the programming language `Python` that can solve a 1D elastic problem in which a bar is subject to axial loading. In addition, the Appendix includes the same code in the programming language `Matlab/Octave`. We adopt the following assumptions and conditions:

1. The students have the basic theoretical background regarding the construction of elementary stiffness matrices, which should be coded by the studentship.
2. In addition, these elementary matrices will become part of a global system through their assembly, a process that is also known by the students.
3. In order to solve the system of equations it is necessary to know the boundary conditions of loading and displacements in the problem. These need to be applied to the matricial system developed in `Python`.

Furthermore, we are going to simulate a 1D bar analogous to that solved in `Python` using the commercial FE software suite `Abaqus`. This will help assimilating the basic concepts of pre-processing, analysis and post-processing that will support the following practical sessions. The results obtained with the two methods will be compared with the analytical (exact) solution to the problem, which should be obtained by the student too, using the available plotting tools for this purpose.

2. Theoretical background and FE discretisation

2.1. Governing equations

The governing equation of a 1D linear elastic bar subject to axial loads is an elliptic differential equation in terms of the along-bar displacement $u(x)$:

$$\frac{\mathrm{d}(\mathrm{A}\sigma)}{\mathrm{d}x} + q(x) = \frac{\mathrm{d}}{\mathrm{d}x}\left(EA\frac{\mathrm{d}u}{\mathrm{d}x}\right) + q(x) = 0 \tag{1}$$

Where E is the Young's modulus, an elastic parameter that describes its stiffness, $q(x)$ represents the distributed forces along the bar per unit length x. The solution to the problem requires finding a displacement distribution $u(x)$ in the open interval of the length of the bar $(0, L)$ so that the system is in equilibrium considering the boundary conditions defined for the problem.

2.2. Boundary conditions and volumetric forces

The possible boundary conditions that we will find in the problem to solve can be summarise d in Fig. 1.

We can find basically two types of boundary conditions:

- *Type Dirichlet or Essentials*
  These boundaries are applied to the response field in which we have defined the governing differential equation; in our case this is $u(x)$ and therefore the Dirichlet boundary conditions refer to "displacements" at specific points (usually the ends of the bar):

$$u(0) = u_0$$

$$u(L) = u_g$$

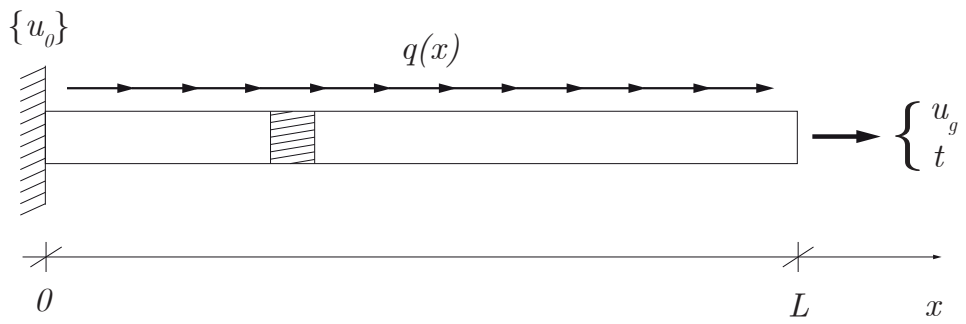


Figura 1: Problem definition and boundary conditions.

- *Type Neumann or Naturals*
  These boundaries are applied to the spatial derivative of the response field in which we have defined the governing differential equation, which considering our case will be:

$$t = EA\frac{\mathrm{d}u}{\mathrm{d}x}\bigg|_{x=L}$$

  Therefore, these could be considered boundary conditions related to "forces" for this problem.

On the other hand, we have the *volumetric or distributed forces*, which are those applied to the whole domain of the structure. In Eq. (1) these forces are defined by $q(x)$. In general they depend on the position x, but they can also be constant. An example of these forces in the mechanical problem are those associated with gravity g, which depend on the area of the cross-section of the bar $A(x)$ and its density $\rho(x)$ that can also be variable along the length:

$$q(x) = A(x)\rho(x)g$$

In this session, $q(x)$ will be a function of x that is given to represent a load per unit length in the bar.

2.3. Strong formulation and analytical solution tothe problem

Eq. (1) is the strong formulation of the problem. It is called like this because its solution is satisfied at any point of the domain $(0, L)$.

As we can see, this strong formulation involves a second derivative of $u(x)$, which implies that his field has to be differentiable of order two with respect to x.

The solution of the strong formulation leads to an analytical solution that is exact if we assume that the area A and the Young's modulus E are constant in the entire bar. We start with the strong formulation of the displacement field between an arbitrary point $y \in (0, L)$ and the end of the bar at $x = L$:

$$A\int_y^L \frac{\mathrm{d}\sigma}{\mathrm{d}x}\mathrm{d}x = -\int_y^L q(x)\mathrm{d}x \tag{2}$$

$$\sigma(L) - \sigma(y) = -\frac{1}{A}\int_y^L q(x)\mathrm{d}x \tag{3}$$

Adopting the definition of the 1D linear elastic constitutive equation, $\sigma = E\,\mathrm{d}u/\mathrm{d}x$:

$$E\frac{\mathrm{d}u(y)}{\mathrm{d}y} = E\frac{\mathrm{d}u}{\mathrm{d}x}\bigg|_{x=L} + \frac{1}{A}\int_y^L q(x)\mathrm{d}x \tag{4}$$

Again, we integrate between the start of the bar at $x = 0$ and a point $z \in (0, L)$:

$$\int_0^z E\frac{\mathrm{d}u(y)}{\mathrm{d}y}\mathrm{d}y = \int_0^z \left(E\frac{\mathrm{d}u}{\mathrm{d}x}\bigg|_{x=L} + \frac{1}{A}\int_y^L q(x)\mathrm{d}x\right)\mathrm{d}y \tag{5}$$

Assuming that the derivative of u with respect to x is known at $x = L$ (Neumann boundary condition), the result of the integral is:

$$Eu(z) - Eu(0) = E\frac{\mathrm{d}u}{\mathrm{d}x}\bigg|_{x=L} z + \frac{1}{A}\int_0^z\int_y^L q(x)\mathrm{d}x\mathrm{d}y \tag{6}$$

Therefore, the solution of $u(z)$ is:

$$u(z) = u(0) + \frac{\mathrm{d}u}{\mathrm{d}x}\bigg|_{x=L} z + \frac{1}{EA}\int_0^z\int_y^L q(x)\mathrm{d}x\mathrm{d}y \tag{7}$$

The analytical solution of Eq. (7) depends on two constants that need to be determined by imposing the boundary conditions at the two ends of the bar. In our case these are: $u(0)$ and $\mathrm{d}u/\mathrm{d}x|_{x=L}$, which correspond to types Dirichlet and Neumann, respectively. In a different case in which the boundary condition at the free end is not natural but based on displacements at $x = L$, the term $\mathrm{d}u/\mathrm{d}x|_{x=L}$ can be obtained by imposing $z = L$, with $u(L)$ being given.

For the specific scenario in which the boundary conditions refer to a bar fixed at the left end and with given axial load at the right end; $u(0) = 0$ and $EA\mathrm{d}u/\mathrm{d}x|_L = P$ (load P at $x = L$). And if the distributed axial load is linear with x so that $q(x) = q_0 + rx$, the analytical expression is:

$$\begin{aligned} u(x) &= \frac{1}{EA}\left[\left(P + q_0 L + r\frac{L^2}{2}\right)x - \left(q_0\frac{x^2}{2} + r\frac{x^3}{6}\right)\right] \\ \sigma(x) = E\frac{\mathrm{d}u}{\mathrm{d}x} &= \frac{1}{A}\left[P + q_0(L - x) + \frac{r}{2}(L^2 - x^2)\right] \end{aligned} \tag{8}$$

Considering the hypothesis made, we can only obtain analytical expressions for relatively simple distributions of $E(x)$ and $q(x)$, but not for more complex situations. The latter can be treated numerically in a simpler way, using the weak formulation of the FE method, although the result is generally not exact.

2.4. Weak formulation

The weak formulation is obtained by multiplying the strong formulation with arbitrary weights $w(x)$ that is then integrated in the domain:

$$\int_0^L w\frac{\mathrm{d}}{\mathrm{d}x}\left(EA\frac{\mathrm{d}u}{\mathrm{d}x}\right)\mathrm{d}x + \int_0^L wq\mathrm{d}x = 0 \tag{9}$$

Integrating by parts the first term:

$$\frac{\mathrm{d}}{\mathrm{d}x}\left[w \cdot EA\frac{\mathrm{d}u}{\mathrm{d}x}\right] = \frac{\mathrm{d}w}{\mathrm{d}x} \cdot EA\frac{\mathrm{d}u}{\mathrm{d}x} \quad + w \cdot \frac{\mathrm{d}}{\mathrm{d}x}\left(EA\frac{\mathrm{d}u}{\mathrm{d}x}\right)$$

$$\left[w \cdot EA\frac{\mathrm{d}u}{\mathrm{d}x}\right]_0^L - \int_0^L \frac{\mathrm{d}w}{\mathrm{d}x} \cdot EA\frac{\mathrm{d}u}{\mathrm{d}x}\mathrm{d}x = \int_0^L w \cdot \frac{\mathrm{d}}{\mathrm{d}x}\left(EA\frac{\mathrm{d}u}{\mathrm{d}x}\right)\mathrm{d}x \tag{10}$$

we obtain the following result by substituting the Eq. (9):

$$\int_0^L \frac{\mathrm{d}w}{\mathrm{d}x} \cdot EA\frac{\mathrm{d}u}{\mathrm{d}x}\mathrm{d}x = \int_0^L wq\,\mathrm{d}x + w(L)t_L - w(0)t_0 \tag{11}$$

where t_L y t_0 are the natural boundary conditions, which physically correspond to the forces applied at the two ends of the structure.

If an essential boundary condition of imposed or fixed (zero) displacement is applied at the end $x = 0$ the weighting function is also $w(0) = 0$, therefore the last term of Eq. (11) vanishes.

2.5. Funciones de forma

In the FE method an approximation of the unknown displacement field $u(x)$ using the so-called *shape functions* is needed. These functions interpolate the displacement field within the element based on the nodal displacements and have the expression:

$$u(x) \approx u_h(x) = \sum_{B=1}^{N_{\text{nod}}} u_B N_B(x) \tag{12}$$

A discrete set of (N_{nod}) shape functions $N_B(x)$ is used. This implies certain error which in principle is reduced with the more nodes N_{nod}, or increasing the order of the interpolating functions.

The Galerkin method is the most common one to define the weighting functions $w(x)$ and it is one that will be used in this session. This method proposes using the same weighting functions as the shape functions used to interpolate $u(x)$:

$$w(x) \approx w_h(x) = \sum_{A=1}^{N_{\text{nod}}} w_A N_A(x) \tag{13}$$

The interpolation of the derivatives that appear in the weak formulation is:

$$\frac{\mathrm{d}u_h}{\mathrm{d}x} = \sum_{B=1}^{N_{\text{nod}}} u_B \frac{\mathrm{d}N_B}{\mathrm{d}x}; \quad \frac{\mathrm{d}w_h}{\mathrm{d}x} = \sum_{A=1}^{N_{\text{nod}}} w_A \frac{\mathrm{d}N_A}{\mathrm{d}x} \tag{14}$$

substituting the integrals of the weak formulation (Eq. (11)):

$$\int_0^L \left(\sum_{A=1}^{N_{\text{nod}}} w_A \frac{\mathrm{d}N_A}{\mathrm{d}x}\right) EA \left(\sum_{B=1}^{N_{\text{nod}}} u_B \frac{\mathrm{d}N_B}{\mathrm{d}x}\right)\mathrm{d}x = \sum_{A,B=1}^{N_{\text{nod}}} w_A \left[\int_0^L \frac{\mathrm{d}N_A}{\mathrm{d}x} EA \frac{\mathrm{d}N_B}{\mathrm{d}x}\mathrm{d}x\right] u_B \tag{15}$$

where

$$\int_0^L \frac{\mathrm{d}N_A}{\mathrm{d}x} EA \frac{\mathrm{d}N_B}{\mathrm{d}x}\mathrm{d}x = K_{AB}$$

which leads to the stiffness matrix $[\boldsymbol{K}]$. Analogously, with the applied actions we obtain the terms of the forcing vector:

$$\int_0^L \left(\sum_{A=1}^{N_{\text{nod}}} w_A N_A \right) q \, \mathrm{d}x = \sum_{A=1}^{N_{\text{nod}}} w_A \underbrace{\left[\int_0^L N_A q \, \mathrm{d}x \right]}_{f_A^{\text{vol}}} \tag{16}$$

$$\left[w \, EA \, \frac{\mathrm{d}u}{\mathrm{d}x} \right]_0^L = w(L)t_L - w(0)t_0 = \sum_{A=1}^{N_{\text{nod}}} w_A f_A^{\text{ext}} \tag{17}$$

From Eqs. (15), (16) and (17) the following linear system of algebraic equations results:

$$\sum_{A,B=1}^{N_{\text{nod}}} w_A K_{AB} u_B = \sum_{A=1}^{N_{\text{nod}}} w_A \left(f_A^{\text{vol}} + f_A^{\text{ext}} \right) = \sum_{A=1}^{N_{\text{nod}}} w_A f_A \tag{18}$$

where the applied forces are obtained as the sum of the volumetric ones (distributed in the interior of the bar) and the exterior ones at the boundaries

$$f_A = f_A^{\text{vol}} + f_A^{\text{ext}}$$

And considering that $w(x)$ are arbitrary, w_A will be arbitrary too and therefore the following matricial equation yields:

$$\sum_{B=1}^{N_{\text{nod}}} K_{AB} u_B = f_A \quad \Leftrightarrow \quad \boxed{[\mathbf{K}]\{\mathbf{u}\} = \{\mathbf{f}\}} \tag{19}$$

2.6. Elementary shape functions

In practice, the calculation of the integrals and the expressions of the interpolating shape functions are defined element-by-element. This is significantly simplified because the algorithm is the same for all elements. Then the resulting element matrices are assembled to obtain the global matrices.

The shape functions are compact, this means that they are zero outside the sub-domain Ω^e of the element (e) to which they correspond. A local node numbering and local coordinate system is defined in each element, and this is later correlated to the global numbering and axes.

When solving 1D problems we can assume linear shape functions of 2 nodes as those in Fig. 2.

Considering the values of N_1 y N_2 in Fig. 2, we obtain the integrals to obtain each component of the element stiffness matrix; for example K_{11}^e is:

$$K_{11}^e = \int_0^h \frac{\mathrm{d}N_1}{\mathrm{d}x} EA \frac{\mathrm{d}N_1}{\mathrm{d}x} \mathrm{d}x = \frac{EA}{h^2} h = \frac{EA}{h} \tag{20}$$

and similarly for the rest of the terms:

$$K_{22}^e = \frac{EA}{h}; \quad K_{12}^{EA} = K_{21}^e = -\frac{EA}{h} \tag{21}$$

resulting

$$[\mathbf{K}^e] = \frac{EA}{h} \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix} \tag{22}$$

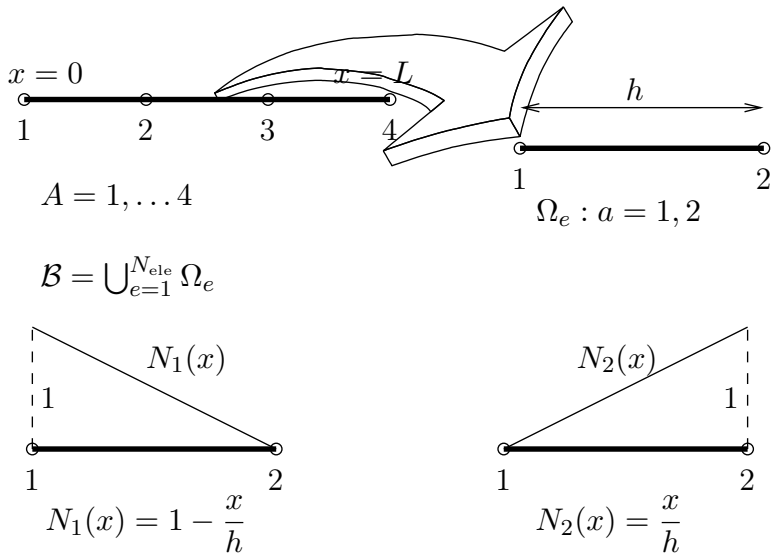


Figura 2: Schematic of the linear shape functions of element 2 in the discretisation of a 1D bar with 3 elements.

Analogously, the volumetric forcing vector is

$$\left\{\mathbf{f}^{\text{vol},e}\right\} = \frac{h}{6}\left\{\begin{array}{c} 2q_1 + q_2 \\ q_1 + 2q_2 \end{array}\right\} \tag{23}$$

After forming the elementary matrix they are expanded to the global system and numbering

$$\begin{aligned} K^e_{ab} &\rightarrow \left[\hat{\mathbf{K}}^e\right] \\ f^e_a &\rightarrow \{\hat{\mathbf{f}}^e\} \end{aligned}$$

and finally these local matrices are assembled in global matrices of the whole structure. In the example of Fig. 2 all the elements are identical and therefore their stiffness matrix is also the same, being their value the one given in Eq. (22).

In order to perform the assembly, the location of each element needs to be considered. If we observe Fig. 2 we see that elements 1 and 2 share node 2 (global numbering), that is why position $(2, 2)$ in the global stiffness matrix will be shared for the local matrices of elements 1 and 2. The shape of the global matrix is:

$$K^{global} = \begin{bmatrix} K^{(1)}_{11} & K^{(1)}_{12} & 0 & 0 \\ K^{(1)}_{21} & K^{(1)}_{22} + K^{(2)}_{11} & K^{(2)}_{12} & 0 \\ 0 & K^{(2)}_{21} & K^{(2)}_{22} + K^{(3)}_{11} & K^{(3)}_{12} \\ 0 & 0 & K^{(3)}_{21} & K^{(3)}_{22} \end{bmatrix} \tag{24}$$

From this assembly we can obtain the global stiffness matrix by consider values in the local stiffness matrices:

$$[\mathbf{K}] = \frac{EA}{h}\begin{pmatrix} 1 & -1 & 0 & 0 \\ -1 & 1+1 & -1 & 0 \\ 0 & -1 & 1+1 & -1 \\ 0 & 0 & -1 & 1 \end{pmatrix}$$

The elementary matrices and global volumetric forcing are:

$$\left\{\mathbf{f}^{\text{vol},e}\right\} = \frac{h}{6}\begin{Bmatrix} 2q_1 + q_2 \\ q_1 + 2q_2 \end{Bmatrix} \Rightarrow \left\{\mathrm{f}^{\text{vol}}\right\} = \frac{h}{6}\begin{Bmatrix} 2q_1^{(1)} + q_2^{(1)} \\ q_1^{(1)} + 2q_2^{(1)} + 2q_1^{(2)} + q_2^{(2)} \\ q_1^{(2)} + 2q_2^{(2)} + 2q_1^{(3)} + q_2^{(3)} \\ q_1^{(3)} + 2q_2^{(3)} \end{Bmatrix}$$

Considering a restriction of the movement at the left end $x = 0$ and a point load q_L applied at the right end $x = L$, the matricial equation that results from the FE method is:

$$[\mathbf{K}]\{\mathbf{u}\} = \left\{\mathbf{f}^{\text{vol}}\right\} + \left\{\mathbf{f}^{\text{ext}}\right\}$$

$$\frac{EA}{h}\left(\begin{array}{c|ccc} 1 & -1 & 0 & 0 \\ \hline -1 & 1+1 & -1 & 0 \\ 0 & -1 & 1+1 & -1 \\ 0 & 0 & -1 & 1 \end{array}\right)\left\{\begin{array}{c} 0 \\ \hline u_2 \\ u_3 \\ u_4 \end{array}\right\} = \frac{h}{6}\left\{\begin{array}{c} 2q_1^{(1)} + q_2^{(1)} \\ \hline q_1^{(1)} + 2q_2^{(1)} + 2q_1^{(2)} + q_2^{(2)} \\ q_1^{(2)} + 2q_2^{(2)} + 2q_1^{(3)} + q_2^{(3)} \\ q_1^{(3)} + 2q_2^{(3)} \end{array}\right\} + \left\{\begin{array}{c} R_0 \\ \hline 0 \\ 0 \\ P \end{array}\right\} \quad (25)$$

The volumetric forcing vector is added to the vector representing the external actions, which in this case it is a load in P. On the other hand, at the left end $x = 0$ there is a reaction given by imposing the displacement in this point, $u(0) = 0$. Such reaction is obtained *afterwards*, once the nodal displacements are obtained $\{\mathbf{u}\}$.

The global stiffness matrix is reduced to remove the degrees of freedom associated with the boundary conditions of the problem. This is followed by the calculation of the nodal displacements solving the algebraic system of equations through the inversion of the reduced global stiffness matrix. After this is done, equilibrium can be checked from the sum of the internal forces in the structure:

$$\left\{\mathbf{f}^{\text{int}}\right\} = [\mathbf{K}]\{\mathbf{u}\}$$

which should be 0.

On the other hand, the reaction in this case could be obtained by multiplying the eliminated row in the global stiffness matrix $[\mathbf{K}]$ (corresponding to the boundary condition of imposed displacement) times the vector of displacements and subtracting the forces applied at the that node (which can only be volumetric forces because the boundary condition of type Dirichlet):

$$R_0 = K_{1j}u_j - f_1^{\text{vol}}$$

Important observation.— You will see that in this structural problem, which is extremely simple, the FE solution in terms of displacements and stresses is exactly the same as the exact analytical value at the nodes of the numerical model. This is called *"superconvergent" problem* and it happens because the only error introduced in the FE model is observed at the intermediate sections between nodes, i.e. within the finite elements, being exact the values at the nodes and at the integration points.

