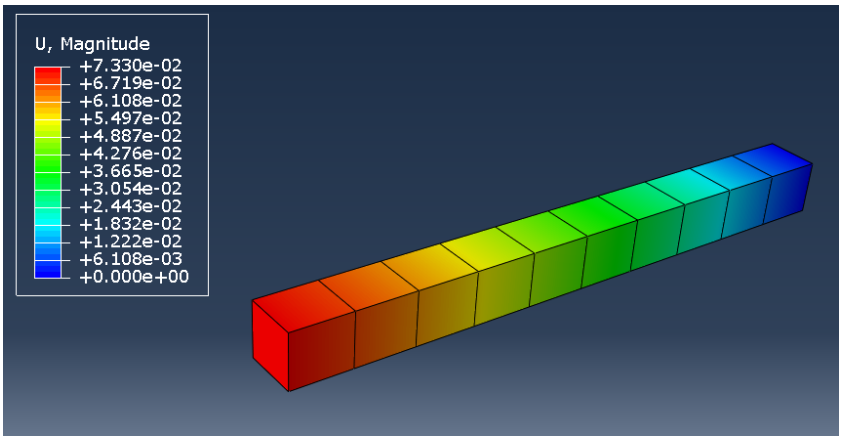
Métodos Computacionales en Ingeniería Civil

Práctica #3, Tema 1.
Modelo de elementos finitos para una barra elástica 1D



Grupo de Mecánica Computacional, ETSICCP, UPM

15-16 de febrero de 2023

Índice

1. Objetivos de la práctica

En esta práctica se ejemplifica la programación en lenguaje `Python` de un código de elementos finitos, capaz de resolver un problema elástico en una dimensión en el que se somete a cargas una barra estructural en 1D. Se incluye también en el Apéndice este programa para ser ejecutado mediante el software `Matlab/Octave`. Partimos de las siguientes premisas:

1. En las bases teóricas tenemos una serie de nociones en cuanto a la construcción de matrices de rigidez elementales, las cuales han de ser programadas por el alumnado.

2. A su vez, estas matrices elementales han de pasar a formar parte de un sistema global a través del ensamblado de la matriz de rigidez global, para la cual también se dispone de ciertas nociones teóricas.

3. Para la resolución del sistema de ecuaciones es necesario conocer las condiciones de contorno de carga y desplazamiento del problema. Se han de aplicar las mismas al sistema matricial que se ha desarrollado en lenguaje `Python`.

A su vez, vamos a simular una barra similar a la resuelta en `Python` en 1D con la suite `Abaqus`. Esto nos ayudará a asimilar los conceptos básicos de dicha suite en 3D, tanto el preproceso, cálculo y postproceso, que nos ayudarán en las siguientes prácticas. De los resultados pretendemos la comparación con una solución analítica.

Los resultados obtenidos por los dos métodos numéricos explicados, `Python` y `Abaqus`, se pueden comparar con solución analítica que el propio alumno ha de obtener. Dicha comparación ha de ser llevada a cabo con las herramientas de dibujo disponibles.

2. Base teórica y discretización con Elementos Finitos

2.1. Ecuaciones de gobierno

La ecuación que gobierna el fenómeno físico es la de una barra elástica cuyo desplazamiento se define por $u(x)$. Como se ha descrito, esta es una ecuación elíptica lineal en una dimensión, con la siguiente forma:

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

donde E es el módulo de Young, un parámetro elástico del material que describe su rigidez. $q(x)$ representa las fuerzas disribuidas para este problema, por unidad de longitud x. La resolución numérica de este problema supone encontrar un $u(x)$, en el intervalo abierto $(0, L)$, tal que el sistema esté en equilibrio teniendo en cuenta unas condiciones de contorno definidas para el problema en cuestión.

2.2. Condiciones de cortorno y fuerzas volumétricas

Las posibles condiciones de contorno que nos encontramos en el problema a resolver se resumen en la Fig. 1.

Sobre las condiciones de contorno, nos podemos encontrar de dos tipos:

- *Tipo Dirichlet o Esenciales*
  Aquellas que se aplican en el campo en el que hemos definido la ecuación diferencial, en nuestro caso $u(x)$. En este caso son condiciones de contorno "en desplazamiento".

$$u(0) = u_0$$

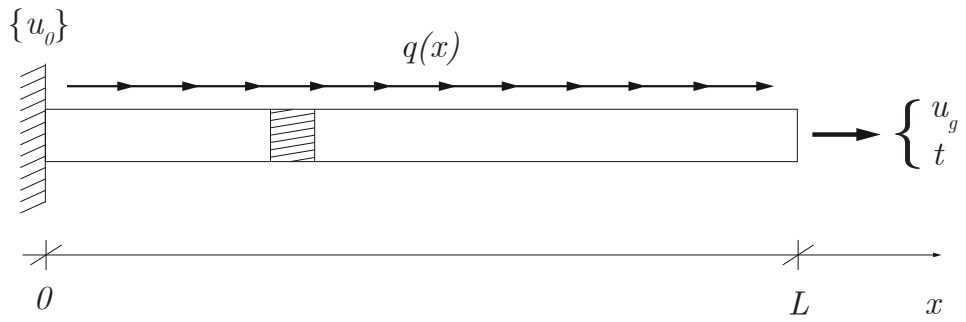


Figura 1: Definición del problema y sus condiciones de contorno.

$$u(L) = u_g$$

- *Tipo Neumann o Naturales*
  Aquellas que se aplican en la derivada espacial del campo en el que hemos definido la ecuación diferencial, que teniendo en cuenta la ecuación constitutiva, sería:

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

  Estas podrían considerarse como condiciones de contorno "en fuerzas".

Por otro lado nos encontramos las *fuerzas volumétricas o distribuidas*, aquellas que se aplican a todo el dominio y que en la ecuación (1) están definidas por $q(x)$. Estas fuerzas pueden depender de la posición o bien ser constantes. Un ejemplo de este tipo de fuerzas en el problema mecánico son aquellas asociadas a la gravedad g, las cuales dependen de una sección $A(x)$ y una densidad que puede variar a lo largo del dominio, por lo que dicha fuerza también variaría con la posición:

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

En el ejemplo a resolver en esta práctica, $q(x)$ será una función dada dependiente de la posición x, que representa la carga aplicada por unidad de longitud.

2.3. Formulación fuerte y solución analítica

La ecuación (1) se puede considerar como la formulación fuerte del problema, que se denomina así porque se obliga a que la ecuación en derivadas parciales se cumpla en cada punto del intervalo de interés, $(0, L)$.

Como vemos, dicha ecuación involucra una derivada segunda del campo $u(x)$, lo que implica que dicho campo ha de ser derivable dos veces con respecto a x.

A continuación se detalla la obtención de la solución analítica, para la cual supondremos que el área A y el módulo de Young E son constantes para toda la barra. Partimos de la integración de la formulación fuerte entre un punto $y \in (0, L)$ y L:

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

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

Si empleamos la definición que nos aporta la ecuación constitutiva, $\sigma = E\,\mathrm{d}u/\mathrm{d}x$:

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

De nuevo, integramos entre 0 y un punto $z \in (0, L)$:

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

Suponiendo conocida la derivada de u con respecto a x en el punto $x = L$, el resultado obtenido de la integral es:

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

Por tanto, la solución de $u(z)$ resulta:

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

La solución analítica de la ecuación (7) depende de dos constantes para las que son necesarias la aplicación de las dos condiciones de contorno que tenemos, $u(0)$ y $\mathrm{d}u/\mathrm{d}x|_{x=L}$, condiciones de contorno esencial y natural respectivamente. Para el caso de no tener una condición de contorno natural en el extremo $x = L$ sino una condición en desplazamientos, el término $\mathrm{d}u/\mathrm{d}x|_{x=L}$ se puede calcular imponiendo $z = L$, siendo que $u(L)$ es conocido.

Para el caso particular en que las condiciones de contorno sean: $u(0) = 0$ (extremo fijo) y $EA\mathrm{d}u/\mathrm{d}x|_L = P$ (carga P en $x = L$), y la fuerza distribuida sea una función lineal $q(x) = q_0 + rx$, la expresión de la solución analítica es:

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

Teniendo en cuenta las hipótesis realizadas, solo podremos obtener expresiones analíticas para valores sencillos de $E(x)$ y $q(x)$, abandonando la idea cuando estos valores se complican, por lo que hemos de buscar soluciones aproximadas. Una de las técnicas más empleadas es la del método de los elementos finitos, para el que necesitamos la forma débil del problema, que pasamos a describir en la siguiente sección.

2.4. Formulación débil

Esta formulación se consigue con una función de ponderación arbitraria $w(x)$, por la que se multiplica la ecuación de la formulación fuerte y se integra sobre todo el dominio:

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

Integrando por partes el primer término:

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

obtenemos el resultado siguiente sustituyendo en la ecuación (9):

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

donde t_L y t_0 son las condiciones de contorno naturales en los extremos, que físicamente corresponden a las fuerzas aplicadas en dichos extremos. Si en el extremo $x = 0$ se aplica una condición (esencial) de desplazamiento impuesto o fijo, en este caso a la función de ponderación se le exige también $w(0) = 0$, por lo que el último sumando de la ecuación (11) desaparece.

2.5. Funciones de forma

En el Método de los elementos finitos es necesario realizar la aproximación de la incógnita $u(x)$ por funciones de aproximación, también llamadas funciones de forma, a través de la expresión:

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

Se emplea un conjunto finito (N_{nod}) de funciones de interpolación $N_B(x)$. Esto conlleva un error que en principio disminuye cuanto mayor sea el número de nodos N_{nod} o el orden de las funciones de interpolación.

El Método de Galerkin, que es el método más empleado y, por tanto, el que vamos a utilizar, emplea para las funciones de ponderación $w(x)$ la misma interpolación que para la función incógnita $u(x)$:

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

La interpolación de las derivadas que aparecen en la forma débil será

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

Sustituyendo en las integrales de la forma débil (Eq. (11)), obtenemos:

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

donde

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

es decir, la denominada matriz de rigidez, $[\boldsymbol{K}]$. Análogamente, para las acciones aplicadas o fuentes se obtienen los términos del vector de fuerzas:

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

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

De las ecuaciones (15), (16) y (17) resulta el siguiente sistema de ecuaciones algebraicas lineales:

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

donde las fuerzas aplicadas se obtienen como suma de las volumétricas (interiores distribuidas) y las exteriores en el contorno

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

Y teniendo en cuenta que $w(x)$ son arbitrarias, w_A también lo serán, por lo que se obtiene la ecuación matricial:

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

2.6. Funciones de forma elementales

En la práctica el cálculo de las integrales y la expresión de las funciones de interpolación se hacen elemento a elemento. Esto facilita sobremanera el cálculo que se hace con el mismo algoritmo en cada uno de los elementos, para luego ensamblar las matrices globales. Las funciones de interpolación tienen soporte compacto, es decir son cero fuera del subdominio Ω^e correspondiente al elemento (e) en cuestión. Se establece una numeración y coordenadas locales en cada elemento, que tienen su correspondencia con las globales.

En el caso a resolver 1D podemos suponer funciones de forma lineales de dos nodos como los de la figura 2.

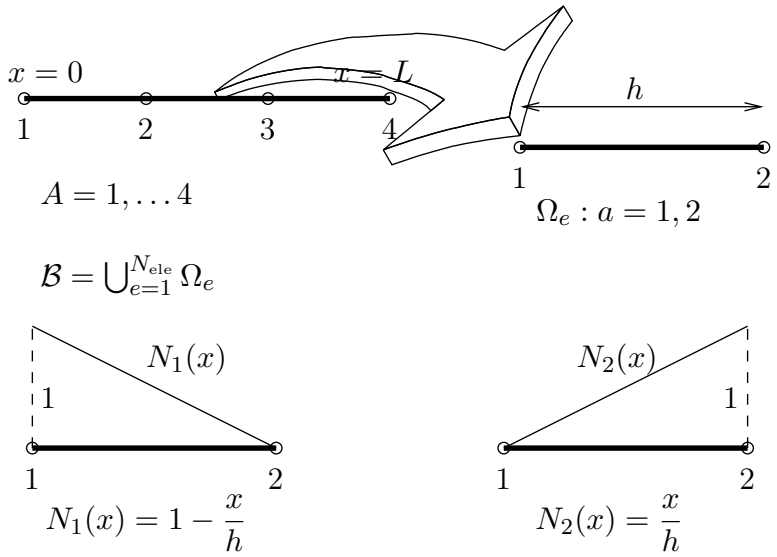


Figura 2: Esquema de las funciones de forma lineales del elemento 2 de una discretización 1D de una barra con 3 elementos

Teniendo en cuenta los valores de N_1 y N_2 que aparecen en la figura 2, se realizan las integrales de cada uno de los términos; por ejemplo K_{11}^e es:

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

y análogamente los demás:

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

resultando

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

Similarmente el vector de fuerzas volumétricas del elemento resulta

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

Una vez formadas las matrices elementales se convierten a numeración global,

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

ensamblándose finalmente estas matrices locales dentro de las matrices globales del sistema. Para el ejemplo de la figura 2, puesto que todos los elementos son iguales, las matrices elementales de rigidez son todas iguales entre sí, siendo su valor el de la matriz de la ecuación (22).