This is not generally the case in other, more complicated, problems in which the solution at the nodes diverges from the exact value at those points.

3. FE simulation with Abaqus

We will analyse the response of a 1D elastic bar with length $L = 10\text{mm}$ and constant section $A = 1\text{mm}^2$. The material of the bar is linear elastic, with a Young Modulus $E = 1000\text{MPa}$. The left end ($x = 0$) is fully fixed and the right end ($x = L$) has an applied point axial load $P = 5\text{N}$, as shown in Fig. 3. It is assumed that deformations are small.

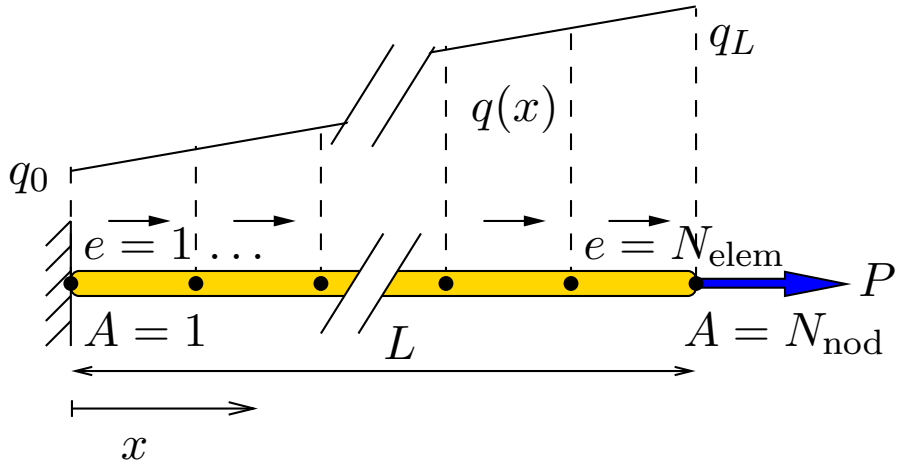


Figura 3: Elastic 1D bar model

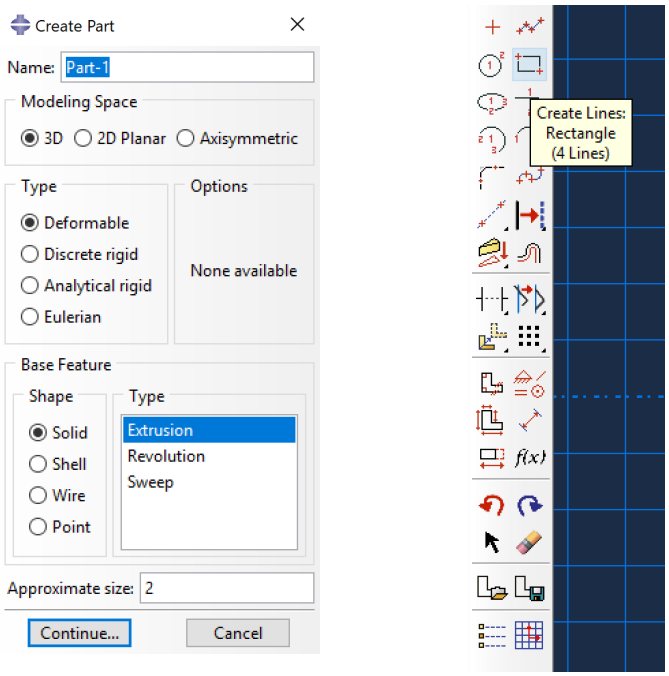
Using the hundreds (c), tens (d) and units (u) of your student number the parameters defining the model will be obtained as:

- Number of elements for the FE discretisation of the bar $N_{\text{elem}} = 11 - c$
- The distributed load $q(x)$, per unit length in the along-bar direction will vary linearly from the two ends of the bar $x = 0$ y $x = L$, with the following values at both ends $q_0 = d/10$ N/mm, $\quad q_L = q_0 + u/10$ N/mm, respectively.

The FE code of the problem should follow these steps:

1. Propose the elastic problem and boundary conditions.
2. Calculate the numerical expressions of the global stiffness matrix and forcing vectors.
3. Application of boundary conditions and solution of the algebraic system of equations.
4. Solution of the displacements $u(x)$ and stresses $\sigma(x)$ at each point. Plot the graphs in terms of the along-bar coordinate x and compare the results with the analytical solution. (Note: the stresses are obtained from the nodal displacements as $\sigma = E\varepsilon = E\mathrm{d}u/\mathrm{d}x$, and using the internal displacement interpolation of each finite element, $\sigma^h = Eu_a \mathrm{d}N_a/\mathrm{d}x = E\left(u_2 - u_1\right)/$

We will start using the commercial FE software `Abaqus` to obtain the results of the proposed problem. Even though it is a 1D problem we will model the bar with 3D elements to help visualising the structure following similar steps to those in the first practical session, but now doing our first 3D model.



(a) Part/solid (b) Rectangle

Figura 4: Definition of the geometry.

3.1. The Part module

First we will create a new model in *Abaqus CAE*. Go to the part module, click in the icon to create a new part and select 3D deformable, extrusion solid (figure 4a). We will create a rectangle (figur 4b) from the corner points with coordinates (-0.5,-0.5) and (0.5,0.5) in mm. Therefore the area will be 1 mm^2 as intended.

After this is done we click in *done* (or press the wheel of your mouse if you have it) and define the length of the bar in direction Z, which in our case is 10 mm. You should see a body similar to that in Fig. 6.

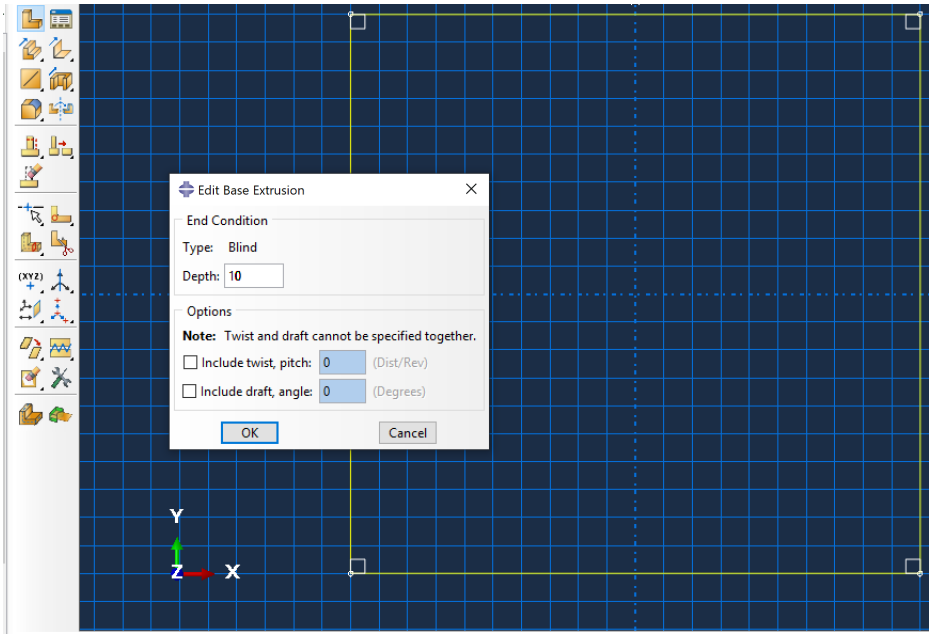


Figura 5: Extrussion of the section.

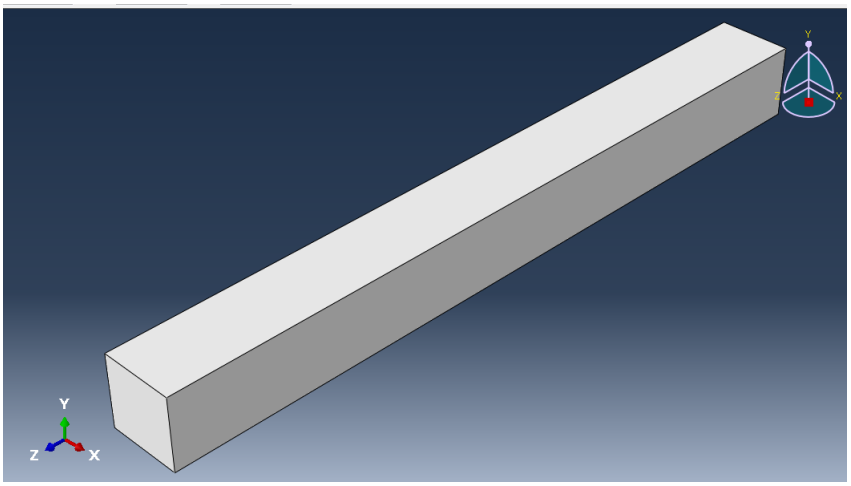


Figura 6: Final geometry of the bar.

3.2. The **Property** module

Create a new material (figure 7) and select elastic linear material with the properties $E = 1000\,\text{MPa}, \nu = 0$. Remember the dimensional relationships defined in the first practical session; we defined the geometry in mm and therefore we need to use MPa=N/mm^2 for the stress and the Young's modulus.

Create a *"section"* of type *solid*-homogeneous (figure 8a) and chose the material previously created.

Assign the newly created section to the bar (figura 8b) by selecting the whole volume. Confirm by clicking the icon *Done*.

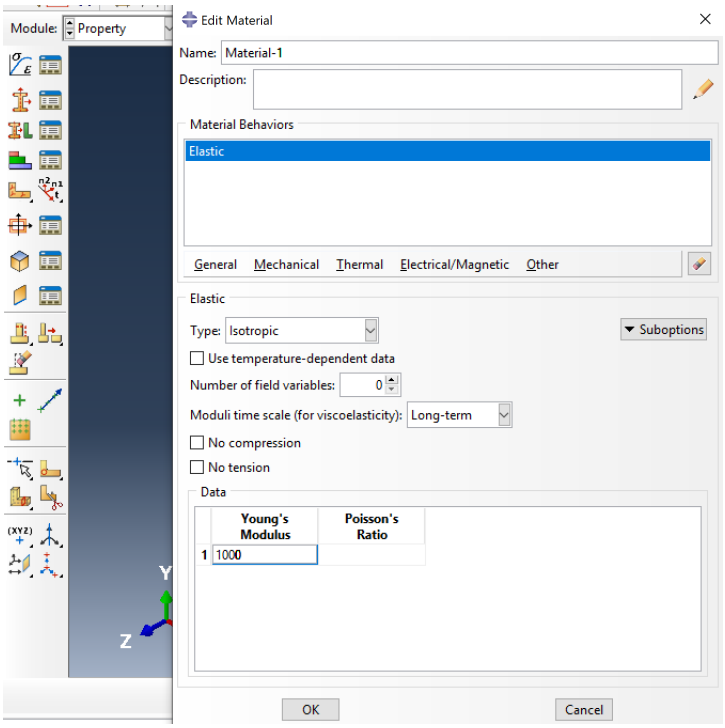
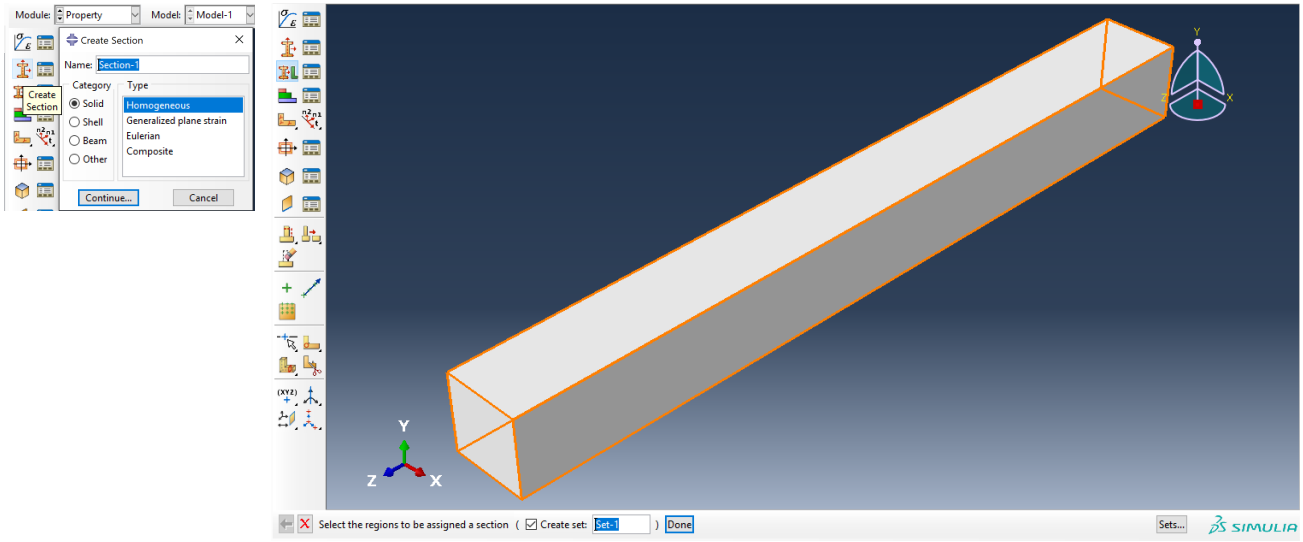


Figura 7: Elastic material.



(a) Create section (b) Assign section

Figura 8: How to select the section

3.3. The **Assembly** module

In this module you only need to create an "Instance" from the part previously defined. This can be done by means of the default options shown in Fig. 9.

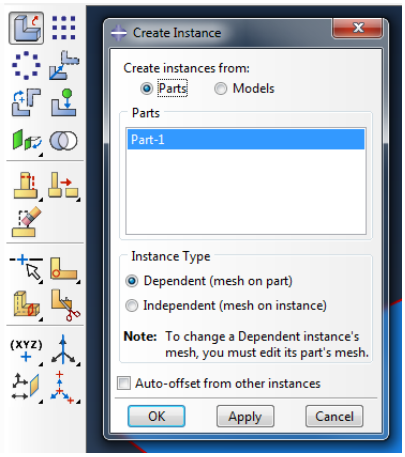


Figura 9: Assembly: create an instance from the part.

3.4. The **Step** module

Create an analysis "step" that is "Static, general" using the default options (figure 10). It is not necessary to edit the requested "field output" because it includes the default response variables that we will need in our study.

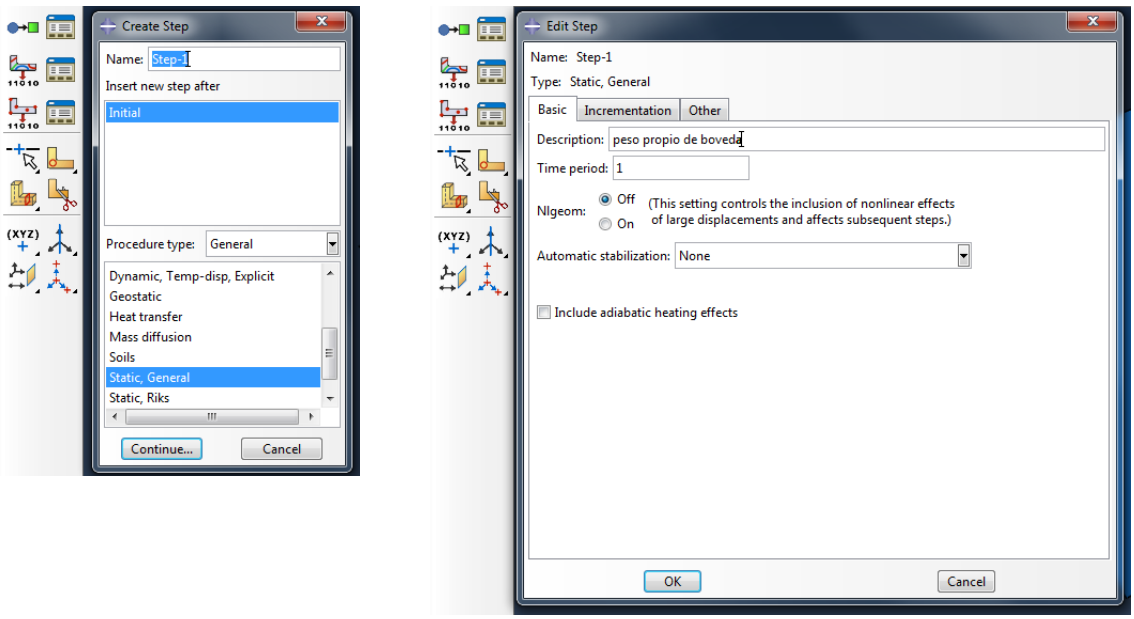


Figura 10: Create and define the analysis "step".

3.5. The Load module

We must define in this module the two types of boundary conditions. First, at the left end *Z=0* we clamp the bar. To this end we create a *Boundary Condition*that is *Mechanical* (category) and the type is *Encastre* (figure 11).

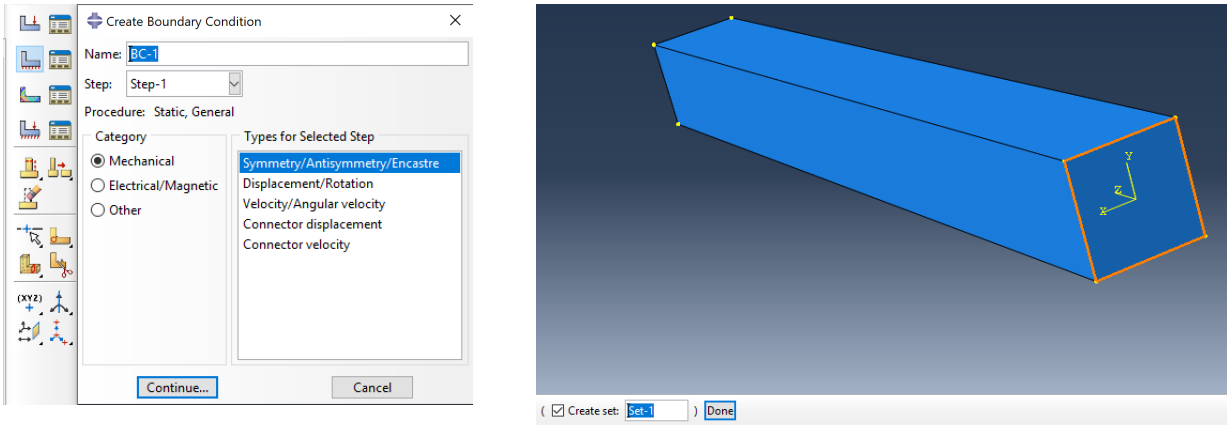


Figura 11: Creation of the encastre conditions

Once the surface to clamp (encastre) is defined and then confirmed clicking *Done*, we must select the type of condition to assign to this region of the bar. To this end we select *Encastre* from the possible options (figure 12.

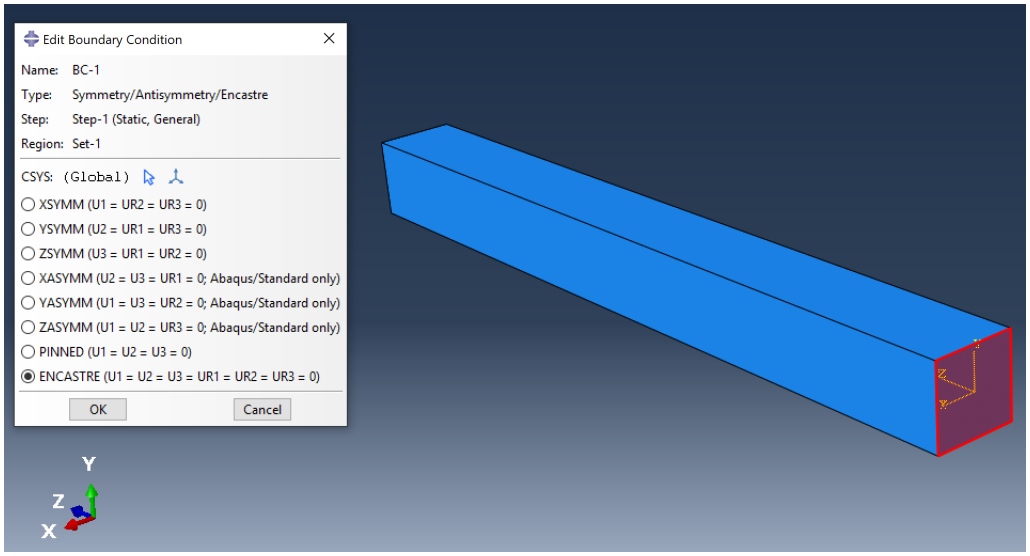
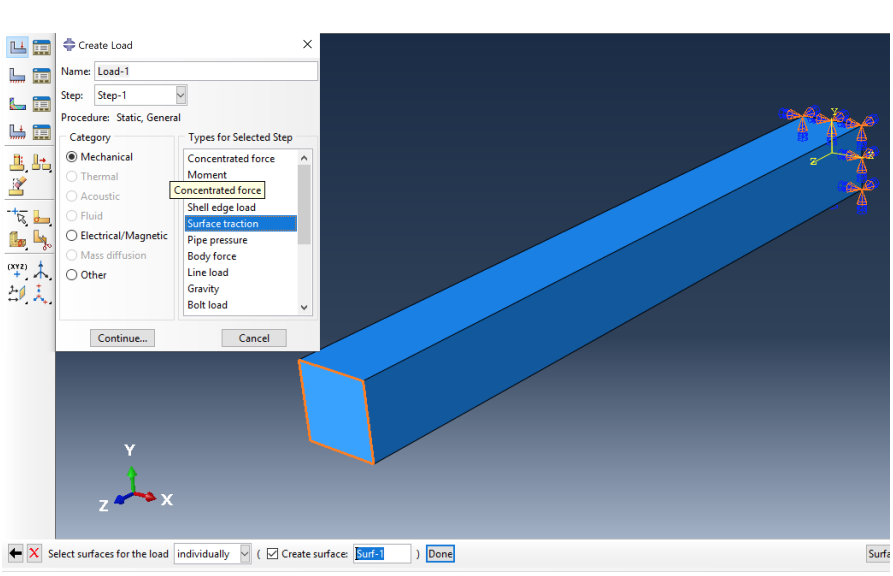
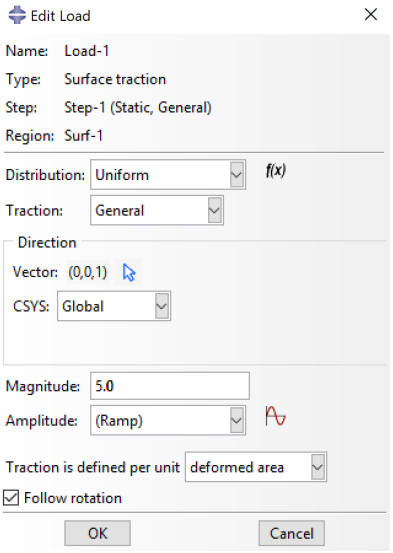


Figura 12: Defition of *Boundary Condition: Encastre.*

Now we define the two loads in our our problem: a point load and a volumetric load. With respect to the point load it can be defined in different ways. The easiest one to apply a traction at the surface *Z=L*, which is a load of category *Mechanical* and type *Surface Traction* in Abaqus (figure 13a). Next, we select the surface where the traction is going to be applied and the window with the options to define the load will be displayed (figura 13b). The load is uniformly distributed, and the traction is of type *General* (i.e. it is not a shear force but a normal action). In order to define the direction of the load we need to define the normal to the surface, which in this case is a vector parallel to the positive Z axis, therefore we can define it as a unit vector (0,0,1). The magnitude of the traction will be $5N$ (force) divided over the surface area where it is applied, 1 mm^2, thus the applied pressure is, 5 MPa.



(a) Definition of the load surface



(b) Definition of the load direction and magnitude

Figura 13: Definition of the load at the end of the bar.

With respect to the volumetric force, it has to be defined in the entire volume of the bar. First you need to select *Body Force* as the forcing type when creating the load. Afterwards, this load has to be assigned to the entire structure (figure 14). Then we need to define the magnitude of the load. To this end we click in the icon with a symbol $f(x)$ because it will not be uniform but a function of the coordinates, in this case Z. This expression is defined in the window *Expression Field* the pops out when hitting $f(x)$. Considering that the student's registration number (número de matrícula) for the development of this exercise is 1024, the expression is $q = 0.2 + 0.04 * Z$. This is what we need to introduce in the corresponding box. Once the expression is created, in the field *Distribution* of the window *Edit Load* we assign the newly generated *Expression Field* which in our case is *AnalyticalField-1*. The component will be (0,0,0), that is, direction Z.

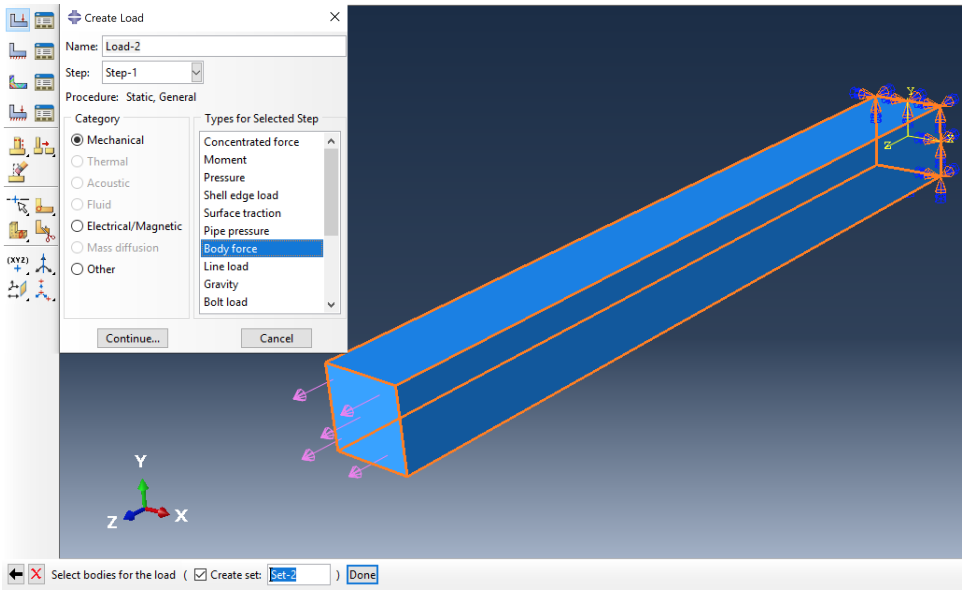


Figura 14: Creation of the *Body Force*

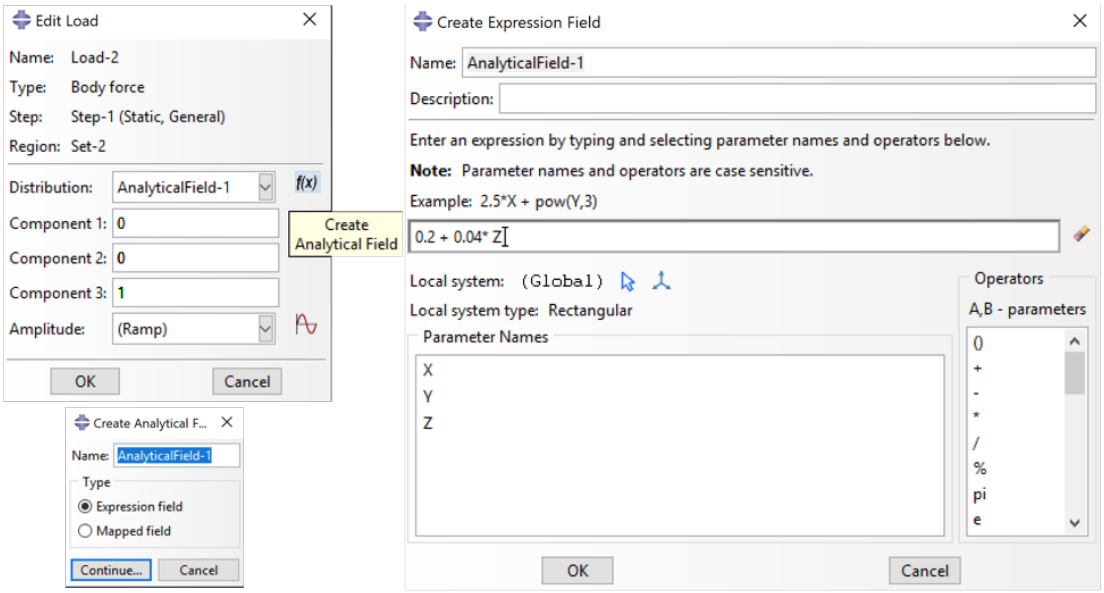


Figura 15: Definition of the load distribution with an *Expression Field*

Once the second load is defined the model should look like the one in Fig. 16.

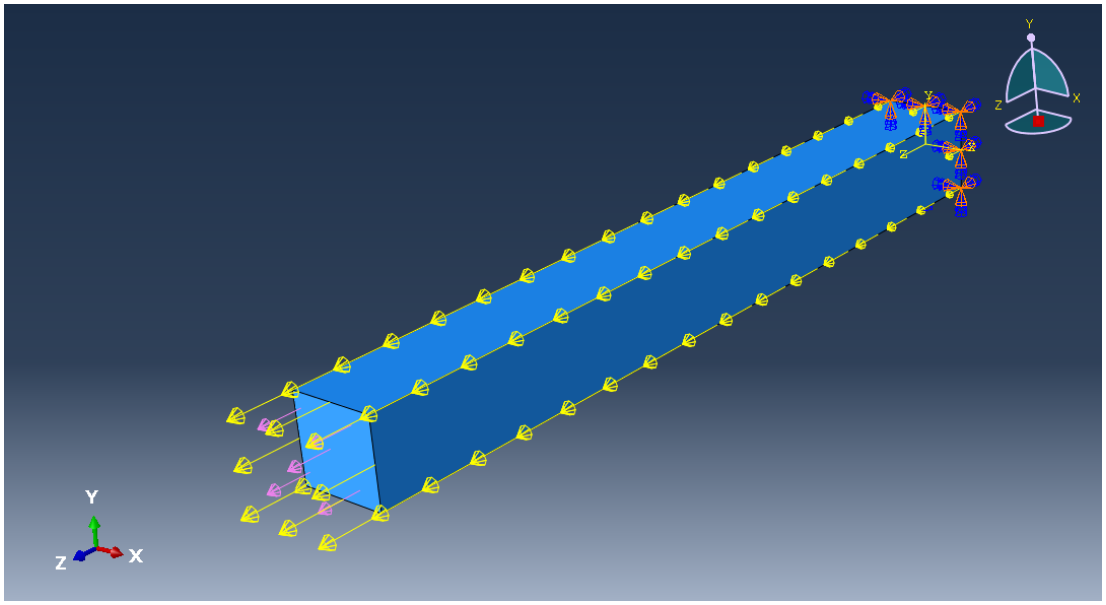


Figura 16: Complete set of loads

3.6. The Mesh module

In the Mesh module we define the finite element discretisation. We start by selecting the Part, expanding the model tree on the left side of the screen, and activating with a right click of your mouse in the icon *"Mesh"*: Fig. 17.

This will open the *Mesh* menu. Go to *Controls* and select "Hex" and "Structured" to generate an structured mesh of regular hexahedra (Fig. 18).

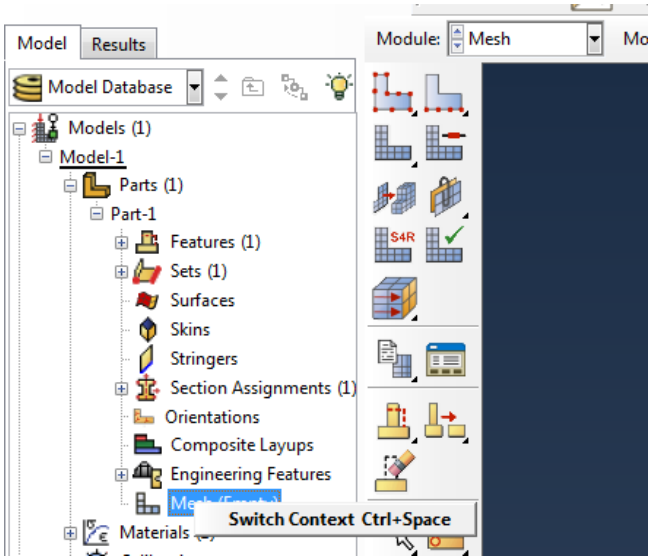


Figura 17: Select the menu to mesh.

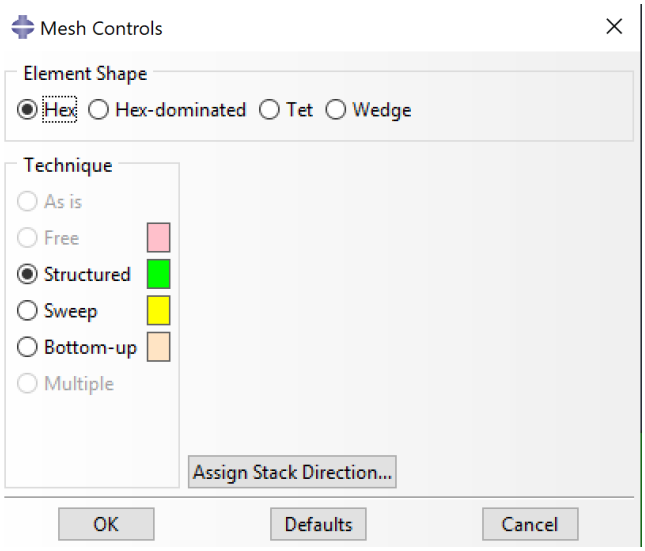


Figura 18: Define options in "Mesh controls"

The following step is to open the menu *Mesh* → *Element type* and select the options "3D Stress", "Linear", *"Reduced integration"*. This will select the element type C3D8R in *Abaqus*, which is sufficient for the analysis that we will perform. (Fig. 19).

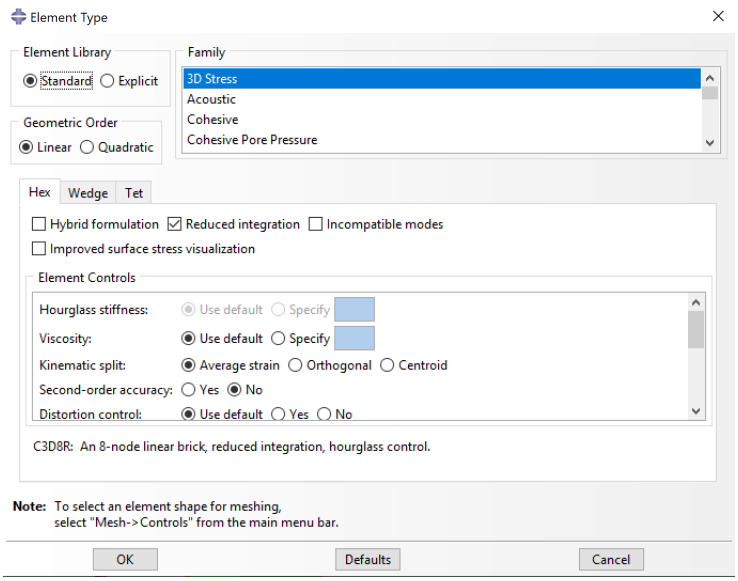


Figura 19: Select the element type in "Mesh/element"

You need to select the *"Seed"* of the mesh, which is the approximate size of the elements in your model. We chose 1 mm as the size of elements in all the space direction, which will lead to 10 hexahedral elements in the longitudinal direction of the bar (Fig. 20). This will result in the mesh shown in Fig. 20.

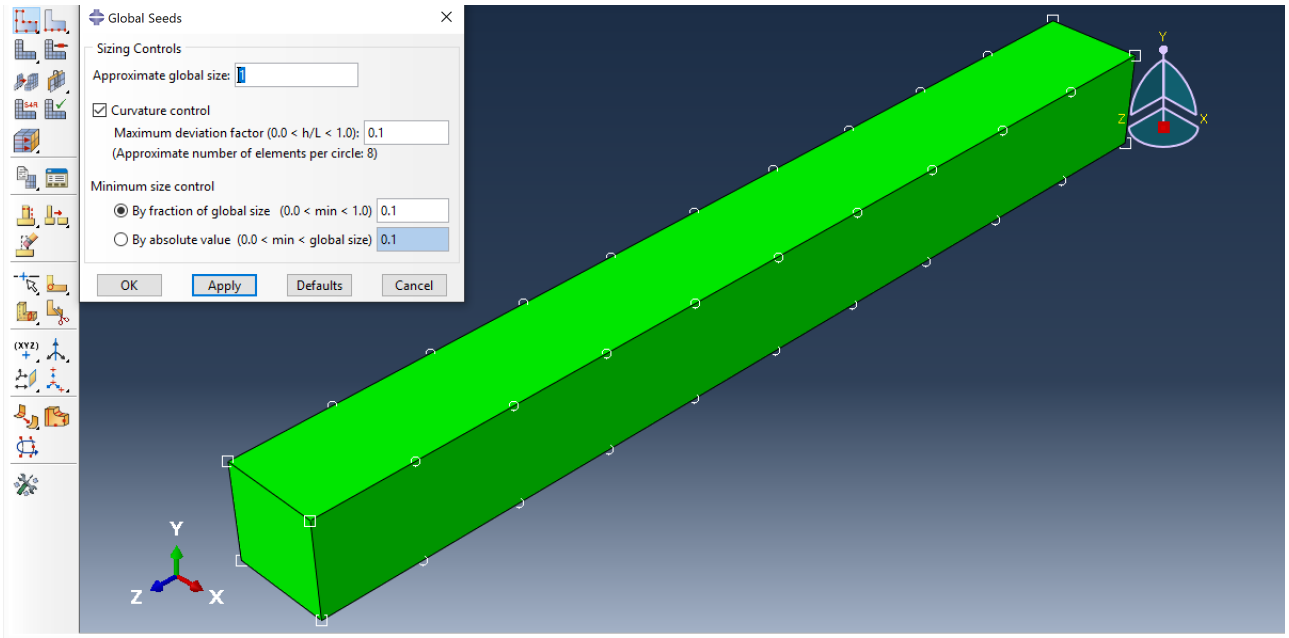


Figura 20: “Global Seeds”

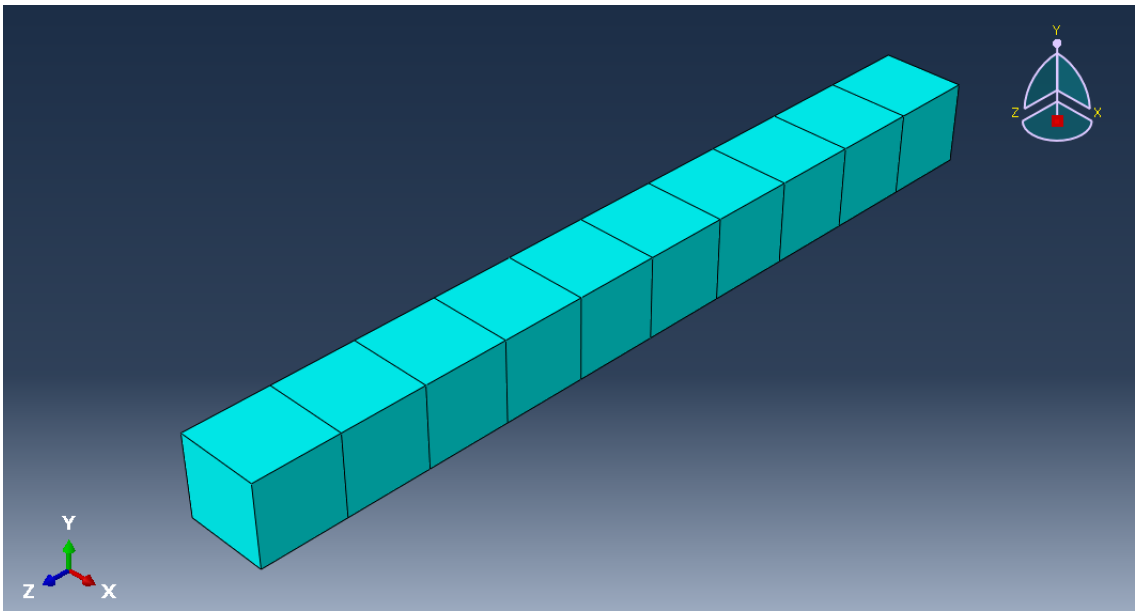


Figura 21: Proposed mesh.

3.7. The Job module

Now that the model is finished, a "Job" with the default analysis options is created (figure 22).

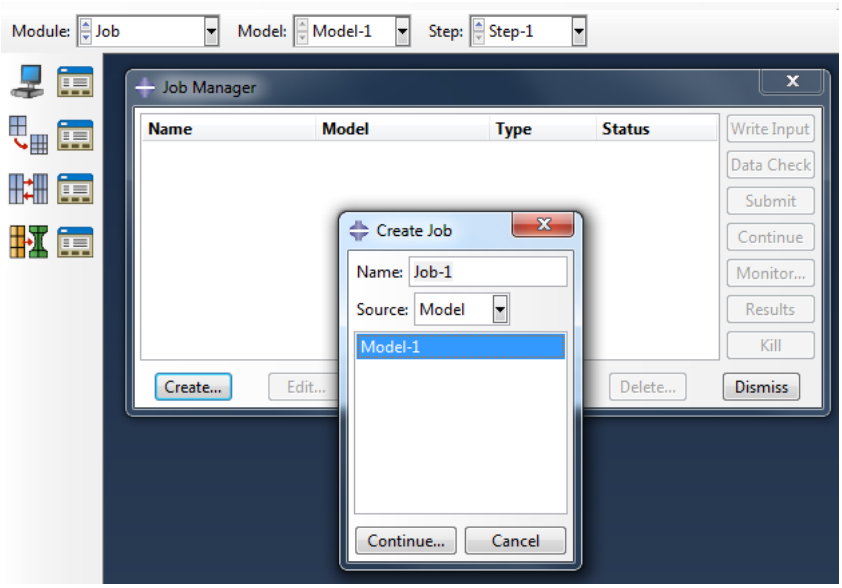


Figura 22: "Job" definition

This Job is submitted for analysis by clicking "Submit". The *Status* changes from "Submitted" → "Running" and then → "Completed" if there is no error. (figure 23). It is always advisable to check for warnings in the "Monitor".

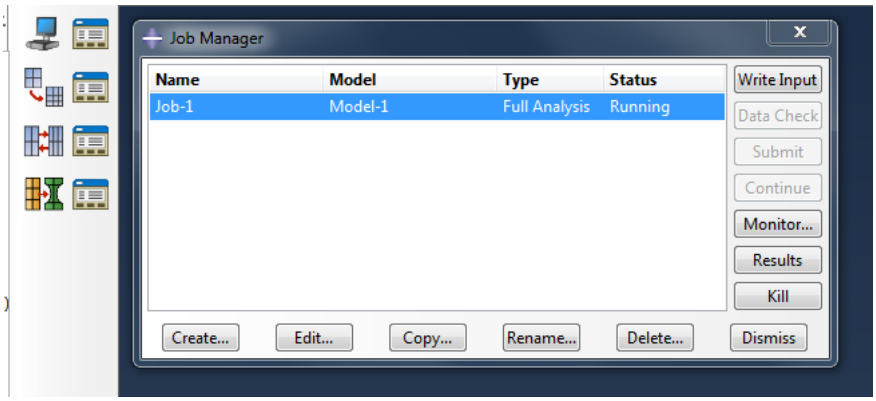


Figura 23: "Job" submission

If there is no error message you can visualise the results.