Para poder realizar el ensamblaje hay que tener en cuenta el lugar que van a ocupar en la matriz de rigidez global las distintas componentes de las matrices de rigidez locales. Si observamos la figura 2 vemos que el elemento 1 y el elemento 2 comparten el nodo número 2, por eso la posición $(2,2)$ de la matriz de rigidez global será compartida por las matrices de rigidez locales de los elementos 1 y 2. La forma que tendrá esta matriz de rigidez global será:

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

De este ensamblaje, teniendo en cuenta los valores de las matrices de rigidez locales, obtenemos la matriz de rigidez global:

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

Las matrices elementales y global de fuerzas volumétricas son

$$\{\mathbf{f}^{\text{vol},e}\} = \frac{h}{6}\begin{Bmatrix} 2q_1 + q_2 \\ q_1 + 2q_2 \end{Bmatrix} \Rightarrow \{\mathrm{f}^{\text{vol}}\} = \frac{h}{6}\begin{Bmatrix} 2q^{(1)}_1 + q^{(1)}_2 \\ q^{(1)}_1 + 2q^{(1)}_2 + 2q^{(2)}_1 + q^{(2)}_2 \\ q^{(2)}_1 + 2q^{(2)}_2 + 2q^{(3)}_1 + q^{(3)}_2 \\ q^{(3)}_1 + 2q^{(3)}_2 \end{Bmatrix}$$

Si considerasemos una restricción del movimiento en el extremo $x = 0$ y una carga q_L aplicada en el extremo $x = L$, la ecuación matricial que resulta de aplicar el método de los Elementos Finitos es:

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

$$\frac{EA}{h}\left(\begin{array}{c|ccc} 1 & -1 & 0 & 0 \\ \hline -1 & 1+1 & -1 & 0 \\ 0 & -1 & 1+1 & -1 \\ 0 & 0 & -1 & 1 \end{array}\right)\left\{\begin{array}{c} 0 \\ \hline u_2 \\ u_3 \\ u_4 \end{array}\right\} = \frac{h}{6}\left\{\begin{array}{c} 2q^{(1)}_1 + q^{(1)}_2 \\ \hline q^{(1)}_1 + 2q^{(1)}_2 + 2q^{(2)}_1 + q^{(2)}_2 \\ q^{(2)}_1 + 2q^{(2)}_2 + 2q^{(3)}_1 + q^{(3)}_2 \\ q^{(3)}_1 + 2q^{(3)}_2 \end{array}\right\} + \left\{\begin{array}{c} R_0 \\ \hline 0 \\ 0 \\ P \end{array}\right\} \tag{25}$$

Al vector de fuerzas volumétricas se le suma el de externas, en este caso una posible carga en P. Por otro lado, en el extremo $x = 0$ existe una reacción para que el desplazamiento sea igual al impuesto, $u(0) = 0$. Dicha reacción se calcula *a posteriori,* una vez calculados los desplazamientos $\{\mathbf{u}\}$.

Una vez calculados los desplazamientos por la inversión de la matriz de rigidez, ya reducida por la aplicación de las condiciones de contorno, cabe la posibilidad de verificar el equilibrio, que se puede obtener como resultado de la suma de las fuerzas internas del cuerpo, que, en el caso del equilibrio, deben ser 0, siendo estas fuerzas internas calculadas como:

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

Por otro lado, la reacciñon en este caso se podría calcular multiplicando la fila eliminada de la matriz $[\mathbf{K}]$ por el vector de desplazamientos y restándole las fuerzas aplicadas en dicho nodo, que, al tratarse de un contorno tipo Dirichlet, solo podrán ser fuerzas volumétricas:

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

Observación importante.— El hecho de que los valores en los puntos discretos, tanto en tensiones como en desplazamientos, vayan a salir exactamente iguales que los de la solución analítica, es un caso que se denomina *"superconvergente"*. Sin embargo, en general las soluciones numéricas en los puntos discretos no coincidirán con la solución exacta. Dicho de otra manera, en este caso extremadamente sencillo el único error cometido por la solución de elementos finitos es para los valores intermedios en el interior de los elementos, siendo exactos los valores en los nodos y en los puntos de integración; sin embargo, en un caso general, habrá un error numérico también para los valores en los nodos y en los puntos de integración.

3. Simulación con Abaqus

Se desea calcular la respuesta de una barra elástica unidimensional, de longitud $L = 10\text{mm}$ y sección uniforme $A = 1\text{mm}^2$. El material es elástico lineal, con módulo de Young $E = 1000\text{MPa}$. El extremo izquierdo ($x = 0$) está fijo mientras que sobre el derecho ($x = L$) actúa una fuerza axial de valor $P = 5\text{N}$, como se indica en la figura 3. Se podrá suponer que las deformaciones son pequeñas.

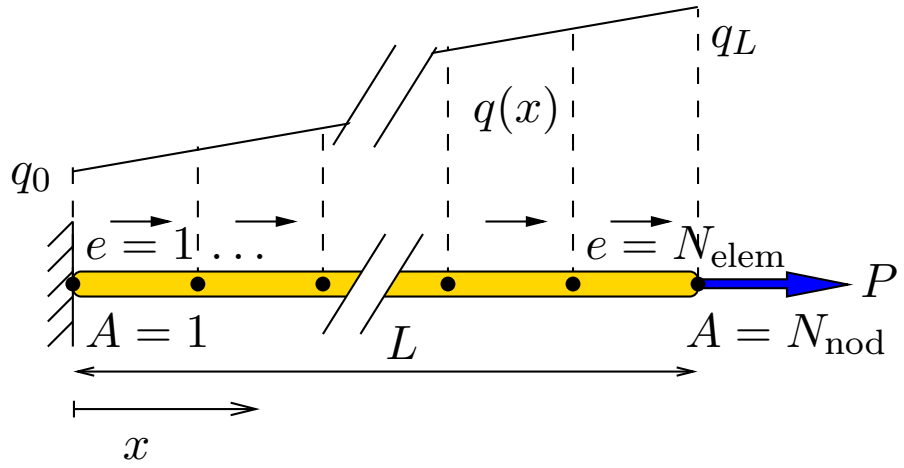


Figura 3: Modelo de fibra elástica 1D

A partir de los dígitos de las centenas (c), decenas (d) y unidades (u) del número de matricula de cada estudiante, $N_{\text{mat}} = mcdu$, se tomarán los siguientes parámetros para el modelo:

- Número de elementos $N_{\text{elem}} = 11 - c$
- La carga distribuida $q(x)$, fuerza longitudinal por unidad de longitud, será lineal entre los extremos $x = 0$ y $x = L$, con los valores en los extremos $q_0 = d/10$ N/mm, $q_L = q_0 + u/10$ N/mm.

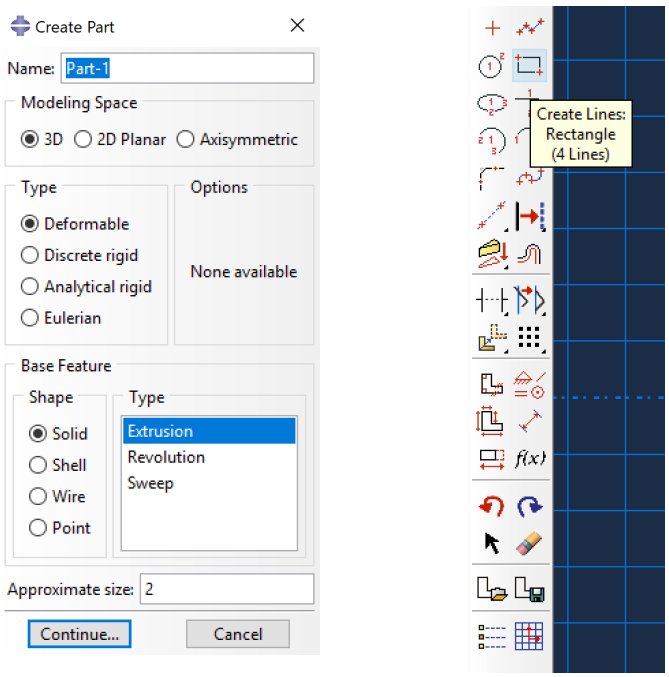
Los pasos que se deberían seguir en una programación del mismo serían:

1. Planteamiento del problema elástico y condiciones de contorno.
2. Expresiones numéricas de las matrices de rigidez y de fuerzas del modelo completo.
3. Aplicación de condiciones de contorno y resolución.
4. Solución para los desplazamientos $u(x)$ y tensiones $\sigma(x)$ en cada punto, dibujando la gráfica en función de x y comparando con la solución analítica. (Nota: las tensiones se calculan a partir de los desplazamientos como $\sigma = E\varepsilon = E\mathrm{d}u/\mathrm{d}x$, y empleando la aproximación en un elemento finito, $\sigma^h = Eu_a \mathrm{d}N_a/\mathrm{d}x = E\left(u_2 - u_1\right)/h$.)

En un primer momento vamos a emplear el programa `Abaqus` para obtener el resultado de dicha barra empleando un software comercial de *elementos finitos*. Aunque el modelo es unidimensional, decidimos emplear elementos 3D para ayudar a visualizar la geometría de la barra e introducir este tipo de modelado. Seguiremos los pasos similares a la práctica 1, solo que en este caso vamos a realizar nuestro primer modelo 3D.

3.1. Módulo **Part**

En primer lugar, se ejecuta *Abaqus CAE* para crear un modelo nuevo. Se entra en el módulo `part`, activando el icono de crear una nueva parte, que se define como 3D, deformable, solid por extrusión (figura 4a). Una vez definido el problema, creamos un rectángulo (figura 4b) en las esquinas (-0.5,-0.5) y (0.5,0.5), por lo que tendremos un area de 1 mm^2.



(a) Part/solid (b) Rectángulo

Figura 4: Creación de geometría

Una vez definido, hacemos clic en *done* y pasamos a definir la longitud de la barra en la dirección Z, que en nuestro caso es 10 mm. Deberíamos obtener un cuerpo similar al del la figura 6.

3.2. Módulo **Property**

Se activa el icono para crear material nuevo,(figura 7), se selecciona el material elástico lineal y se introducen las propiedades $E = 1000\,\text{MPa}, \nu = 0$. Hemos de tener en cuenta que al trabajar en mm y N para la carga, debemos introducir el módulo elástico en MPa=N/mm^2. A continuación se crea una *"sección"*, en la que se definen sus propiedades, en este caso solo el tipo de material descrito anteriormente, tipo *solid*-homogéneo (figura 8a). A continuación se asigna la sección a la parte creada (figura 8b), seleccionando todo el volumen y haciendo clic en *Done*.

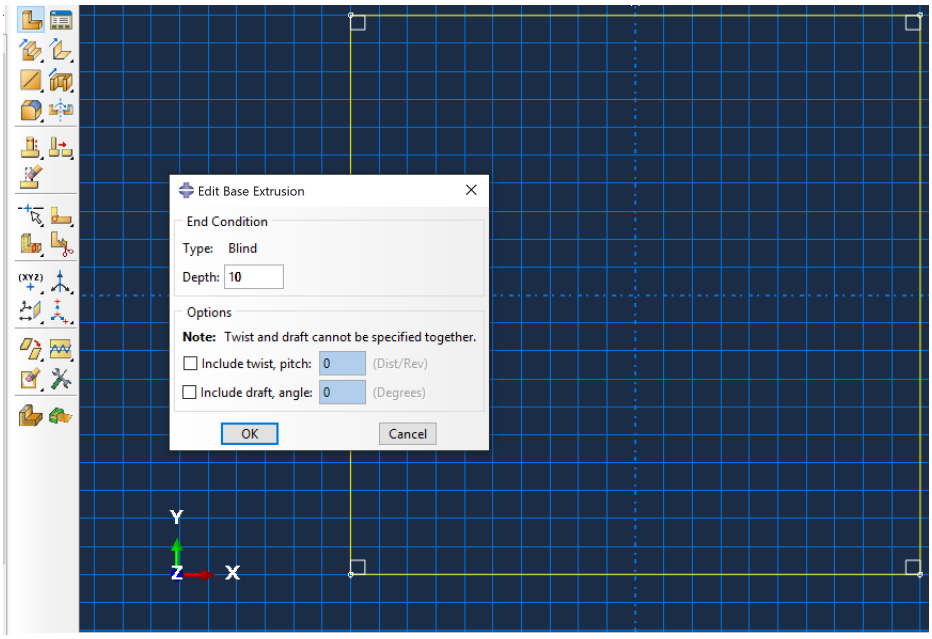


Figura 5: Extrusión de la sección.