3.8. The **Visualization** module

In order to check, the first field to visualise is the displacement in the along-bar direction Z. The peak displacement is 0.073 mm (figure 24). We will plot the Z displacement with respect of the coordinate Z along the bar.

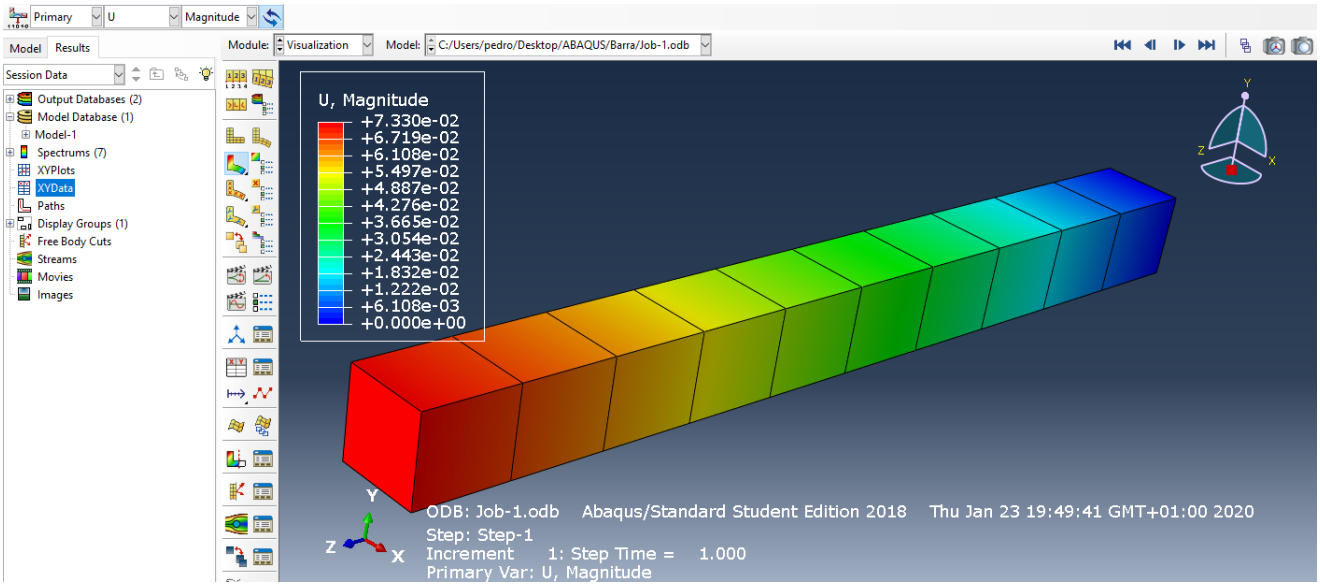


Figura 24: Displacement field.

The first thing we need to do is to indicate the *Path* (recall the first lab session) with two nodes distributed in Z. We select 'path' and with a right-click of the mouse we open a menu where we select *create* (figure 25). The type of path will be based on a selection of the nodes. Our mode of selection will be advancing in the positive-Z direction, therefore we chose *Add After* and make sure that we select a node in the face $Z = 0$ and the next one in $Z = L$. We can chose any edge because all of them have the same response. This defines the path as shown in figure 26.

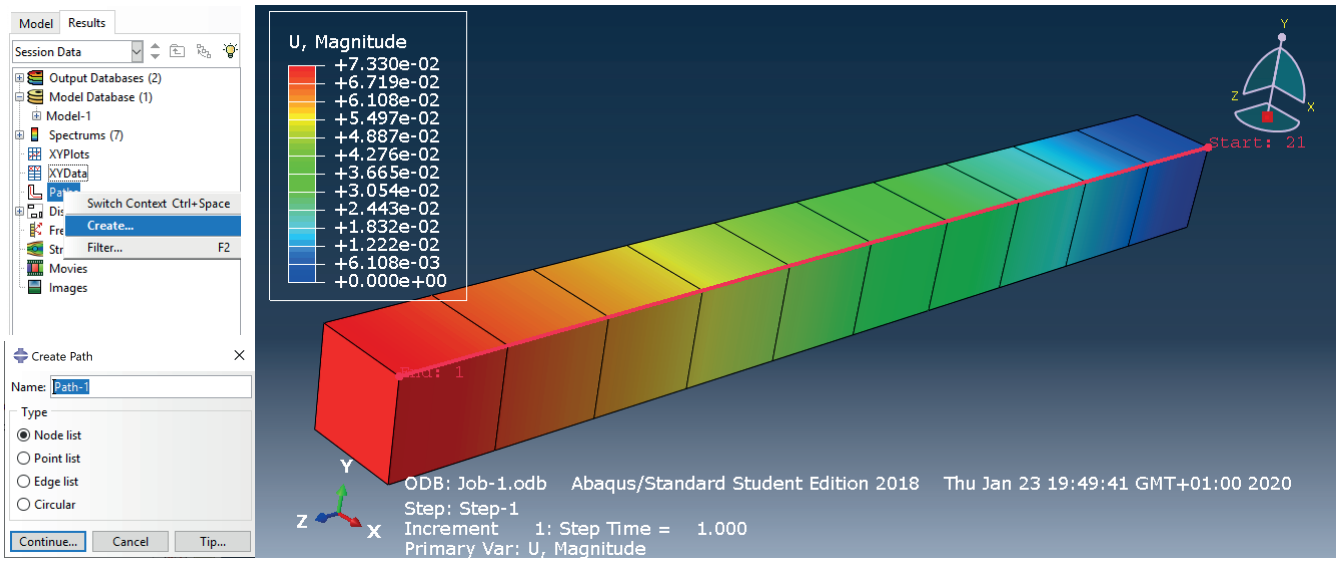


Figura 25: Path definition.

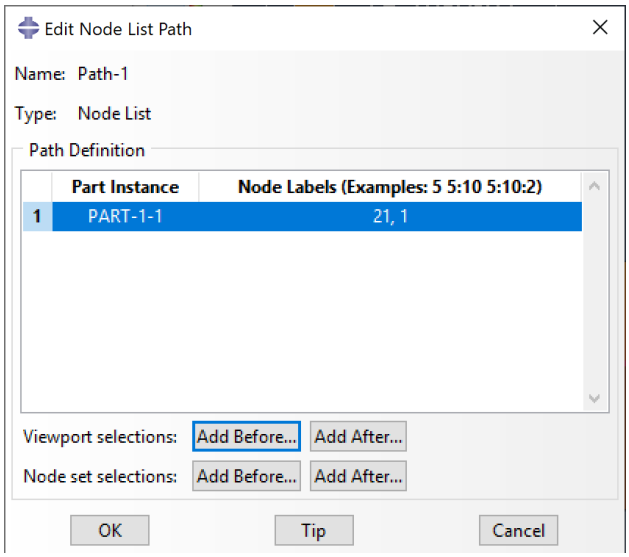


Figura 26: Node list in the path.

Now we can create a graph with the along-bar displacement (U_Z) in the structure. We need to go to *XY-Data* and in this field, with a right-click of the mouse a menu opens. There we select *Create* and select a displacement field $U3$, which corresponds to the along-bar displacement (U_Z). In the window *XY Data from Path* (figure 27) we chose the Path defined previously (in the undeformed configuration) and select the option to include intersections because this will add the intermediate nodes along the start and end nodes of the path. Finally, we select the horizontal axis in our plot as the coordinate Z.

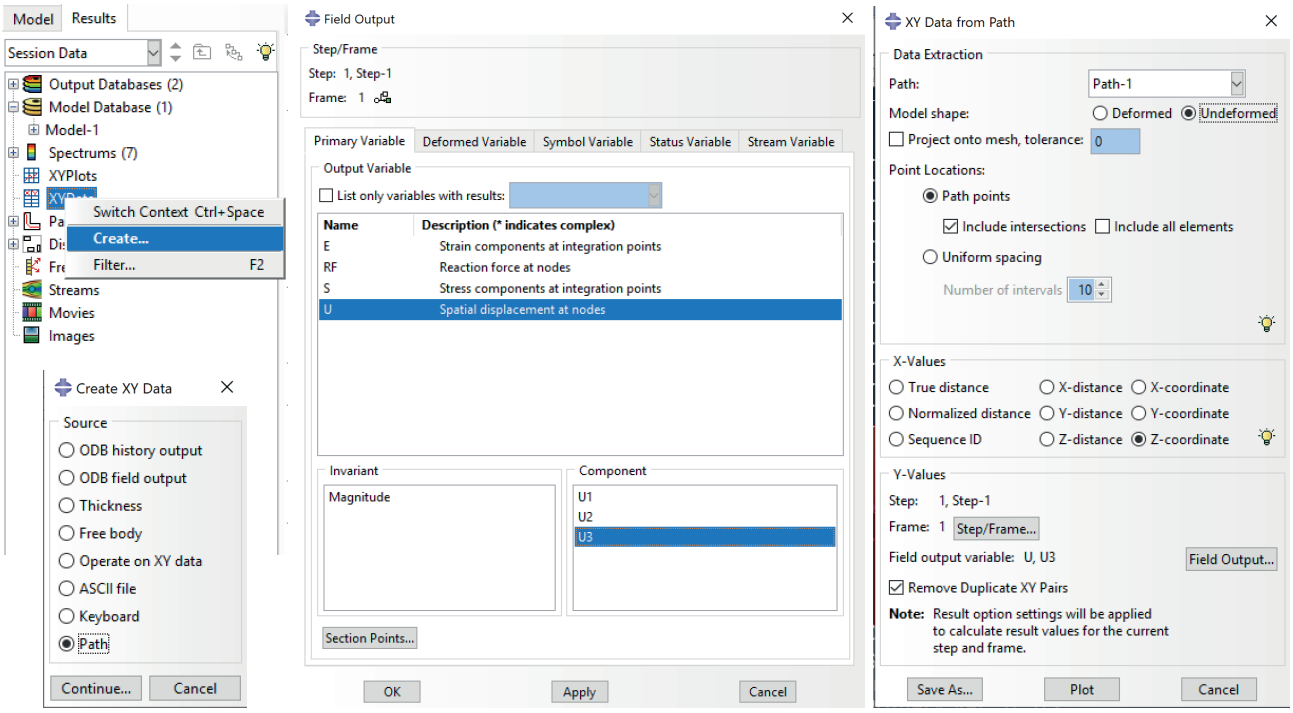


Figura 27: Creation of the XY-data

Plotting this graph we should obtain figure 28. We will compare this with the results obtained in the `Python` FE solver script.

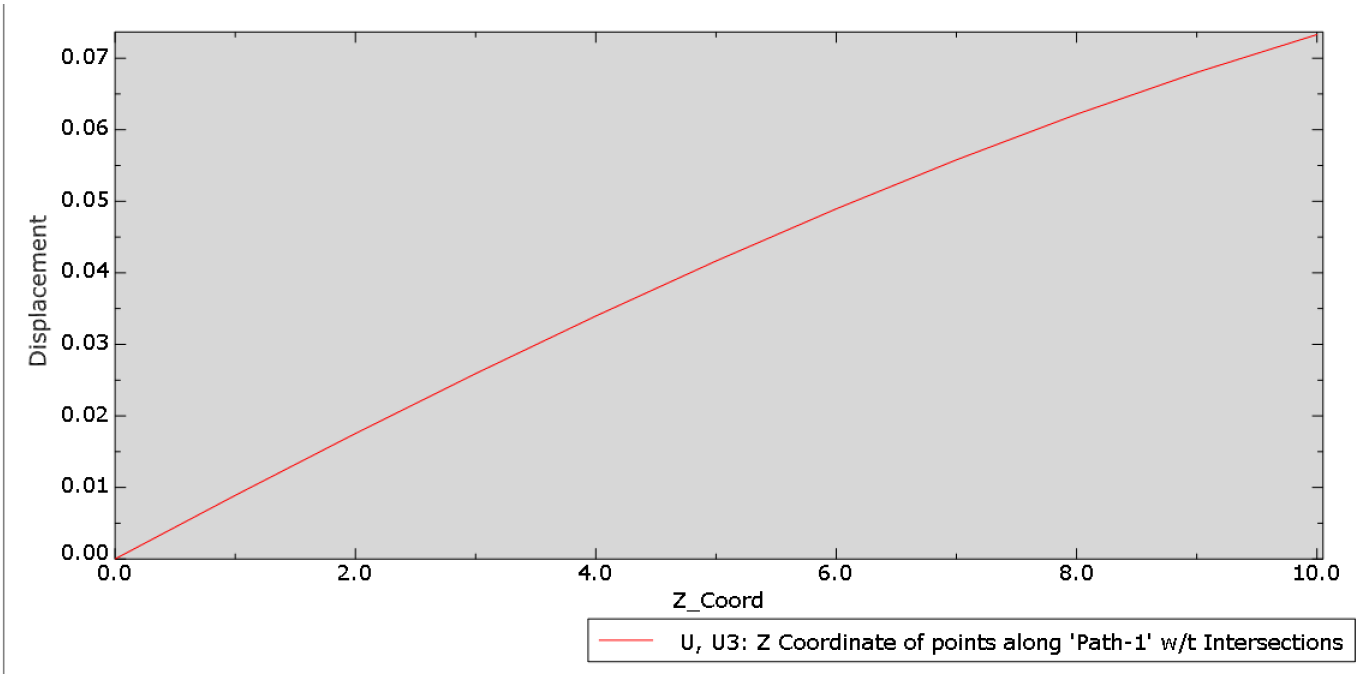


Figura 28: Along-bar displacement obtained in Abaqus.

We will save this result to compare it later with that obtained in `Python` and with the analytical solution given by the strong formulation of the problem. This is done by clicking the icon *Save as...* We will call the file `u3.txt`.

If we change the response field to stresses and plot S33, which corresponds to the normal stress σ_z we will obtain a result similar to figure 29. You will observe that that the trend is to be expected but at the ends some strange things happen. This is because the stress is calculated at the integration points and not at the nodes, where the path is defined. Therefore the stress needs to extracted in a different way.

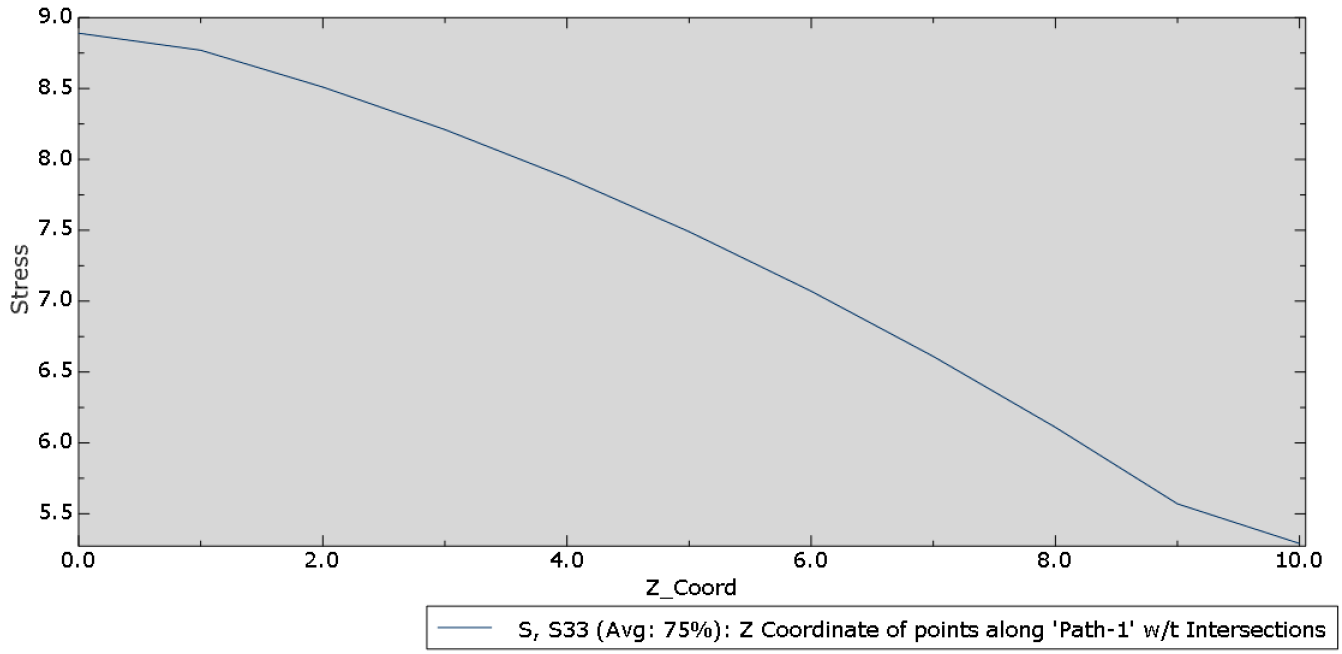
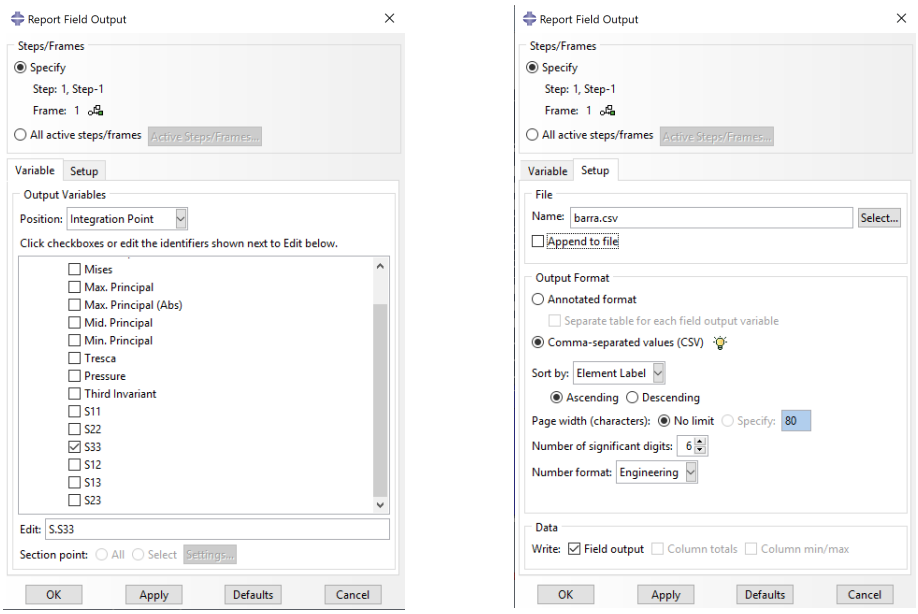


Figura 29: Stress σ_z along the axis Z.

To this end we go to tab "Report/" and there we select the suboption *Field Output*. This will



(a) Variable (b) Setup

Figura 30: Field Output

open a wind with a tab called *Variable* where we select the field output S33 at the integration points (figure 30a), while in the tab "Setup" we will specify the file name as `s33.csv` (figure 30b).

4. Example of finite element code in Python

In the following link we provide a *Notebook* of *Python* with a FE code applied to the case of a 1D elastic bar.

The file `*.ipynb` can be downloaded from the repository below,

`Repositorio Git-Hub`

or alternatively it can be open directly in any of the programs that will be detailed now. The *Notebook* can be run with the program `Jupyter`, which belongs to the suite `Anaconda`, open for plataforms *Windows, Mac y Linux*. It is necessary to download this suite and load the *Notebook*, either opening the file with extension `*.ipynb`, or from the repository in Git-Hub mentioned previously.

Another option is to run the script *in the cloud*, without the need to install anything. Some of the most widely used are:

- `Colab de Google` It is necessary to have a gmail account.
- `MyBinder` Only available from a repository in Git-Hub

4 Ejemplo de programación elementos finitos en Python.

4.1 Introducción.

El objetivo de esta práctica es ver una pequeña introducción a como preparar un programa de cálculo de elementos finitos. Para ello partiremos de un ejemplo de un modelo de barras en dos dimensiones (celosías). A partir del ejemplo se irán desarrollando paso a paso las distintas funciones necesarias para su solución. La programación se realizará en python utilizando las siguientes librerías:

1. math
2. numpy
3. scipy
4. matplotlib

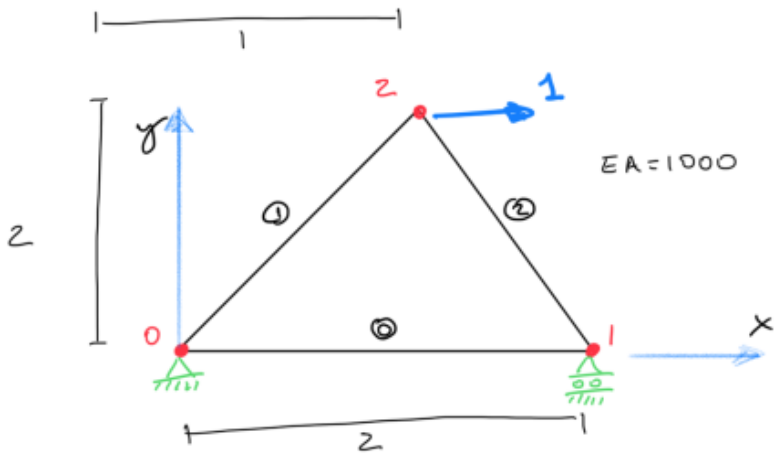
El código se implementará en un notebook de jupyter que permite la mezcla de texto, gráficos y código de forma sencilla mediante el uso de celdas de tipo *markdown* o tipo *code*.

Primeramente se cargan las librerías o módulos de python que se usarán en el cálculo.