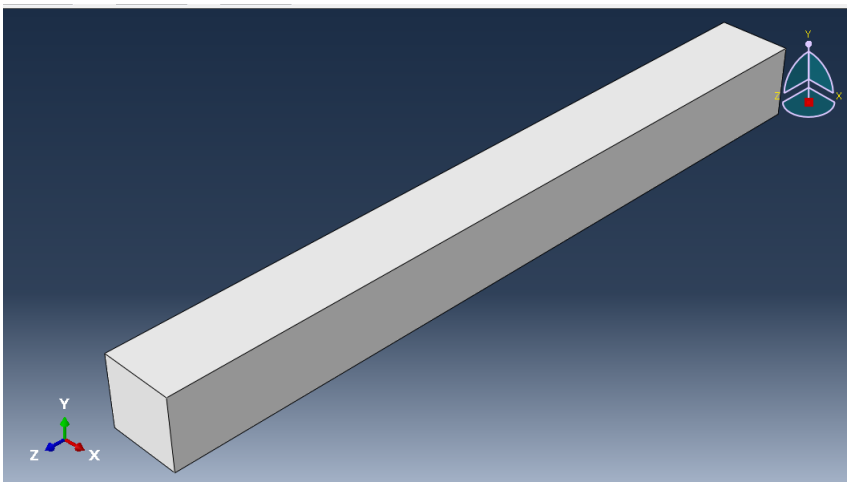


Figura 6: Barra final

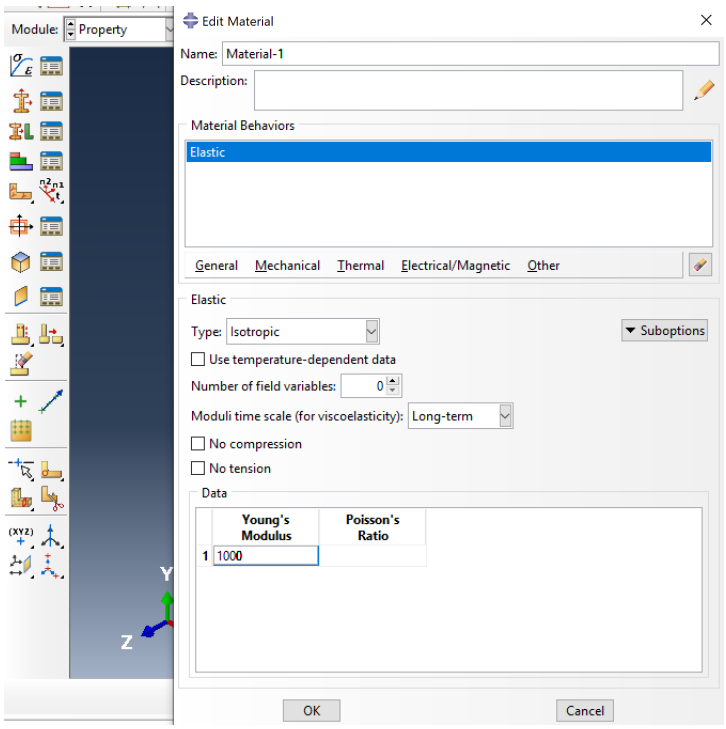
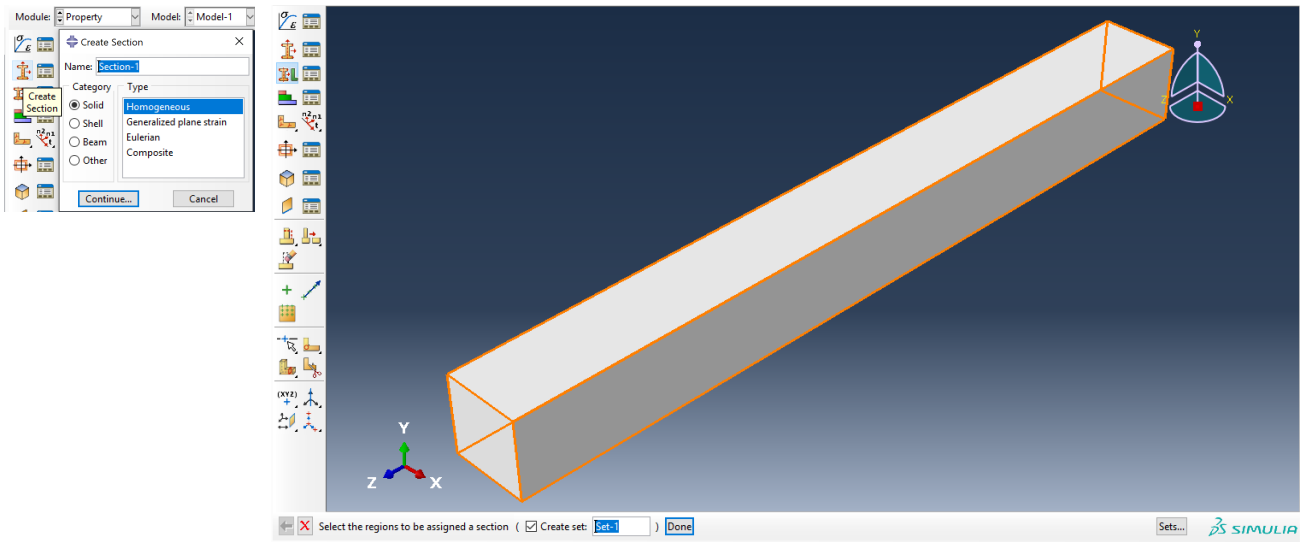


Figura 7: Material elástico.



(a) Crear sección (b) Asignar sección

Figura 8: Selección de la sección

3.3. Módulo **Assembly**

En este módulo tan solo hay que crear una "instancia" a partir de la parte, mediante las opciones por defecto:

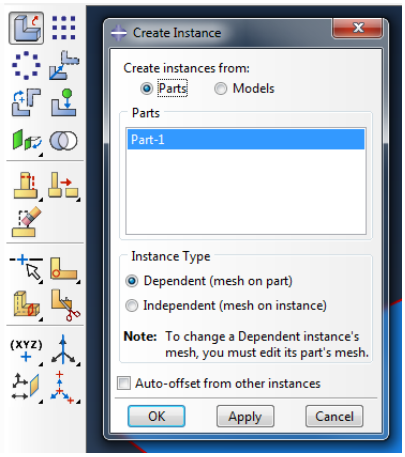


Figura 9: Assembly: crea una instancia de la parte

3.4. Módulo **Step**

Se crea un "step" para el procedimiento de cálculo "Static, general". Se toman las opciones por defecto (figura 10). No es necesario editar el "field output", pues el conjunto de variables de salida por defecto incluye ya las que necesitaremos en nuestro análisis.

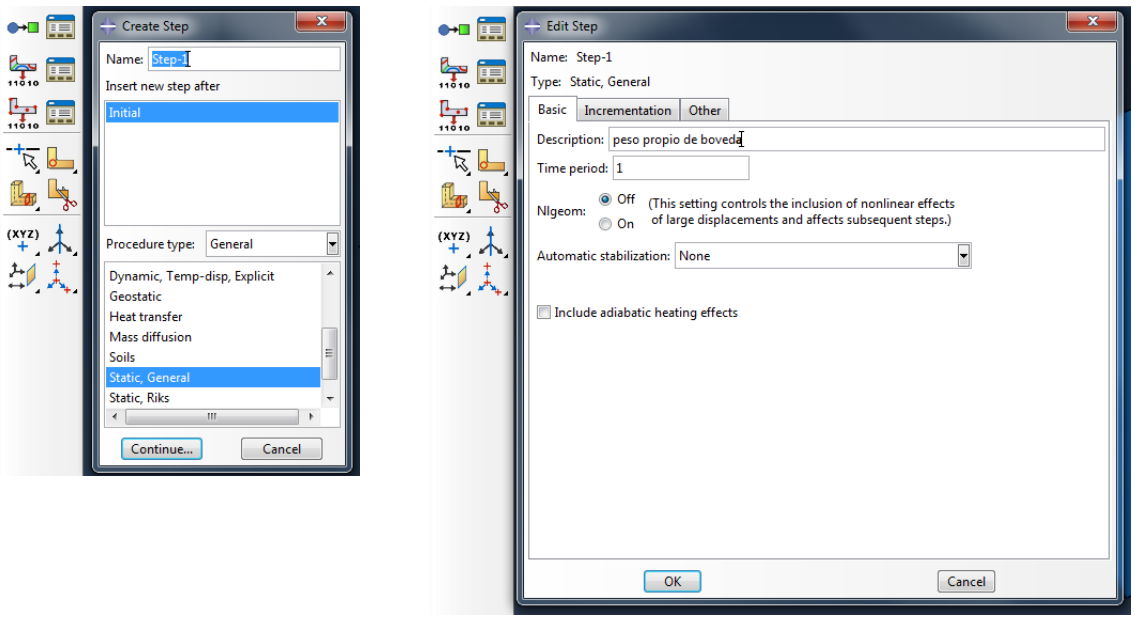


Figura 10: Crear y definir el "step"

3.5. Módulo **Load**

Debemos definir en este módulo ambos tipos de condiciones de contorno. En primer lugar, en *Z=0*, empotramos la barra. Para ello creamos *Boundary Condition*, de categoría *Mechanical* y tipo *Encastre* (figura **??**).

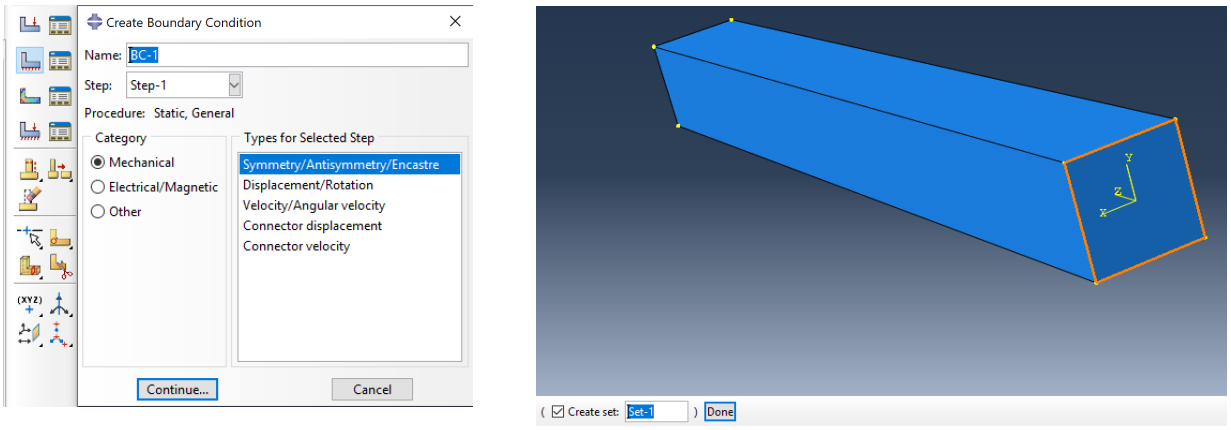


Figura 11: Creación del empotramiento

Una vez seleccionada la superficie a empotrar y habiendo hecho clic en *Done*, debemos especificar el tipo de condición a asignar dentro de este subgrupo, por lo que seleccionaremos la última de esta nueva pestaña (figura **??**), donde menciona la opción *Encastre*.