[1]:
```
import numpy as np
import scipy.sparse as sp
from scipy.sparse.linalg import spsolve
import matplotlib.pyplot as plt
import math
```

4.2 Ejemplo

Vamos a utilizar el siguiente ejemplo como base para desarrollar las funciones que nos permitirán resolver el problema, pero que serán funciones genéricas aptas para otros problemas:



Como en el lenguaje python los arrays, listas, etc. se indexan empezando por cero, uilizaremos ese convenio en la descripción del modelo. Los nodos se numerarán de forma correlativa empezando

por cero, con los elementos se hará de forma similar y para establecer las fuerzas concentradas en nodos se usará el nodo, la dirección (0 para el eje x y 1 para el eje y) ysu valor. Las condiciones de contorno se definirán de forma similar a las fuerzas pero sin el dato valor. Si resumimos la información del modelo relevante para el cálculo observamos lo siguiente:

- Coordenadas de los nodos.

Nodo	x	y
0	0	0
1	2	0
2	1	2

- Elementos (barras articuladas)

Elemento	nodo 1	nodo 2	EA
0	0	1	1000
1	0	2	1000
2	1	2	1000

- fuerzas aplicadas

Nodo	dirección	valor
2	0	1

- Condiciones de contorno

Nodo	dirección
0	0
0	1
1	1

Lo primero que se hará es definir formalmente el modelo en el lenguaje en cuestión. Para el caso que nos ocupa usaremos un array del tipo numpy para las coordenadas y listas para los elementos, las fuerzas y las condiciones de contorno. Crearemos las variables correspondientes con los nombres que nos parezcan oportunos.

Para los nodos un array con sus coordenadas `[[x0,y0],[x1,y1],...]`
Para los elementos una lista con los mismos `[el1,el2,....]`, donde cada elemento es `[nodo 1, nodo 2, EA]`
Para las fuerzas una lista con cada una de las componentes `[[nodo,dirección,F_val],...]`
Y para las condiciones de contorno una lista similar `[[nodo,dirección],...]`
Escribiéndolo en python tendríamos lo siguiente

[2]:
```
x = np.array([[0,0],[2,0],[1,2]])
elementos = [[0,1,1000],[0,2,1000],[1,2,1000]]
```

```
fuerzas = [[2,0,1]]
cc = [[0,0],[0,1],[1,1]]
```

4.3 Matriz de rigidez del modelo

El objetivo del cálculo es llegar a una ecuación del tipo $\mathbf{Kd} = \mathbf{f}$ donde $\mathbf{K}$ es la matriz de rigidez global del modelo que se obtiene a partir de las matrices de rigidez de cada elemento ensambladas (sumadas en una cierta posición dependiente de los nodos) en la global.

El vector de fuerzas $\mathbf{f}$ también se obtiene de forma similar ensamblando las fuerzas dadas cada una de ellas en la posición correspondiente.

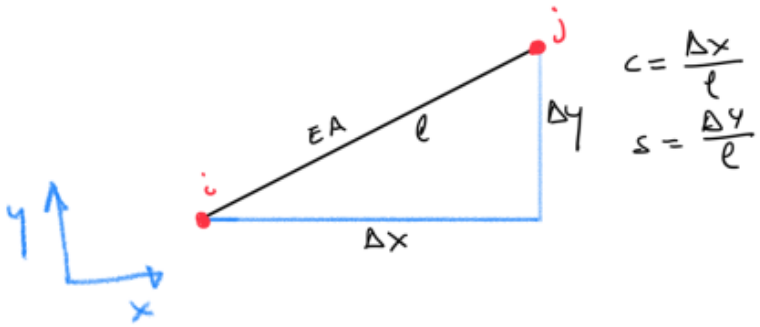
Como los grados de libertad considerados son dos por nodo (u según x,v según y) las dimensiones de las matrices serán, siendo n el número de nodos.

Matriz	dimensión
$\mathbf{K}$	$2n \times 2n$
$\mathbf{d}$	$2n$
$\mathbf{f}$	$2n$

Al haber indexado a partir de cero la relación entre nodo, dirección y grado de libertad es muy sencilla: Al nodo i, dirección j le corresonde el grado de libertad (índice en las matrices) $2i + j$

4.4 Matriz de rigidez elemental

Primeramente obtendremos la función matriz de rigidez de un elemento. Una vez obtenida se aplicará esa función a todos los elementos y se ensamblarán.



La matriz de rigidez del elemento es:

$\mathbf{K}\_\{el\} = \frac{EA}{l}\begin{pmatrix} c^2 & cs & -c^2 & -cs \\ cs & s^2 & -cs & -s^2 \\ -c^2 & -cs & c^2 & cs \\ -cs & -s^2 & cs & s^2 \end{pmatrix}$ siendo c el coseno del ángulo que forma la barra con la horizontal y s el seno.

Como se puede observar, en este caso, está formada por un bloque de 2×2 que se repite cuatro veces, dos de ellas con el signo cambiado.

Si denominamos al bloque básico $\mathbf{K}_{11}$ tendríamos lo siguiente $\mathbf{K_{11}} = \frac{EA}{l}\begin{pmatrix} c^2 & cs \\ cs & s^2 \end{pmatrix}$

y por tanto: $\mathbf{K_{el}} = \begin{pmatrix} \mathbf{K}_{11} & -\mathbf{K}_{11} \\ -\mathbf{K}_{11} & \mathbf{K}_{11} \end{pmatrix}$

Vamos a proceder paso a paso escribiendo funciones para obtener la matriz de rigidez elemental.
Partimos de un elemento que está definido por: `[nodo 1,nodo 2, EA]`.
Además necesitaremos las coordenadas de los nodos (en concreto las diferencias) para obtener la longitud y los senos y los cosenos.

La primera función que vamos incluir es la que accede a las coordenadas de los nodos.
Argumentos: 1. elemento,coordenadas
`[nodo 1,nodo 2, EA]`, coordenadas
Resultado: 1. elemento modificado
`[nodo 1, nodo 2, EA,x2-x1,y2-y1]`

[3]:
```
fk_0 = lambda el,coor:␣
↪[el[0],el[1],el[2],coor[el[1]][0]-coor[el[0]][0],coor[el[1]][1]-coor[el[0]][1]]
```

Para aplicar la función al elemento número 1 (elementos[1]) se haría del siguiente modo:

[4]:
```
fk_0(elementos[1],x)
```

[4]:
```
[0, 2, 1000, 1, 2]
```

La siguiente función calcula la longitud del elemento
Argumentos:
1. elemento
`[nodo 1,nodo 2, EA,x2-x1,y2-y1]`
Resultado:
1. elemento modificado
`[nodo 1, nodo 2, EA,l,x2-x1,y2-y1]`

[5]:
```
fk_1 = lambda el: [el[0],el[1],el[2],math.hypot(el[3],el[4]),el[3],el[4]]
```

Vemos como se aplicaría al elemento cero. Primeramente se aplicaría `fk_0`y posteriormente `fk_1`.

[6]:
```
fk_1(fk_0(elementos[0],x))
```

[6]:
```
[0, 1, 1000, 2.0, 2, 0]
```

Una vez que tenemos la longitud y las diferencias de coordenadas se puede obtener el coseno, el seno y la rigidez axial $\frac{EA}{l}$ del elemento donde tanto el coseno como el seno se obtienen a partir de las diferencias de coordenadas y de la longitud.
Se hace mediante una nueva función `fk_2`

[7]:
```
fk_2 = lambda el: [el[0],el[1],el[2]/el[3],el[4]/el[3],el[5]/el[3]]
```

Obsérvese que cada función parte de los resultados de la función anterior.

[8]:
```
fk_2(fk_1(fk_0(elementos[0],x)))
```

[8]:
```
[0, 1, 500.0, 1.0, 0.0]
```

Ahora se puede obtener el bloque elemental $\mathbf{K_{11}}$ de la matriz de rigidez elemental
$\mathbf{K_{11}} = \frac{EA}{l}\begin{pmatrix} c^2 & cs \\ cs & s^2 \end{pmatrix}$

[9]:
```
fk_11 = lambda el: el[2]*np.
↪array([[el[3]**2,el[3]*el[4]],[el[3]*el[4],el[4]**2]])
```

El bloque elemental es un array del tipo numpy.

[10]:
```
fk_11(fk_2(fk_1(fk_0(elementos[0],x))))
```

[10]:
```
array([[500.,   0.],
       [  0.,   0.]])
```

Y a partir del bloque $\mathbf{K_{11}}$ podemos obtener la matriz de rigidez local del elemento
$\mathbf{K_{el}} = \begin{pmatrix} \mathbf{K_{11}} & -\mathbf{K_{11}} \\ -\mathbf{K_{11}} & \mathbf{K_{11}} \end{pmatrix}$

[11]:
```
fk_loc = lambda k11: np.vstack((np.hstack((k11,-k11)),np.hstack((-k11,k11))))
```

[12]:
```
fk_loc(fk_11(fk_2(fk_1(fk_0(elementos[0],x)))))
```

[12]:
```
array([[ 500.,    0., -500.,   -0.],
       [   0.,    0.,   -0.,   -0.],
       [-500.,   -0.,  500.,    0.],
       [  -0.,   -0.,    0.,    0.]])
```

Todas las funciones auxiliares para obtener la matriz de rigidez las podemos juntar en una sola que a partir de los datos de un elemento y del array de coordenadas del modelo devuelve la matriz de rigidez del elemento.

[13]:
```
fk_el = lambda elemento,coor: fk_loc(fk_11(fk_2(fk_1(fk_0(elemento,coor)))))
```

[14]:
```
fk_el(elementos[2],x)
```

```
[14]: array([[  89.4427191, -178.8854382,  -89.4427191,  178.8854382],
             [-178.8854382,  357.7708764,  178.8854382, -357.7708764],
             [ -89.4427191,  178.8854382,   89.4427191, -178.8854382],
             [ 178.8854382, -357.7708764, -178.8854382,  357.7708764]])
```

4.5 Matriz de rigidez de un conjunto de elementos y ensamblaje

Una vez que tengo una función para obtener la matriz de rigidez de un elemento, la forma de obtener las matrices de rigidez de todos los elementos es aplicar esa función a todos los elementos.

```
[15]: fk_els = lambda elementos,coor: list(map(lambda elemento:␣
       ↪fk_el(elemento,coor),elementos))
```

```
[16]: fk_els(elementos,x)
```

```
[16]: [array([[ 500.,    0., -500.,   -0.],
              [   0.,    0.,   -0.,   -0.],
              [-500.,   -0.,  500.,    0.],
              [  -0.,   -0.,    0.,    0.]]),
       array([[  89.4427191,  178.8854382,  -89.4427191, -178.8854382],
              [ 178.8854382,  357.7708764, -178.8854382, -357.7708764],
              [ -89.4427191, -178.8854382,   89.4427191,  178.8854382],
              [-178.8854382, -357.7708764,  178.8854382,  357.7708764]]),
       array([[  89.4427191, -178.8854382,  -89.4427191,  178.8854382],
              [-178.8854382,  357.7708764,  178.8854382, -357.7708764],
              [ -89.4427191,  178.8854382,   89.4427191, -178.8854382],
              [ 178.8854382, -357.7708764, -178.8854382,  357.7708764]])]
```

Con vistas al ensamblaje nos conviene tener todos los elementos de las matrices de rigidez en un array unidimensional. Para ello usaremos la función np.ravel y definimos una nueva función. Esta función simplemeente calcula todas las matrices de rigidez de todos los elementos y las devuelve en un array de una dimensión.

```
[17]: f_kg1 = lambda elementos,coordenadas:np.ravel(fk_els(elementos,coordenadas))
```

```
[18]: f_kg1(elementos,x)
```

```
[18]: array([ 500.       ,    0.       , -500.       ,   -0.       ,
                0.       ,    0.       ,   -0.       ,   -0.       ,
             -500.       ,   -0.       ,  500.       ,    0.       ,
               -0.       ,   -0.       ,    0.       ,    0.       ,
                89.4427191,  178.8854382,  -89.4427191, -178.8854382,
               178.8854382,  357.7708764, -178.8854382, -357.7708764,
               -89.4427191, -178.8854382,   89.4427191,  178.8854382,
              -178.8854382, -357.7708764,  178.8854382,  357.7708764,
                89.4427191, -178.8854382,  -89.4427191,  178.8854382,
              -178.8854382,  357.7708764,  178.8854382, -357.7708764,
```

```
-89.4427191,  178.8854382,   89.4427191, -178.8854382,
178.8854382, -357.7708764, -178.8854382,  357.7708764])
```

Para poder enamblar las matrices de rigidez es necesario saber a que índices de la matriz de rigidez global va cada elemento de una matriz de rigidez elemental. Vamos a crear unas funciones que nos den esos índices en función de los nodos del elemento, una función para las filas de todos los elementos de la matriz y otra función para las columnas.

[19]:
```
columnask = lambda y:  np.array(list(( map(lambda x: [x*2,x*2+1],y)))*4).
 ↪flatten()
filask = lambda y:  np.array(list(( map(lambda x:␣
 ↪[list([x*2])*4,list([x*2+1])*4],y)))).flatten()
```

Si se desea saber a que filas de la matriz de rigidez global van los elementos de la matriz de rigidez del elemento que une los nodos 0 y 1 usaríamos la función filask((0,1)).
Del mismo modo para las columnas de la matriz de rigidez del elemento que va del nodo 1 al nodo 2 las columnas serían columnask((1,2)).

[20]:
```
print(filask((1,2)))
print(columnask((1,2)))
```

```
[2 2 2 2 3 3 3 3 4 4 4 4 5 5 5 5]
[2 3 4 5 2 3 4 5 2 3 4 5 2 3 4 5]
```

Si se aplican las funciones a un conjunto de elementos se definen las siguientes funciones

[21]:
```
fco = lambda elementos: list(np.ravel(list(map(lambda u:␣
 ↪columnask((u[0],u[1])),elementos))))
ffi = lambda elementos: list(np.ravel(list(map(lambda u:␣
 ↪filask((u[0],u[1])),elementos))))
```

Como ejemplo para obtener las columnas de todos los elementos de todas las matrices de rigidez se ejecutaría `fco(elementos)`

[22]:
```
print(fco(elementos))
```

```
[0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 4, 5, 0, 1, 4, 5, 0, 1,
4, 5, 0, 1, 4, 5, 2, 3, 4, 5, 2, 3, 4, 5, 2, 3, 4, 5, 2, 3, 4, 5]
```

Una vez que se dispone de las matrices de rigidez de todos los elementos y las filas y columnas de cada elemento de cada matriz se pueden ensamblar en la matriz de rigidez global. Utilizaremos el formato de matriz sparse (`sp.csr_matrix`) del módulo de python `scipy` que a la vez que genera la matriz sparse va sumando los distintos elementos.

[23]:
```
f_kg = lambda elems,x: sp.csr_matrix(sp.
 ↪coo_matrix((f_kg1(elems,x),(fco(elems),ffi(elems))),shape=(len(x)*2,len(x)*2)))
```

La matriz obtenida es de tipo *sparse*. Para poder escribirla en el formato convencional se convierte a densa.

[24]:
```
print(f_kg(elementos,x).todense())
```

```
[[ 589.4427191  178.8854382 -500.           0.          -89.4427191
  -178.8854382]
 [ 178.8854382  357.7708764    0.           0.         -178.8854382
  -357.7708764]
 [-500.           0.          589.4427191 -178.8854382  -89.4427191
   178.8854382]
 [   0.           0.         -178.8854382  357.7708764  178.8854382
  -357.7708764]
 [ -89.4427191 -178.8854382  -89.4427191  178.8854382  178.8854382
     0.       ]
 [-178.8854382 -357.7708764  178.8854382 -357.7708764    0.
   715.5417528]]
```

4.6 Condiciones de contorno

La matriz obtenida es singular dado que no hemos incluido ninguna condición de contorno, como se puede comprobar obteniendo el determinante.

[25]:
```
np.linalg.det(f_kg(elementos,x).todense())
```

[25]: 0.0

Las condiciones de contorno podrían incluirse eliminando las filas y columnas de la matriz de rigidez correspondientes a los grados de libertad restringidos, pero una forma más simple es imponiendo las restricciones por penalización. Se añaden en los elementos de la diagonal de la matriz de rigidez correspondientes a los grados de libertad restringidos unos valores de rigidez muy elevados que imponen de forma aproximada el cumplimiento de la restricción.
De este modo se mantienen las dimensiones de las matrices que en el otro caso habrían cambiado.

Si denominamos al valor elevado de rigidez *kpen* y tenemos en el nodo i restringido el grado de libertad j basta con añadir el valor *kpen* al elemento de la matriz de rigidez global $(2i+j, 2i+j)$

Para poder incluir estas condiciones de contorno de forma sencilla se prepara una función `f_kgp` que utiliza la matriz de rigidez global y añade los elementos correspondientes (`f_cc1` devuelve el valor de los elementos y `f_cc2` los índices fila y columna en la matriz de rigidez global) en el sitio adecuado.

[26]:
```
kpen = 1e20
f_cc1 = lambda cc: list(map(lambda u: kpen,cc))
f_cc2 = lambda cc: list(map(lambda u: u[0]*2+u[1],cc))
f_kgp = lambda elems,x,cc: sp.csr_matrix(sp.coo_matrix((np.
 ↪hstack((f_kg1(elems,x),f_cc1(cc))),(fco(elems)+f_cc2(cc),ffi(elems)+f_cc2(cc))),shape=(len(x)
```

Como se puede comprobar, ahora la matriz de rigidez modificada ya no es singular.

[27]:
```
np.linalg.det(f_kgp(elementos,x,cc).todense())
```

[27]:
```
6.400000000000038e+67
```

4.7 Fuerzas

El vector de fuerzas se obtiene de forma similar a la matriz de rigidez. Se crea una matriz sparse con una sola columna a partir de los valores de las fuerzas aplicadas y de los grados de libertad correspondientes.
Para ello usaremos dos funciones auxiliares.
1. `f_b1` transforma la lista de fuerzas en una lista con los valores de las fuerzas y las filas del vector global f (además de un 0 para la columna) 2. `f_b` ensambla el vector de fuerzas en una matriz sparse de forma similar a `f_kg1`

[28]:
```
f_b1 = lambda fuerzas: (list(map(lambda u: u[2],fuerzas)),(list(map(lambda u:␣
 ↪u[0]*2+u[1],fuerzas)),list(map(lambda u: 0,fuerzas))))
f_b = lambda fuerzas,x: sp.csr_matrix(sp.
 ↪coo_matrix((f_b1(fuerzas)),shape=(len(x)*2,1)))
```

[29]:
```
print(f_b(fuerzas,x))
```

```
  (4, 0)        1
```

Hemos obtenido una matriz sparse de 6×1 con un elemento no nulo de valor 1 en el índice 4 que se corresponde con el nodo 2 y dirección x

Una vez que tenemos la matriz de rigidez global y el vector de fuerzas global podemos resolver el sistema de ecuaciones

[30]:
```
d = spsolve(f_kgp(elementos,x,cc),f_b(fuerzas,x)).reshape(3,2)
print(d)
```

```
[[ 1.00000000e-20  1.00000000e-20]
 [ 1.00000000e-03 -1.00000000e-20]
 [ 6.09016994e-03 -2.50000000e-04]]
```

Podemos crear una función para resolver el problema

[31]:
```
f_solve = lambda x,els,fuerzas,cc: spsolve(f_kgp(els,x,cc),f_b(fuerzas,x)).
 ↪reshape(len(x),2)
```

[32]:
```
d = f_solve(x,elementos,fuerzas,cc)
print(d)
```

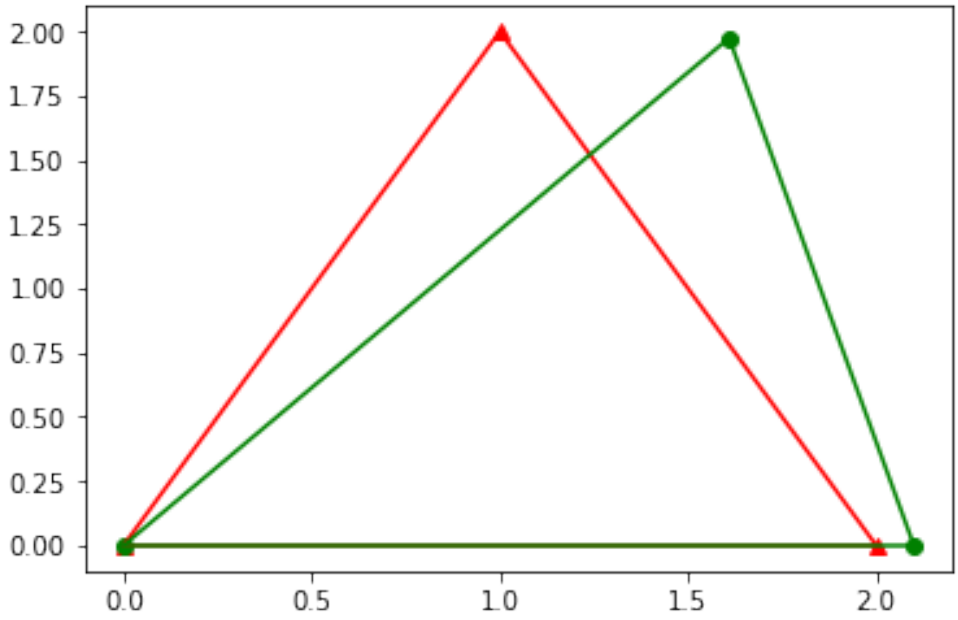
```
[[ 1.00000000e-20  1.00000000e-20]
 [ 1.00000000e-03 -1.00000000e-20]
 [ 6.09016994e-03 -2.50000000e-04]]
```

Utilizaremos las siguientes funciones para dibujar los nodos o los elementos y que tienen como argumentos las coordenadas de los nodos, para el caso de los nodos y las coordenadas y los elementos para la de los elementos. Aparte tienen un argumento opcional para indicar el símbolo y/o color a usar.

[33]:
```
f_dibnodos= lambda x,color='go': plt.plot(x[:,0],x[:,1],color)
f_dibelems = lambda x,elems,color='r':plt.plot(*(sum(list(map(lambda u:␣
 ↪[(x[u[0]][0],x[u[1]][0]),(x[u[0]][1],x[u[1]][1]),color],elems)),[] )))
```

[34]:
```
f_dibnodos(x,'r^')
f_dibnodos(x+100*d,'go')
f_dibelems(x,elementos,'r')
f_dibelems(x+100*d,elementos,'g')
```

[34]:
```
[<matplotlib.lines.Line2D at 0xa79af270>,
 <matplotlib.lines.Line2D at 0xa79af330>,
 <matplotlib.lines.Line2D at 0xa79af570>]
```



Para obtener los esfuerzos en las barras hay que obtener el movimiento axial y multiplicarlo por la rigidez de la barra. Utilizaremos primeramente la función `f_Ne` que devuelve el axil de un elemento dado, utilizando como argumentos el elemento, las coordenadas de los nodos y los movimientos de los nodos.
Posteriormente para obtener los axiles de todos los elementos tenemos la función `f_N` que aplica `f_Ne` a todos los elementos.

[35]:
```
f_N1 = lambda elemento,x: fk_1(fk_0(elemento,x))
f_Ne = lambda elemento,x,u: (elemento[2]/(fk_1(fk_0(elemento,x))[3])**2)*(np.
 ↪dot(f_N1(elemento,x)[-2:],f_N1(elemento,u)[-2:]))
f_N = lambda elementos,x,u: list(map(lambda v: f_Ne(v,x,u),elementos))
```

Axil del elemento 1

[36]:
```
f_Ne(elementos[1],x,d)
```

[36]:
```
1.1180339887498947
```

Obtención de todos los axiles

[37]:
```
f_N(elementos,x,d)
```

[37]:
```
[0.5, 1.1180339887498947, -1.118033988749895]
```

4.8 Movimientos impuestos

Si tenemos algún movimiento impuesto en el modelo se puede imponer mediante penalización. Se impone una restricción en el grado de libertad correspondiente y se aplica una fuerza en ese mismo grado de libertad de valor el movimiento impuesto multiplicado por la constante de penalización.
Veamos el ejemplo anterior donde en vez de aplicar una fuerza se desea que el nodo 2 se mueva -0.2 en direción x Añadiríamos una nueva condición de contorno y aplicaríamos la fuerza correspondiente.

[38]:
```
cc2 = cc+[[2,0]]
fuerzas2 = [[2,0,-kpen*0.2]]
```

Resolvemos y obtenemos el resultado previsto.

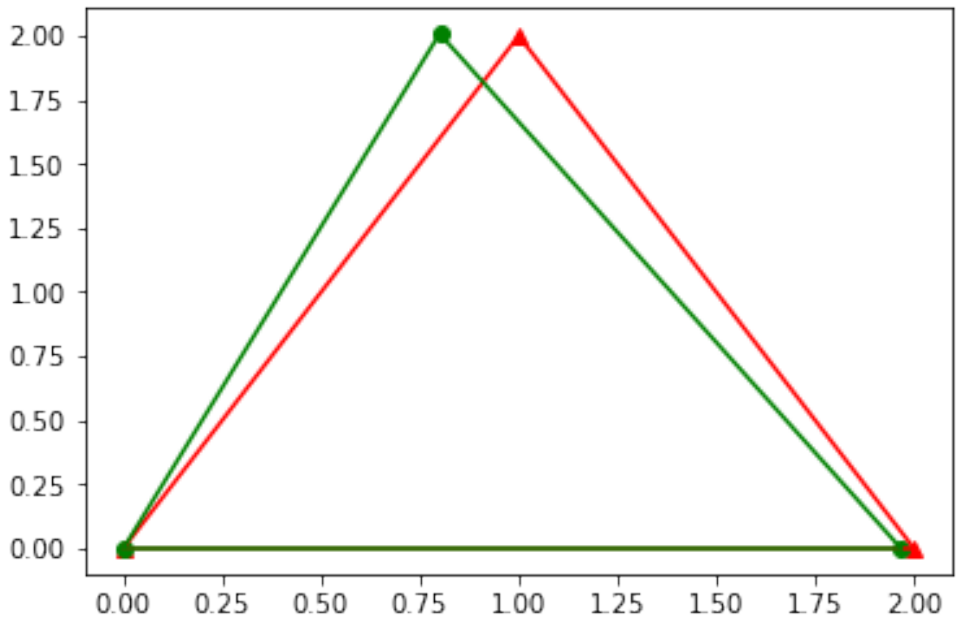
[39]:
```
d2 = f_solve(x,elementos,fuerzas2,cc2)
print(d2)
```

```
[[-3.28398061e-19 -3.28398061e-19]
 [-3.28398061e-02  3.28398061e-19]
 [-2.00000000e-01  8.20995152e-03]]
```

que se puede dibujar utlizando los movimientos a escala 1:1

[40]:
```
f_dibnodos(x,'r^')
f_dibnodos(x+d2,'go')
f_dibelems(x,elementos,'r')
f_dibelems(x+d2,elementos,'g')
```

[40]:
```
[<matplotlib.lines.Line2D at 0xa791cf10>,
 <matplotlib.lines.Line2D at 0xa791cfd0>,
 <matplotlib.lines.Line2D at 0xa7924230>]
```



4.9 Generación de nodos

Con el fin facilitar la generación de los datos de los modelos vamos a incluir unas funciones para la generación de coordenadas y nodos. La función que usaremos es gennodos. Tiene los siguientes argumentos:

1. xf. Función de las variables (u,v)
2. yf. Función de las variables (u,v)
3. ues. Rango de los valores de u en la forma [u0,uf,nu]
4. ves. Rango de los valores de v en la forma [v0,vf,nv]
5. patterns. Esquema de numeración de los nodos.
6. n0. Número del primer nodo generado.
7. ntot. Número total de nodos del modelo.

y devuelve el array de coordenadas de los nodos.
Las funciones `xf(u,v)->u` y `yf(u,v)->v` están ya predefinidas.

El único argumento que hay que explicar es `patterns` que establece como se van asignando números de nodos a las coordenadas que se van generando. Se generan $nu \times nv$ coordenadas que se obtienen evaluando las funciones `xf` e `yf` sobre el producto cartesiano de los `nu` valores generados entre `u0` y `uf` por los `nv` valores generados entre `v0` y `vf`.
La forma de asignar los nodos es utilizando la lista patterns `[[d1,s1],[d2,s2],...]`
Se van asignando nodos a las coordenadas empezando por `n0`, a cada incremento de `s1` de las coordenadas (las $nu \times nv$ coordenadas) se incrementa `d1` el índice del nodo. Se hace lo mismo con cada par de valores `[d,s]` de la lista patterns.

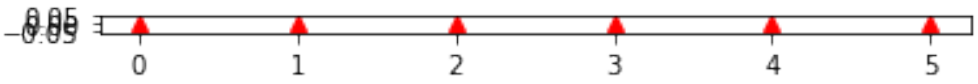
[41]:
```
xf = lambda u,v: u
yf = lambda u,v: v
f1  =lambda u: np.linspace(u[0],u[1],u[2])
f2 = lambda u,v : np.meshgrid(f1(u),f1(v))
f3 = lambda xu,u,v: xu(*f2(u,v)).reshape(u[2]*v[2],1)
f4 = lambda xu,yu,u,v : np.hstack((f3(xu,u,v),f3(yu,u,v)))
faux = lambda patterns,ntot: map(lambda x: [x[0],x[1],ntot],patterns)
fn1 = lambda d1,s1,n : (np.cumsum((np.ones(int(n/s1)))).reshape((int(n/
 ↪s1),1))*d1-d1 + np.zeros(s1)).reshape(1,n)
fn2 = lambda d1,s1,n: fn1(d1,s1,(int((n-1)/s1)+1)*s1)[0][0:n].reshape(1,n)
fnumer = lambda ues,ves,patterns,n0: np.sum(list(map(lambda u:␣
 ↪fn2(*u),faux(patterns,ues[2]*ves[2]))),dtype=int,axis=0)+n0
nauxf = lambda nnodos: np.hstack((np.ones((1,nnodos),dtype=int)*0,np.
 ↪ones((1,nnodos),dtype=int))).reshape(nnodos*2)
ensambf = lambda puntos,nodos,ntot: sp.coo_matrix((np.hstack((puntos[:
 ↪,0],puntos[:,1])), (np.hstack((nodos,nodos)).reshape(nodos.size*2),␣
 ↪nauxf(nodos.size))),shape=(ntot,2)).toarray()
gennodos=lambda xf,yf,ues,ves,patterns,n0,ntot:
 ↪ensambf(f4(xf,yf,ues,ves),fnumer(ues,ves,patterns,n0),ntot)
```

Si queremos generar 6 nodos separados 1 en el eje x a partir del origen.

[42]:
```
xl1 = gennodos(xf,yf,[0,5,6],[0,0,1],[[1,1]],0,6)
```

[43]:
```
ax = plt.gca()
ax.set_aspect('equal')
f_dibnodos(xl1,'r^')
```

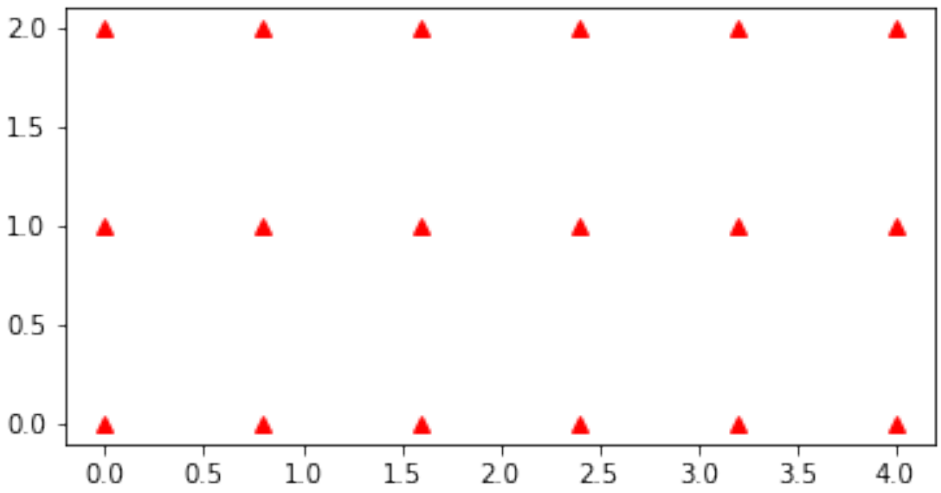
[43]:
```
[<matplotlib.lines.Line2D at 0xa78f5390>]
```



Si queremos generar una malla de nodos en un rectángulo de lados 4x2 y queremos 6 nodos en el lado sobre el eje x y 3 sobre el eje y, con la numeración empezando por cero y creciendo según el eje x

[44]:
```
xl2 = gennodos(xf,yf,[0,4,6],[0,2,3],[[1,1]],0,18)
ax = plt.gca()
ax.set_aspect('equal')
f_dibnodos(xl2,'r^')
```

[44]:
```
[<matplotlib.lines.Line2D at 0xa78ab590>]
```



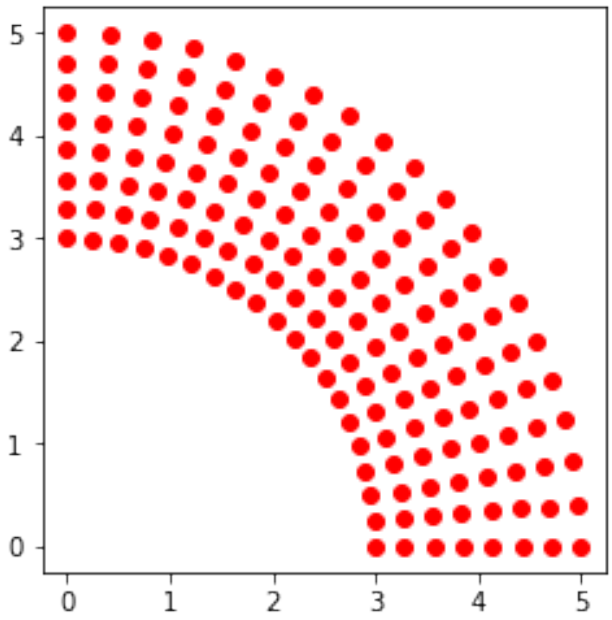
[45]:
```
print(xl2)
```

```
[[0.  0. ]
 [0.8 0. ]
 [1.6 0. ]
 [2.4 0. ]
 [3.2 0. ]
 [4.  0. ]
 [0.  1. ]
 [0.8 1. ]
 [1.6 1. ]
 [2.4 1. ]
 [3.2 1. ]
 [4.  1. ]
 [0.  2. ]
 [0.8 2. ]
 [1.6 2. ]
 [2.4 2. ]
 [3.2 2. ]
 [4.  2. ]]
```

Si queremos una malla de nodos sobre un cuarto de un sector circular entre los radios 3 y 5 con 8 nodos en la dirección del radio y 20 nodos circunferencialmente.

[46]:
```
xf3 = lambda u,v: u*np.cos(v)
yf3 = lambda u,v: u*np.sin(v)
xl3 = gennodos(xf3,yf3,[3,5,8],[0,np.pi/2,20],[[1,1]],0,160)
ax = plt.gca()
ax.set_aspect('equal')
f_dibnodos(xl3,'ro')
```

[46]: [<matplotlib.lines.Line2D at 0xa78741f0>]



4.10 Generación de elementos

Para la generación de elementos se seguirá un esquema similar al de la generación de nodos salvo que cada elemento tiene varios nodos sobre los que se aplicará el mismo esquema. De hecho usaremos la función `fnumer` de la generación de nodos. Utilizaremos la función genelem que tiene los siguientes argumentos:
1. `patterns` funciona de forma similar a la generación de nodos y tendrá un valor del tipo `[[d11,d12,s1],[d21,d22,s2]...]` 2. `elbase`. Son los nodos del primer elemento. 3. `nelem`. Número de elementos a generar. 4. `x`. Coordenadas del modelo. 5. `eaf`. Una función dependiente de (x,y) que evaluada en el punto medio de cada barra nos devuelve el valor de EA, siendo E el módulo de Young y A la sección de la barra.

La función `fc` predefinida es la función que con un argumento constante genera la función constante.

[47]:
```
fc = lambda f: lambda x,y: f
fpat3 = lambda pat,elbase,nelem: list(map(lambda i:␣
 ↪[[0,1,nelem],[0,1,1]]+[list(map(lambda x: [x[i],x[-1]],pat))]+[elbase[i]],np.
 ↪arange(len(elbase))))
genelem0 = lambda patterns,elbase,nelem: np.hstack(list(map(lambda x: fnumer(*x).
 ↪reshape(nelem,1),fpat3(patterns,elbase,nelem)))).tolist()
genelem1 = lambda elementos,x,eaf:list(map(lambda u: [u[0],u[1],eaf(0.
 ↪5*(x[u[0]][0]+x[u[1]][0]),0.5*(x[u[0]][1]+x[u[1]][1]))],elementos))
```

```
genelem = lambda patterns,elbase,nelem,x,eaf:
 ↪genelem1(genelem0(patterns,elbase,nelem),x,eaf)
```

4.11 Generacion de fuerzas nodales con una funcion de (x,y) evaluada en el nodo

Se generan de forma similar a los elementos.

[48]:
```
genfuerzas0 = lambda patterns,elbase,nelem: np.hstack(list(map(lambda x:␣
 ↪fnumer(*x).reshape(nelem,1),fpat3(patterns,elbase,nelem)))).tolist()
genfuerzas1 = lambda elementos,x,ffuer:list(map(lambda u:␣
 ↪[u[0],u[1],ffuer(x[u[0]][0],x[u[0]][1])],elementos))
genfuerzas = lambda patterns,elbase,nelem,x,eaf:
 ↪genfuerzas1(genfuerzas0(patterns,elbase,nelem),x,eaf)
```

4.11.1 Ejemplos de generación de elementos.

Generación de elementos en una dimensión correspondiente a los nodos xl1 generados anteriormente.

[49]:
```
genelem([[1,1,1]],[0,1],4,xl1,fc(1000))
```

[49]:
```
[[0, 1, 1000], [1, 2, 1000], [2, 3, 1000], [3, 4, 1000]]
```

5 Solución del problema de abaqus 1d

A continuación se va a resolver con python el problema 1d resuelto con abaqus.
La generación de nodos y de elementos es trivial.

[50]:
```
x = gennodos(xf,yf,[0,10,11],[0,0,1],[[1,1]],0,11)
```

[51]:
```
elems = genelem([[1,1,1]],[0,1],10,x,fc(1000))
```

Las condiciones de contorno se pueden generar con una de las funciones auxiliares de generar elementos.
Hay que tener en cuenta que como el problema va a ser unidimensional se deben restringir los movimientos en la otra dirección.
Es lo que haremos con las condiciones `cc1`

[52]:
```
cc1 = genelem0([[1,0,1]],[0,1],11)
```

[53]:
```
cc1
```

[53]:
```
[[0, 1],
 [1, 1],
 [2, 1],
 [3, 1],
 [4, 1],
```

```
 [5, 1],
 [6, 1],
 [7, 1],
 [8, 1],
 [9, 1],
 [10, 1]]
```

Y se añade la condición de contorno que falta de movimiento coaccionado en x en el nodo 0.

[54]:
```
cc = cc1 + [[0,0]]
```

Para la inclusión de la carga volumétrica se preparan unas funciones auxiliares para poder generar las cargas nodales equivalentes. `fq1` calula las fuerzas nodales equivalentes en los nodos iniciales de cada barra y `fq2` las de los nodos finales.

[55]:
```
fq = lambda x,y: 0.2+0.04*x
fq1 = lambda x,y: 1/6*(2*fq(x,0)+fq(x+1,0))
fq2 = lambda x,y: 1/6*(fq(x-1,0)+2*fq(x,0))
```

[56]:
```
fuerz1 = genfuerzas([[1,0,1]],[0,0],10,x,fq1)
fuerz2 = genfuerzas([[1,0,1]],[1,0],10,x,fq2)
```

Añado la fuerza nodal del extremo libre.

[57]:
```
fuerzas = fuerz1+fuerz2+[[10,0,5]]
```

[58]:
```
fuerzas
```

[58]:
```
[[0, 0, 0.10666666666666666],
 [1, 0, 0.12666666666666665],
 [2, 0, 0.14666666666666667],
 [3, 0, 0.16666666666666666],
 [4, 0, 0.18666666666666668],
 [5, 0, 0.20666666666666667],
 [6, 0, 0.22666666666666668],
 [7, 0, 0.24666666666666665],
 [8, 0, 0.26666666666666666],
 [9, 0, 0.2866666666666667],
 [1, 0, 0.11333333333333334],
 [2, 0, 0.13333333333333333],
 [3, 0, 0.15333333333333332],
 [4, 0, 0.17333333333333334],
 [5, 0, 0.19333333333333336],
 [6, 0, 0.21333333333333332],
 [7, 0, 0.23333333333333334],
 [8, 0, 0.2533333333333333],
 [9, 0, 0.2733333333333333],
 [10, 0, 0.29333333333333333],
```

```
 [10, 0, 5]]
```

[59]:
```
u = f_solve(x,elems,fuerzas,cc)
print(u)
```

```
[[9.00000000e-20 0.00000000e+00]
 [8.89333333e-03 0.00000000e+00]
 [1.75466667e-02 0.00000000e+00]
 [2.59200000e-02 0.00000000e+00]
 [3.39733333e-02 0.00000000e+00]
 [4.16666667e-02 0.00000000e+00]
 [4.89600000e-02 0.00000000e+00]
 [5.58133333e-02 0.00000000e+00]
 [6.21866667e-02 0.00000000e+00]
 [6.80400000e-02 0.00000000e+00]
 [7.33333333e-02 0.00000000e+00]]
```

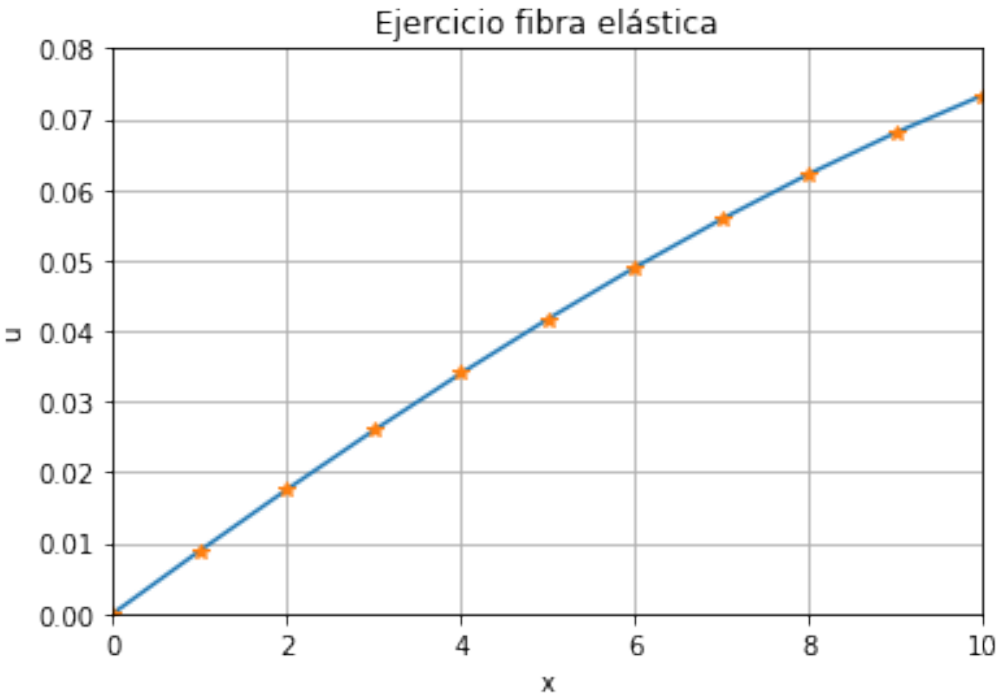
Leemos los resultados de abaqus que estarán preparados en un fichero con dos columnas de números separados por comas. Utilizaremos el módulo pandas para la lectura.