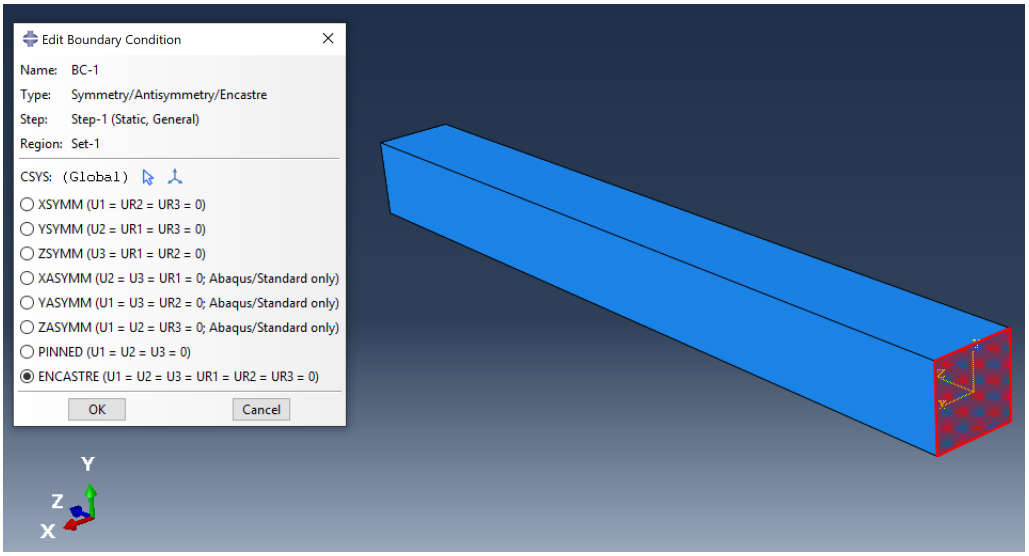
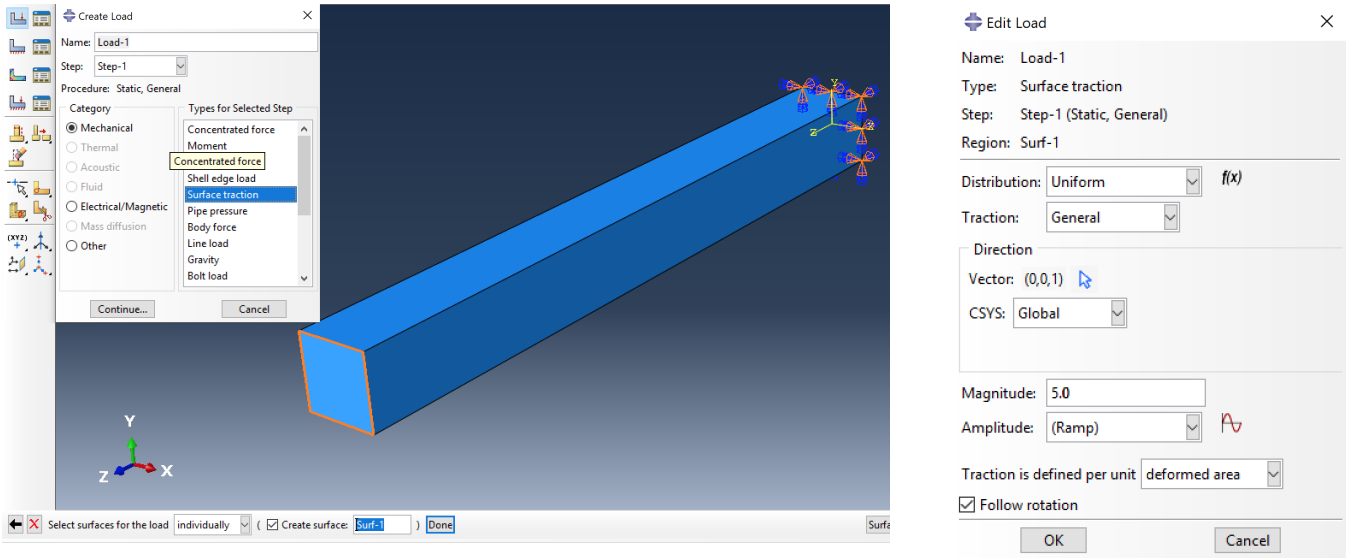


Figura 12: Tipo de *Boundary Condition: Encastre*

Pasamos entonces a definir las dos condiciones de carga que aplican en nuestro problema: carga puntual y volumétrica. Con respecto a la primera se puede realizar de diversas formas. La que consideramos más inmediata es aplicar una tracción a la superficie en *Z=L*, teniendo en cuenta que en `Abaqus` es una carga de categoría *Mechanical* y tipo *Surface Traction* (figura **??**). Definimos a continuación la superficie donde se va a aplicar y se nos abrirá la pestaña de edición de carga (figura **??**). La carga es de distribución uniforme, y la tracción de tipo *General*. Para la dirección debemos dibujar en el modelo un vector que siga la dirección creciente del eje *Z*, por lo que se nos debe escribir un vector unitario (0,0,1). La magnitud será de 5 N dividido de la superficie donde se aplica, 1 mm^2, es decir, 5 MPa.



(a) Creación de la carga y asignación (b) Definición

Figura 13: Carga puntual

En lo que respecta a la carga volumétrica repartida en todo el cuerpo se ha de seleccionar el tipo *Body Force* en el apartado de creación de cargas y posteriormente asignarsela a todo el cuerpo (figura **??**).

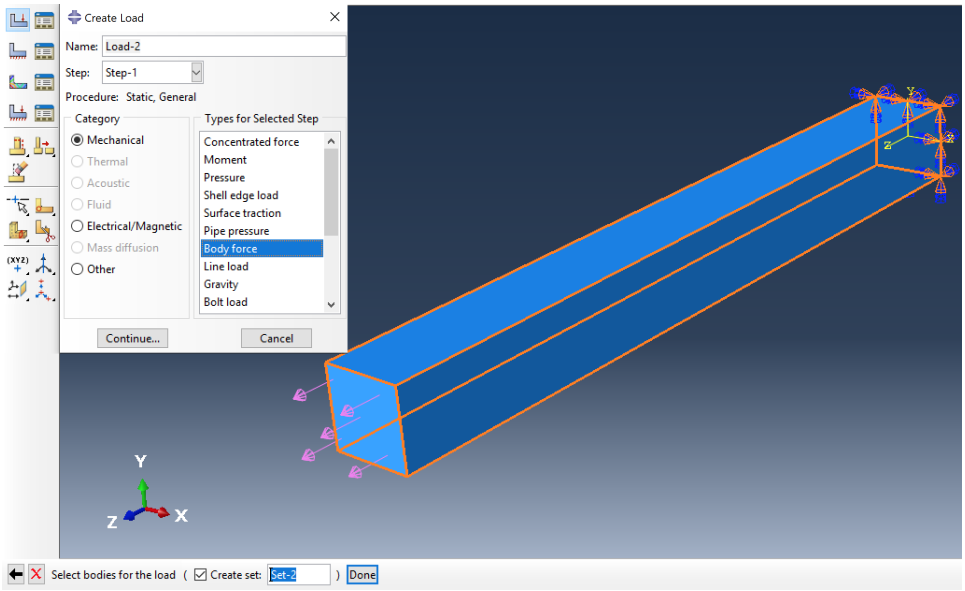


Figura 14: Creación de la carga *Body Force*

A continuación hemos de definir dicha carga. Para ello hemos de hacer clic primero en un botón con el símbolo $f(x)$, puesto que no será un valor puntual sino una función. Para la resolución de este problema optamos por asignar una expresión. Se nos abrira una ventana de creación de *Expression Field*. Teniendo en cuenta que el número de matrícula asignado en esta práctica es 1024, deberemos crear una carga que siga la ecuación $q = 0.2 + 0.04 * Z$, puesto que nuestra barra sigue dicha dirección en el modelo de **Abaqus**. Será esta la ecuación que deberemos escribir en el campo correspondiente. Una vez creado, en el campo *Distribution*, en la ventana *Edit Load*, asiganaremos el *Expression Field* creado, en nuestro caso *AnalyticalField-1*. La componente será la (0,0,0), es decir, siguiendo la dirección *Z*.