[60]:
```
import pandas as pd
data = pd.read_csv('u3.txt', header = None)
u3_abaqus = data.to_numpy()
```

Y podemos representar la comparación entre los resultados de abaqus (los puntos del dibujo) y los calculados directamente.

[61]:
```
plt.xlabel('x')
plt.ylabel('u')
plt.title('Ejercicio fibra elástica')
plt.axis([0, 10, 0, 0.08])
plt.grid(True)
plt.plot(x[:,0],u[:,0],x[:,0],u3_abaqus[:,1],'*')
```

[61]:
```
[<matplotlib.lines.Line2D at 0xa5f49110>,
 <matplotlib.lines.Line2D at 0xa5f49210>]
```

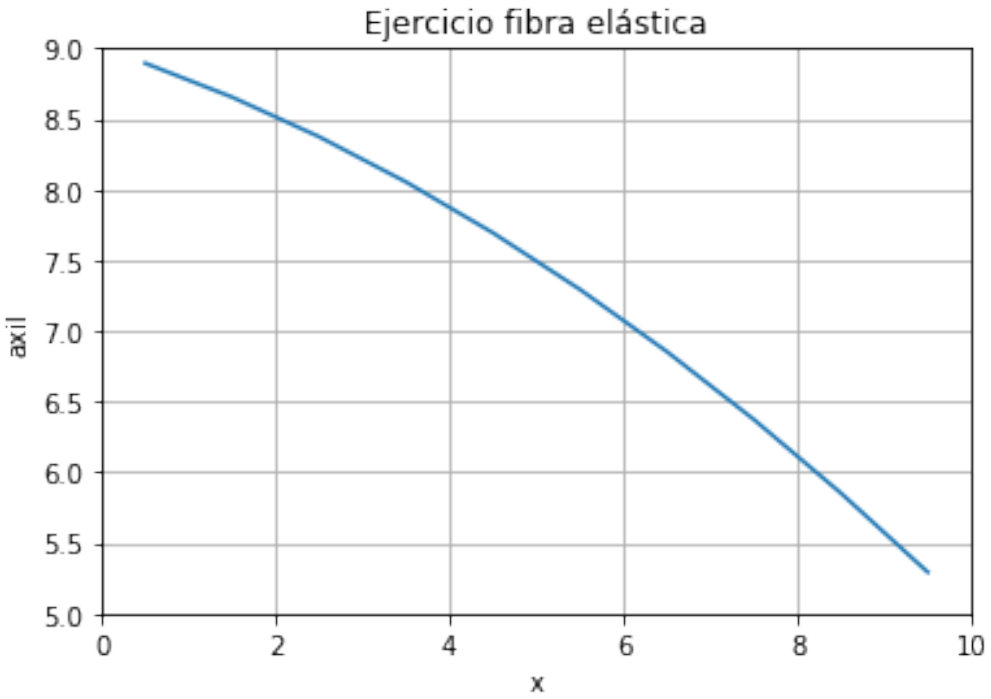


Se puede hacer lo mismo entre los axiles y la tensión S33 obtenida en **abaqus** dado que al ser la sección de 1×1 coinciden numéricamente. Las tensiones se representarán en el punto geométrico en el que han sido obtenidas, es decir, el punto de Gauss del elemento.

[62]:
```
axiles = f_N(elems,x,u)
```

[63]:
```
plt.xlabel('x')
plt.ylabel('axil')
plt.title('Ejercicio fibra elástica')
plt.axis([0, 10, 5, 9])
plt.grid(True)
plt.plot(np.linspace(0.5,9.5,10),axiles[:])
```

[63]:
```
[<matplotlib.lines.Line2D at 0xa5f077b0>]
```



Lectura de las tensiones.

[64]:
```
import pandas as pd
data = pd.read_csv('s33.rpt', header = None)
s33_abaqus = data.to_numpy()
```

Al comprobar que están ordenadas al revés invertiremos la lista de valores mediante `np.flip`

[65]:
```
s33_abaqus[:,2]
```

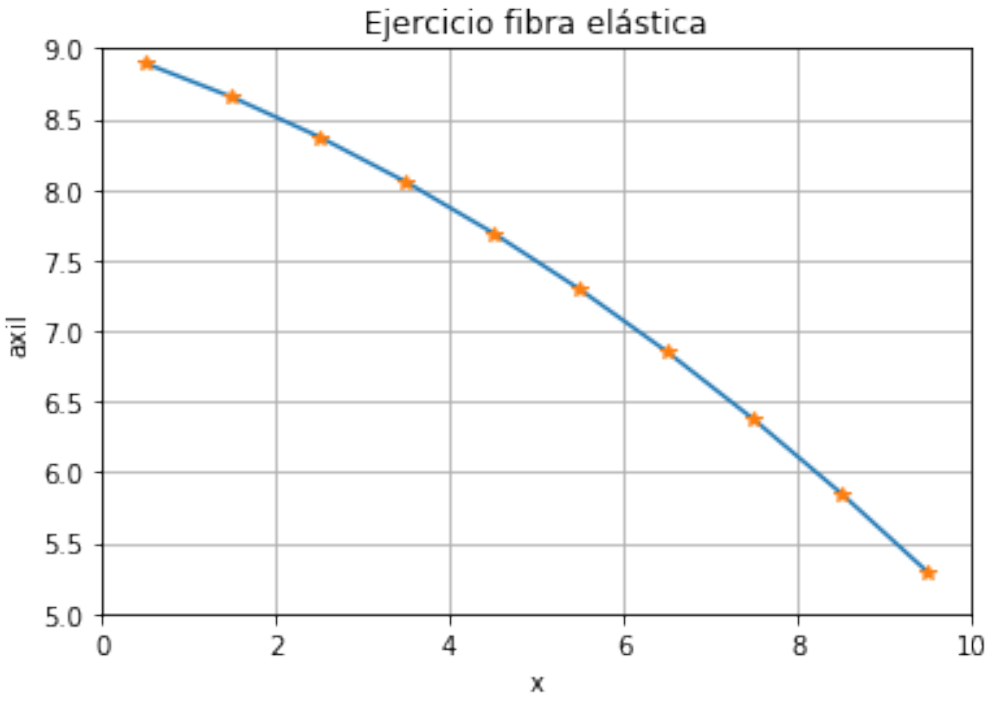
[65]:
```
array([5.29, 5.85, 6.37, 6.85, 7.29, 7.69, 8.05, 8.37, 8.65, 8.89])
```

[66]:
```
s33_ab = np.flip(s33_abaqus[:,2])
```

Y por último obtenemos la representación de la comparación entre los dos resultados (S33 de **abaqus** y axil del cálculo en python).

[67]:
```
plt.xlabel('x')
plt.ylabel('axil')
plt.title('Ejercicio fibra elástica')
plt.axis([0, 10, 5, 9])
plt.grid(True)
plt.plot(np.linspace(0.5,9.5,10),axiles[:],np.linspace(0.5,9.5,10),s33_ab,'*')
```

```
[67]: [<matplotlib.lines.Line2D at 0xa5ecc470>,
       <matplotlib.lines.Line2D at 0xa5ecc550>]
```

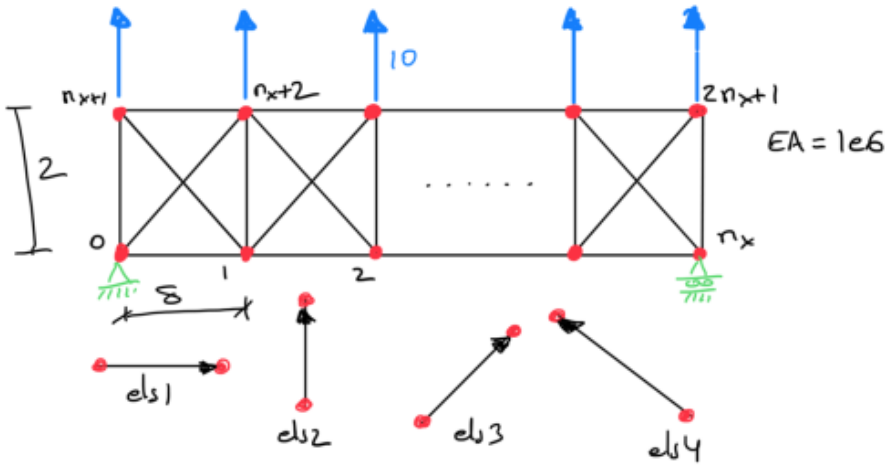


6 Ejemplos

A continuación se resolverán algunos ejemplos donde se vea también el uso de las funciones generadoras de nodos, elementos, etc...

6.1 Viga recta de celosía.

El esquema que usaremos es el siguiente:



[68]:
```
nx = 15
x = gennodos(xf,yf,[0,nx*8,nx+1],[0,2,2],[[1,1]],0,2*nx+2)
```

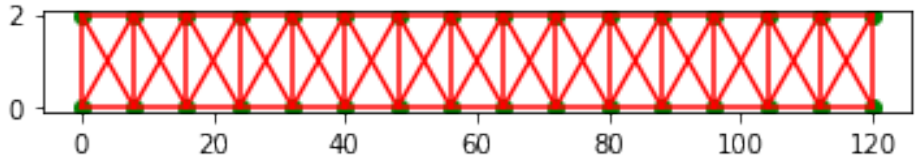
[69]:
```
els1 = genelem([[1,1,1],[1,1,nx]],[0,1],2*nx,x,fc(1e6))
```

[70]:
```
els2 = genelem([[1,1,1]],[0,nx+1],nx+1,x,fc(1e6))
```

[71]:
```
els3 = genelem([[1,1,1]],[0,nx+2],nx,x,fc(1e6))
els4 = genelem([[1,1,1]],[1,nx+1],nx,x,fc(1e6))
```

[72]:
```
elementos = els1+els2+els3+els4
```

[73]:
```
ax = plt.gca()
#ax.set_aspect('equal')
ax.set_aspect(7)
f_dibnodos(x);
f_dibelems(x,elementos);
```



Las condiciones de contorno son: nodo inferior izquierdo fijo y nodo inferior derecho con un carrito.
Las fuerzas son unas fuerzas concentradas en los nodos del cordón superior.

[74]:
```
cc = [[0,0],[0,1],[nx,1]]
fuerzas = genfuerzas([[1,0,1]],[nx+1,1],nx+1,x,fc(10))
```

[75]:
```
fuerzas
```

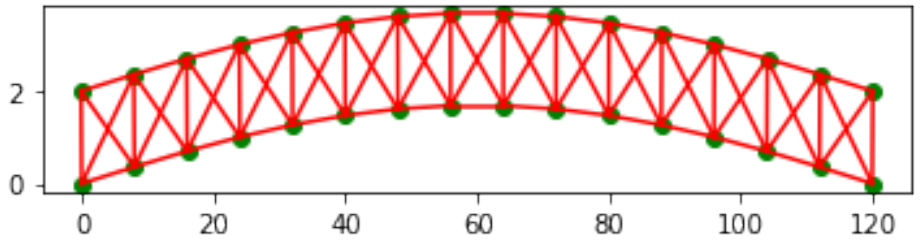
[75]:
```
[[16, 1, 10],
 [17, 1, 10],
 [18, 1, 10],
 [19, 1, 10],
 [20, 1, 10],
 [21, 1, 10],
 [22, 1, 10],
 [23, 1, 10],
 [24, 1, 10],
 [25, 1, 10],
 [26, 1, 10],
 [27, 1, 10],
 [28, 1, 10],
 [29, 1, 10],
 [30, 1, 10],
 [31, 1, 10]]
```

Una vez que están creadas todas las variables que definen el modelo se resuelve.

[76]:
```
u=f_solve(x,elementos,fuerzas,cc)
```

Dibujo de los movimientos de la viga.

[77]:
```
ax = plt.gca()
#ax.set_aspect('equal')
ax.set_aspect(7)
f_dibnodos(x+u);
f_dibelems(x+u,elementos);
```



[78]:
```
plt.show()
```

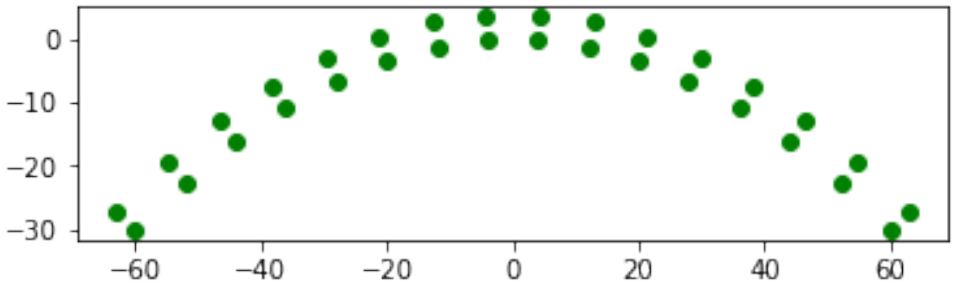
6.2 Arco parabólico de celosía

Podemos generar el modelo con forma de arco parabólico. Basta con utilizar las ecuaciones de la parábola.
En este caso para una luz de 120 metros y una flecha de 30 metros.
Se ha considerado que el canto de la viga se genera según la normal a la parábola.

[79]:
```
nx = 15
xf2 = lambda u,v: u+v*(2/120*u)/np.sqrt(1+4*(1/120)*(1/120)*u*u)
yf2 = lambda u,v: -1/120*u*u+v*1/np.sqrt(1+(4/120)*(1/120)*u*u)
x = gennodos(xf2,yf2,[-nx*4,nx*4,nx+1],[0,4,2],[[1,1]],0,2*nx+2)
```

Dibujamos los nodos solamente.

[80]:
```
ax = plt.gca()
ax.set_aspect('equal')
f_dibnodos(x);
```



Los elementos son exactamente los mismos que en el caso anterior.

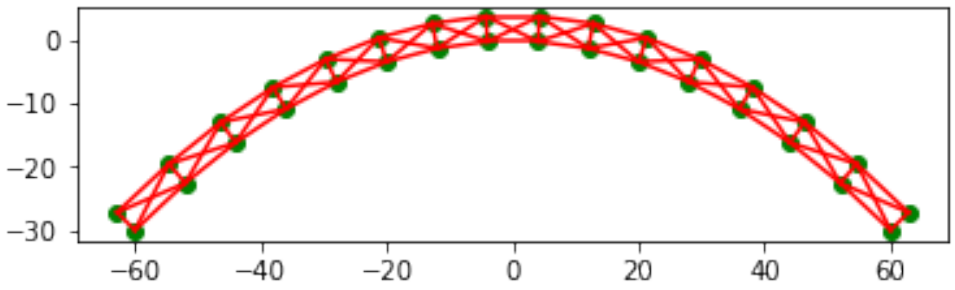
[81]:
```
els1 = genelem([[1,1,1],[1,1,nx]],[0,1],2*nx,x,fc(1e6))
```

[82]:
```
els2 = genelem([[1,1,1]],[0,nx+1],nx+1,x,fc(1e6))
```

[83]:
```
els3 = genelem([[1,1,1]],[0,nx+2],nx,x,fc(1e6))
els4 = genelem([[1,1,1]],[1,nx+1],nx,x,fc(1e6))
```

[84]:
```
elementos = els1+els2+els3+els4
```

[85]:
```
ax = plt.gca()
ax.set_aspect('equal')
f_dibnodos(x);
f_dibelems(x,elementos);
```

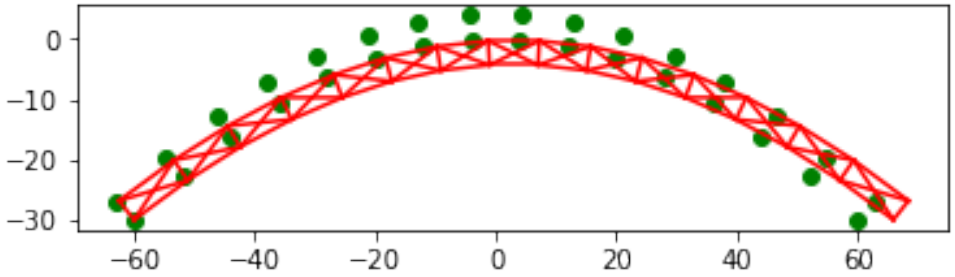


[86]:
```
cc = [[0,0],[0,1],[nx,1]]
fuerzas = genfuerzas([[1,0,1]],[nx+1,1],nx+1,x,fc(-10))
```

[87]:
```
u=f_solve(x,elementos,fuerzas,cc)
```

Dibujamos los nodos sin deformar y los elementos según la deformada amplificada 10 veces.

[88]:
```
ax = plt.gca()
ax.set_aspect('equal')
f_dibnodos(x);
f_dibelems(x+u*10,elementos);
```



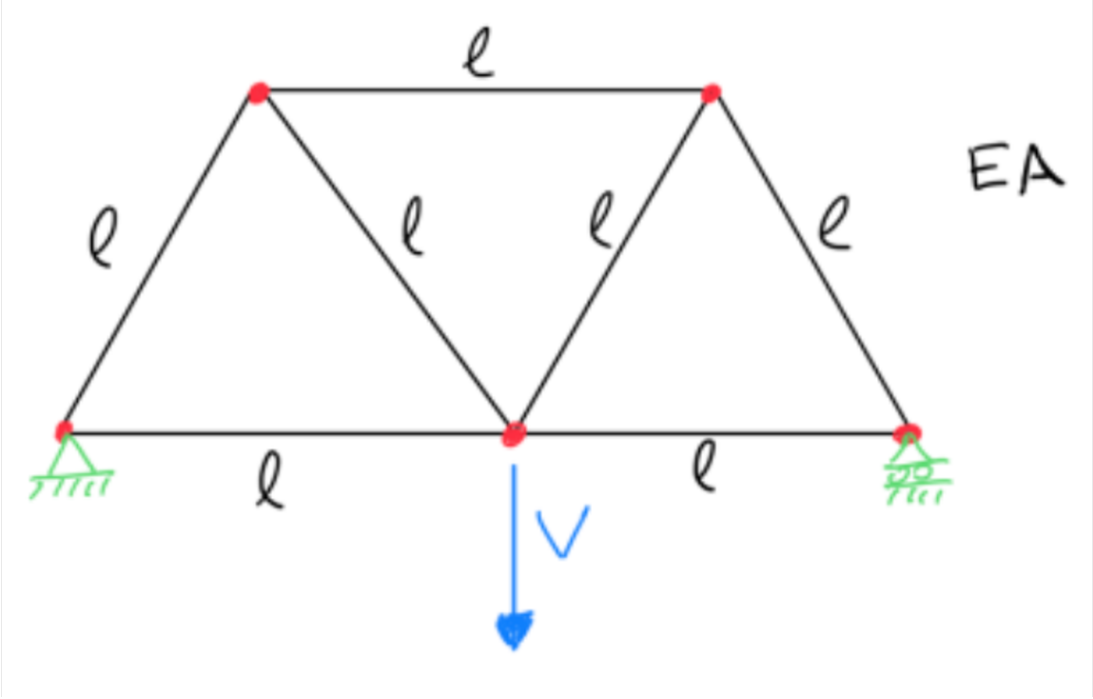
Aparte de los movimientos y de los axiles que ya hemos visto como se obtienen también podemos obtener las reacciones, que serán las fuerzas asociadas a las restricciones. En este caso, por ejemplo,las reacciones verticales están asociadas a los nodos `u[0]` y `u[nx]`.
Como el apoyo derecho es un carrito y no hay fuerzas horizontales las únicas reacciones son las verticales que se pueden obtener multiplicando los movimientos de los nodos coaccionados por la constante de penalización *kpen*
Las fuerzas aplicadas son unas fuerzas concentradas en los nodos superiores de valor -10 que hacen un total de -160.

[89]:
```
u[0]
```

```
[89]: array([ 5.85482703e-32, -8.00000000e-19])

[90]: u[nx]

[90]: array([ 5.93152636e-01, -8.00000000e-19])

[91]: u[0][1]*kpen

[91]: -79.99999999996999

[92]: u[nx][1]*kpen

[92]: -79.99999999996753
```

7 Ejercicio propuesto

Se propone resolver la siguiente cercha:



Con los siguientes valores:

l	EA	V
10	1000	10

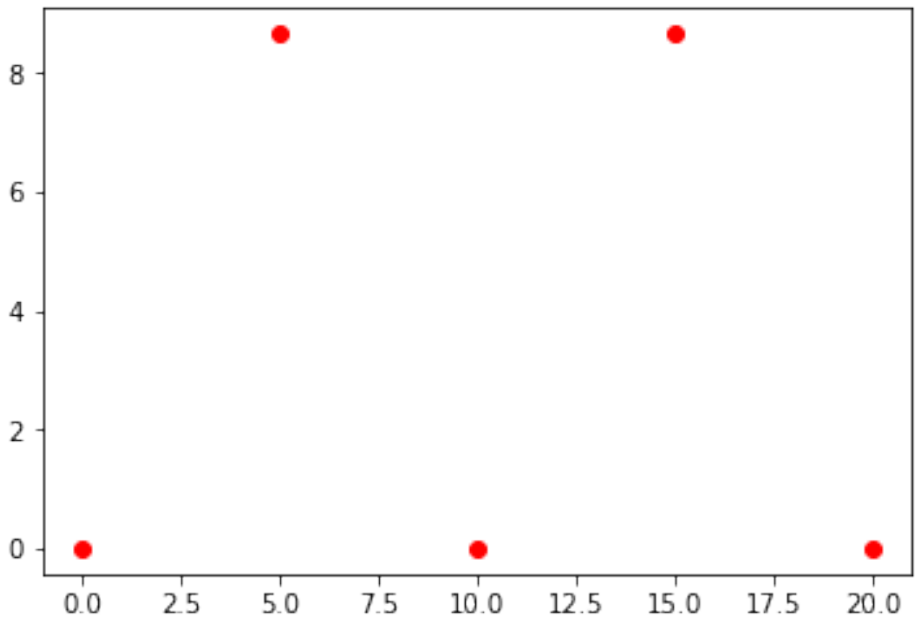
Y obtener el movimiento horizontal del apoyo derecho.

Se desarrollará la solución en función de los parámetros de la misma, `l,EA,V`, de modo que sea fácil cambiar los valores de los mismos.

Primeramente se definen las coordenadas de los nodos.

[93]:
```
l = 10
EA = 1000
V = 10
x = np.array([[0,0],[l,0],[2*l,0],[l/2.,l*math.sqrt(3)/2],[1.5*l,l*math.sqrt(3)/
 ↪2]])
f_dibnodos(x,'or')
```

[93]:
```
[<matplotlib.lines.Line2D at 0xa5e18b10>]
```

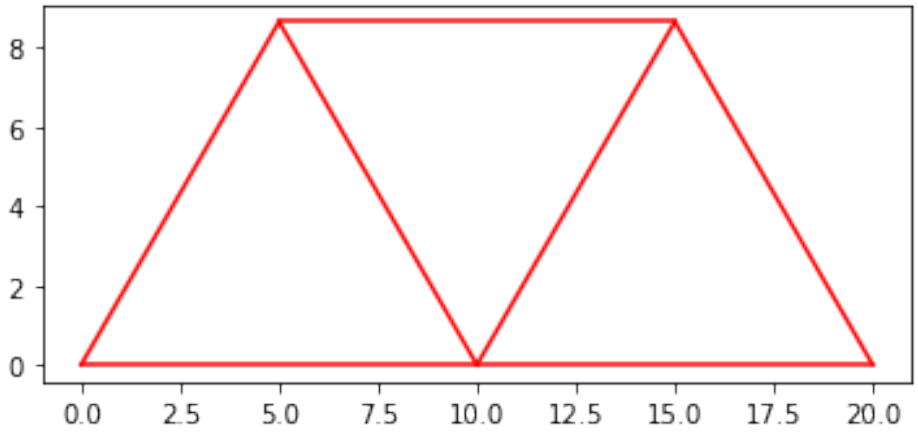


Y a continuación los elementos.

[94]:
```
elementos = [[0,1,EA],[1,2,EA],[0,3,EA],[1,3,EA],[3,4,EA],[1,4,EA],[2,4,EA]]
ax = plt.gca()
ax.set_aspect('equal')
f_dibelems(x,elementos)
```

[94]:
```
[<matplotlib.lines.Line2D at 0xa5e67fb0>,
 <matplotlib.lines.Line2D at 0xa5e6f0f0>,
 <matplotlib.lines.Line2D at 0xa5e6f390>,
 <matplotlib.lines.Line2D at 0xa5e6f5f0>,
```

```
<matplotlib.lines.Line2D at 0xa5e6f850>,
<matplotlib.lines.Line2D at 0xa5e6fab0>,
<matplotlib.lines.Line2D at 0xa5e6fcf0>]
```



[95]:
```
fuerzas = [[1,1,-V]]
```

[96]:
```
cc = [[0,0],[0,1],[2,1]]
```

[97]:
```
d =f_solve(x,elementos,fuerzas,cc)
```

[98]:
```
print(d)
```

```
[[ 1.20370622e-35 -5.00000000e-20]
 [ 2.88675135e-02 -1.83333333e-01]
 [ 5.77350269e-02 -5.00000000e-20]
 [ 5.77350269e-02 -1.00000000e-01]
 [ 2.16840434e-17 -1.00000000e-01]]
```

Se pide el movimiento del nodo del apoyo derecho.

[99]:
```
d[2][0]
```

[99]:
```
0.0577350269189626
```

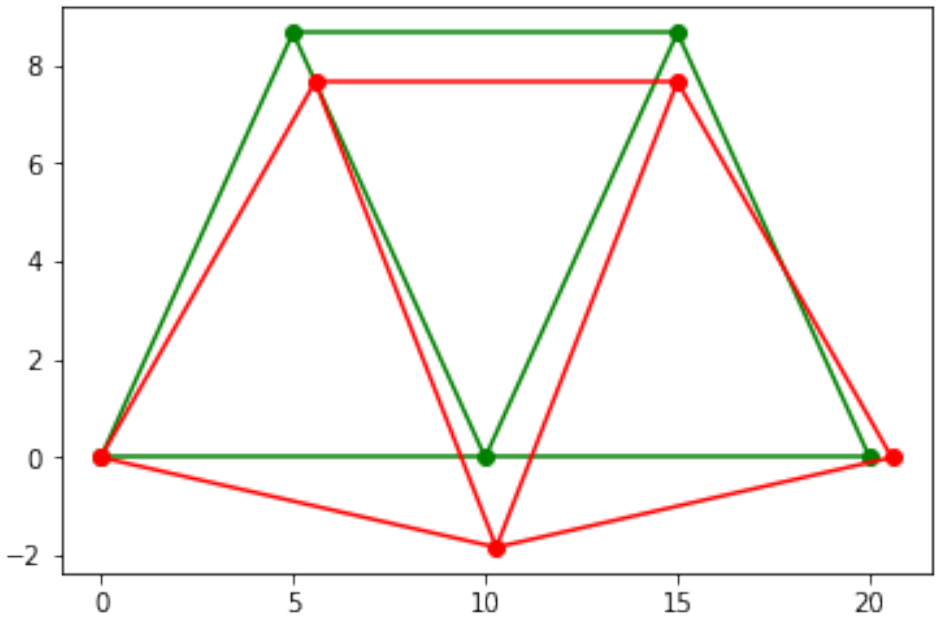
que si comparamos con la solución.

[100]:
```
(V*l)/(math.sqrt(3)*EA)
```

[100]:
```
0.05773502691896258
```

vemos que coincide.
Representamos el modelo con un factor de escala de 10

[101]:
```
f_dibnodos(x)
f_dibelems(x,elementos,'g')
f_dibnodos(x+10*d,'or')
f_dibelems(x+10*d,elementos)

plt.show()
```



A continuación convertiremos el problema a una función. Obtención del movimiento del apoyo derecho en función de `l,V y EA` y comparación con la solución analítica.

[102]:
```
f_x = lambda l: np.array([[0,0],[l,0],[2*l,0],[l/2.,l*math.sqrt(3)/2],[1.
↪5*l,l*math.sqrt(3)/2]])
```

[103]:
```
f_elementos = lambda EA:␣
↪[[0,1,EA],[1,2,EA],[0,3,EA],[1,3,EA],[3,4,EA],[1,4,EA],[2,4,EA]]
```

[104]:
```
f_fuerzas = lambda V:  [[1,1,-V]]
```

[105]:
```
f_fin = lambda l,V,EA: f_solve(f_x(l),f_elementos(EA),f_fuerzas(V),cc)
```

[106]:
```
f_fin(10,20,1000)[2,0]
```

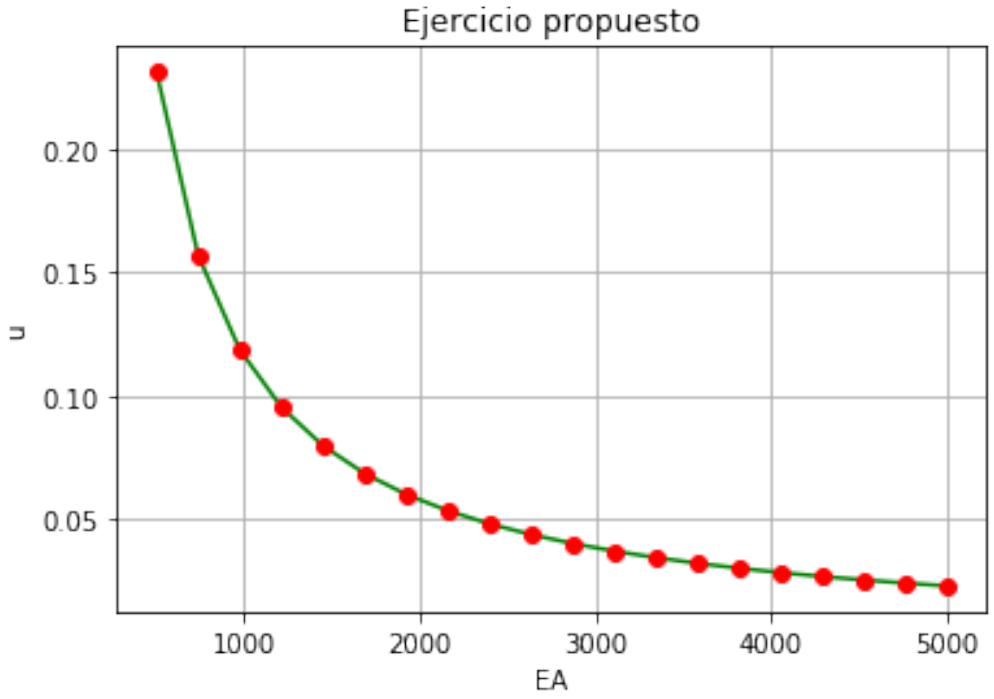
[106]:
```
0.1154700538379252
```

Y representaremos la solución obtenida para distintos valores de EA comparada con la solución analítica.

[107]:
```
xx = np.linspace(500,5000,20)
yy = list(map(lambda u: f_fin(10,20,u)[2,0],xx))
yy2 = list(map(lambda u: 20*10/(math.sqrt(3)*u),xx))
```

[108]:
```
plt.xlabel('EA')
plt.ylabel('u')
plt.title('Ejercicio propuesto')
plt.grid(True)
plt.plot(xx,yy,'g',xx,yy2,'or')
```

[108]:
```
[<matplotlib.lines.Line2D at 0xa5dc3670>,
 <matplotlib.lines.Line2D at 0xa5dc3d10>]
```



[ ]:

A. Simulation with MatLab / Octave

We give here the same code developed in `MatLab` for those of you familiar with this programming language (or with the opensource program `Octave`).

Código MatLab / Octave

The organisation of the `Matlab` / `Octave` code is free. However, given that this is the first contact of the student with FE coding, it is recommended to follow some patterns.

- Pre-process

  In the pre-processing we define the material, geometry of the structure, boundary conditions and finite element discretisation (mesh). Depending on the type of problem we may need to define also the total number of degrees of freedom.

Código 1: Matlab scheme *(a)*

```
% Definition of the parameters for each student:
Nmat=1124                       % Registration number of the student
m=floor(Nmat/1000)              % Thousands
c=floor((Nmat-m*1000)/100)      % Hundreds
d=floor((Nmat-m*1000-c*100)/10)  % Tens
u=Nmat-m*1000-c*100-d*10        % Units

% A: PRE-PROCESS
%-----------------------------------------------------------------

% 1. Geometry
A=1;                % Area
L=10;               % Length of the bar

% 2. Material
E=1000;             % Young's modulus

% 3. Forcing boundaries
P=5;                  % Force applied at en x=L
q0=d/10;              % Value of the volumetric force at x=0
qf=q0+u/10;          % Value of the volumetric force at x=L

% 4. Mesh
Nele=11-c;            % Number of elements
Nnod=Nele+1;          % Number of nodes
q=linspace(q0,qf,Nnod);
h=L/Nele;             % Length of each element

% 5. Number of degrees of freedom
% Dim = GDL total = degrees of freedom of each node x number of nodes
gdl=1;
Dim=Nnod*gdl;
```

- Solver - construction of the element and global matrices

  The first part of the *solver* builds the global stiffness and forcing matrices from the elementary ones. The elementary stiffness matrix is the same for all the elements because they have the same material and dimensions. However, the volumetric force varies along the bar. In addition, the last node (right end) has an applied force 5 N.

Código 2: Matlab scheme *(b)*

```
% B: CONSTRUCTION of GLOBAL MATRICES and VECTORS
%-----------------------------------------------------------------
% 0. Initialisation
K=zeros(Dim);
F=zeros(Dim,1);
Fext=zeros(Dim,1);

% 1. Elementary stiffness matrix (the same for all elements)
k=(E*A/h)*[1,-1;-1,1];

% 2. Assembly to obtain the global stiffness matrix (note capital letter in K)
for i=1:Nele
    K(i:i+1,i:i+1)=K(i:i+1,i:i+1)+k;        % Assembly
end

% 3. Assembly of the forcing vector (note capital letter in F)
for i=1:Nele
    % elementary vector of distributed forces
    fvol=(h/6)*[2*q(i)+q(i+1);q(i)+2*q(i+1)];
    % Assembly of loads
    F(i:i+1)=F(i:i+1)+fvol;
end

% 4. Sum of external forces
Fext(Nnod)=Fext(Nnod)+P;
F=F+Fext;           % Sum of forces at the ends and also distributed along the bar
```

- Solver - enforcement of boundary conditions and solution

  The core of the solver requires the definition of the boundary conditions to reduce the matrix system and make it non-singular to be able to invert the reduced matrix. Equilibrium is checked after the solution is obtained.

Código 3: Esquema Matlab *(c)*

```
% C: APPLICATION of the BOUNDARY CONDITIONS AND SOLUTION
%-----------------------------------------------------------------

% 1. Reduction of the stiffness matrix
Kg=K(2:Nnod,2:Nnod);

% 2. Reduction of the forcing vector
Fg=F(2:Nnod);

% 3. Solution of the system and calculation of nodal displacements
ug=Kg\Fg;

% 4. Displacements and reactions. Verification of equilibrium: Sum of forces = 0
un=[0;ug];
fint=K*un;
r_0=fint(1)-F(1);
Eq=sum(fint);
if abs(Eq)>1e-10
    Disp('Equilibrium not reached');
end
```

- Post-process

After the solution is obtained we want to obtain strains and then stresses at the element level. This is called *stress recovery*. The solution will then be compared with that from Abaqus and also with the analytical response in terms of displacements and stresses.

Código 4: Matlab scheme *(d)*

```
% D: POST-PROCESS
%-----------------------------------------------------------------

% 1. Strains and stresses
epsilon=zeros(Nele,1);
for i=1:Nele
    epsilon(i)=(un(i+1)-un(i))/h;
end
sigma=E*epsilon;

% 2. Analytical solution
dx=20;
xc=linspace(0,L,dx);
dq=qf-q0;
r=dq/L;
uc=1/(E*A)*((P+q0*L+1/2*r*L^2)*xc-(1/2*q0*xc.^2+1/6*r*xc.^3));
sigmac=1/A*(P+q0*(L-xc)+1/2*r*(L^2-xc.^2));

% 3. Graph

% 3.1. Plot the displacements along the bar (X)
figure
xn=linspace(0,L,Nele+1);
plot(xn,un,'-+',xc,uc,'-','LineWidth',2,'MarkerSize',10);
% hold
title(['Study of a 1D elastic fibre, Nmat=' num2str(Nmat)]);
legend({'FE Solution','Analytical solution'},'Location','Northwest');
xlabel('Coordinate x (mm)');
ylabel('Displacement u (mm)');

% 3.2. Plot the stress along the bar (x)
figure
xe=linspace(h/2,Nele*h-h/2,Nele);
plot(xe,sigma,'o',xc,sigmac,'-','LineWidth',2,'MarkerSize',10);
hold;
xs=linspace(0,L,Nele+1);
stairs(xs,[sigma;sigma(Nele)],'LineWidth',2);
title(['Study of a 1D elastic fibre, Nmat=' num2str(Nmat)]);
legend({'FE Solution','Analytical solution'},'Location','Northwest');
xlabel('Coordinate x (mm)');
ylabel('Stress \sigma (N/mm^2)');

pause;
```

Figure 31 compares the displacements obtained in the FE code in `MatLab` / `Octave` and analytically (exact solution). The FE solution in displacements is obtained at discrete nodes, whilst the stresses are obtained at the integration point of each element, in this case the the middle point. For this reason we must plot the stresses at the integration points, being constant stress within the element. We use the function *stairs* to represent this in the plot.

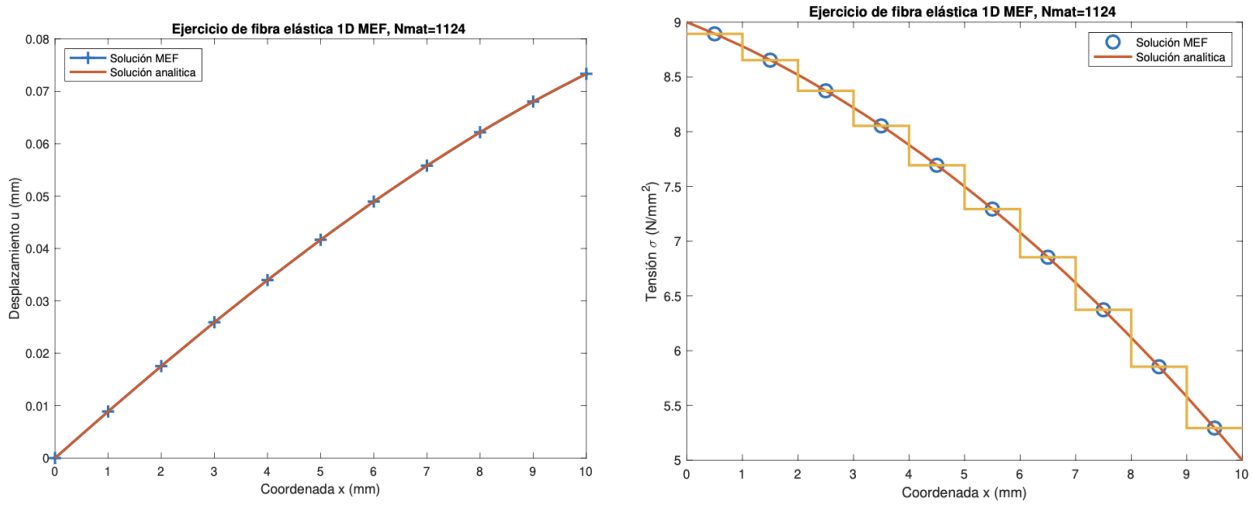


Figura 31: Comparison between the solution in MatLab and the analytical solution.

- Comparison between `MatLab` / `Octave` and the analytical solution

  We prepare the `MatLab` / `Octave` code to also plot the result that we saved from *Abaqus*:

Código 5: Scheme Matlab-Abaqus

```
% E: ABAQUS
%-----------------------------------------------------------------

xn_ab=[

    ];

un_ab=[

    ];

xe_ab=[

    ];

sigma_ab=[

    ];

% % 1. Plot displacements in x
figure
plot(xn_ab,un_ab,'-+',xc,uc,'-','LineWidth',2,'MarkerSize',10)
%hold
title(['Study of a 1D elastic fibre, Nmat=' num2str(Nmat)]);
legend({'FE Solution with Abaqus','Analytical solution'},'Location','Northwest')
xlabel('Coordinate x (mm)');
ylabel('Displacement u (mm)');

% 3.2. Plot the stress along the bar (x)
figure
plot(xe_ab,sigma_ab,'o',xc,sigmac,'-','LineWidth',2,'MarkerSize',10)
hold
xs=linspace(L,0,Nele+1);
```

```
stairs(xs,[sigma_ab;sigma_ab(Nele)],'LineWidth',2)
title(['Study of a 1D elastic fibre, Nmat=' num2str(Nmat)]);
legend({'FE Solution with Abaqus','Analytical solution'},'Location','Northwest');
xlabel('Coordinate x (mm)');
ylabel('Stress \sigma (N/mm^2)');
```

Figure 32 shows the results obtained running this code. It can be observed that the result is identical with the FE code in Matlab (figura 31) and that in Python, and they coincide with the exact analytical value at the nodes for displacements, and at the middle of the elements for the stresses.

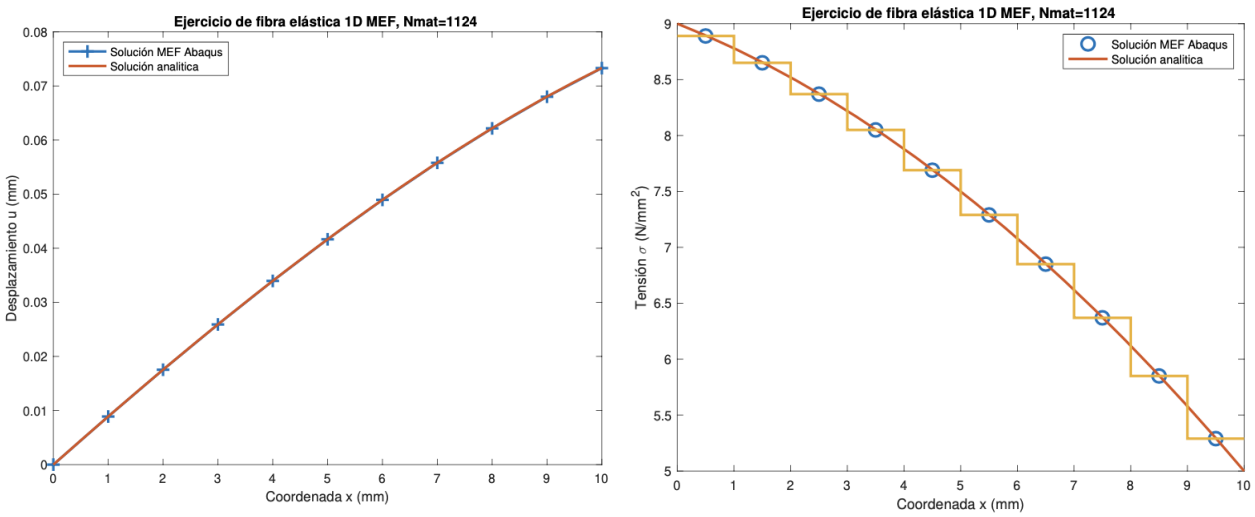


Figura 32: Comparison between the analytical solution and the FE solution in *Abaqus*.