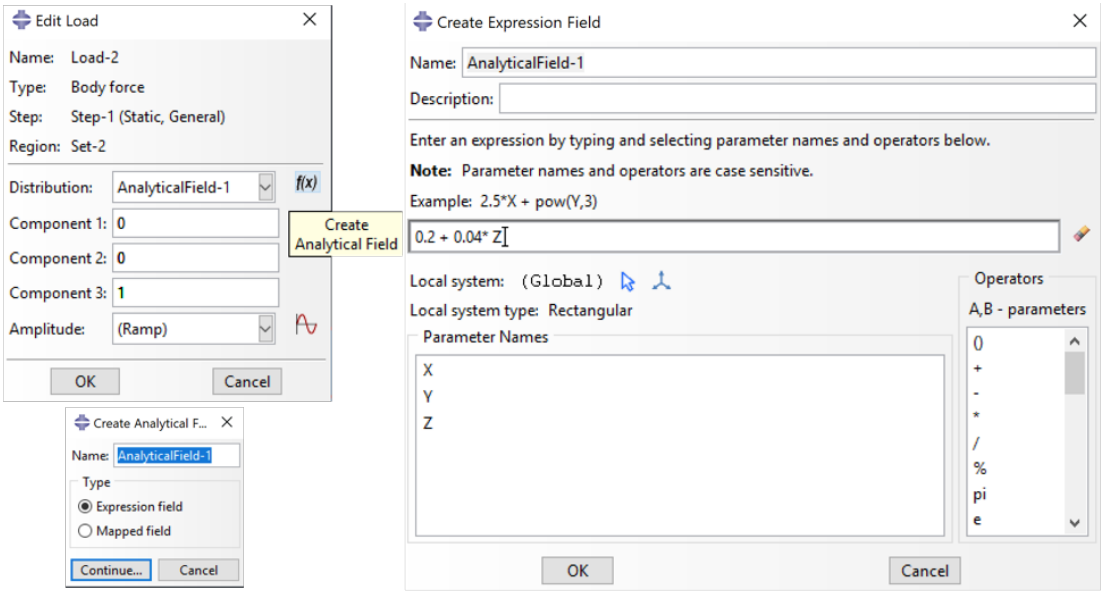


Figura 15: Definición de la distribución de carga con una expresión

Una vez definida esta segunda carga, nuestro modelo debe tener una apariencia como la que vemos a continuación:

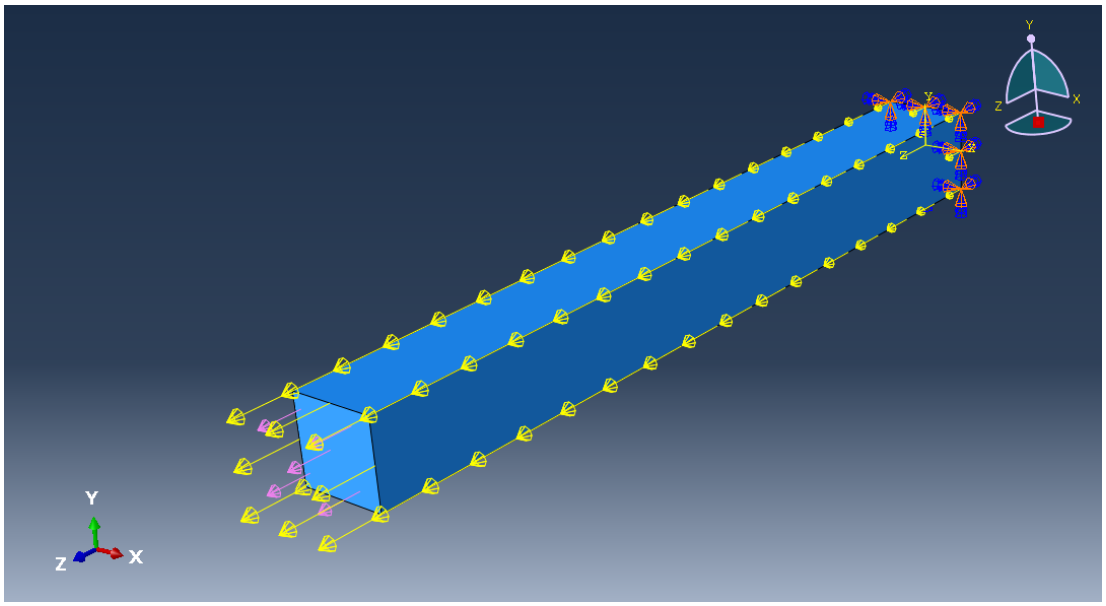


Figura 16: Distribución de cargas final

3.6. Módulo **Mesh**

En el módulo mesh hay que comenzar por seleccionar la parte expandiendo el árbol del modelo / parte de la izquierda, y activando con el botón derecho el icono *"Mesh"*: Fig. **??**. Se abre en el menú superior *Mesh* → *Controls* y se selecciona "Hex" y "Structured", para generar una malla estructurada de cuadriláteros (Fig. **??**).

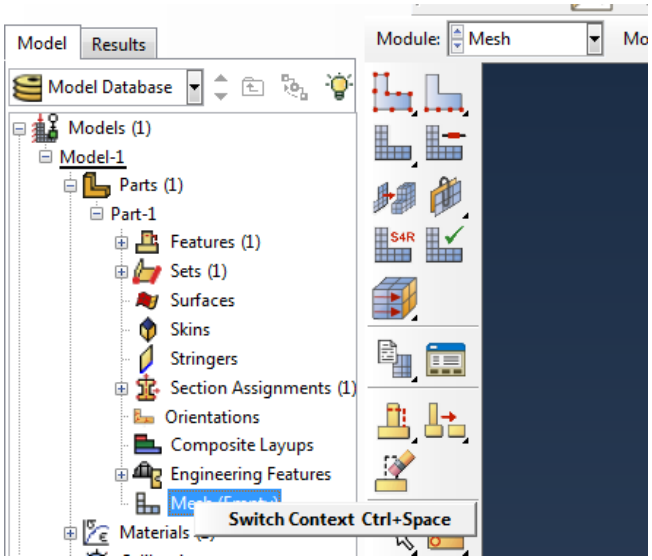


Figura 17: Seleccionar en el menú desplegado del modelo para mallar.

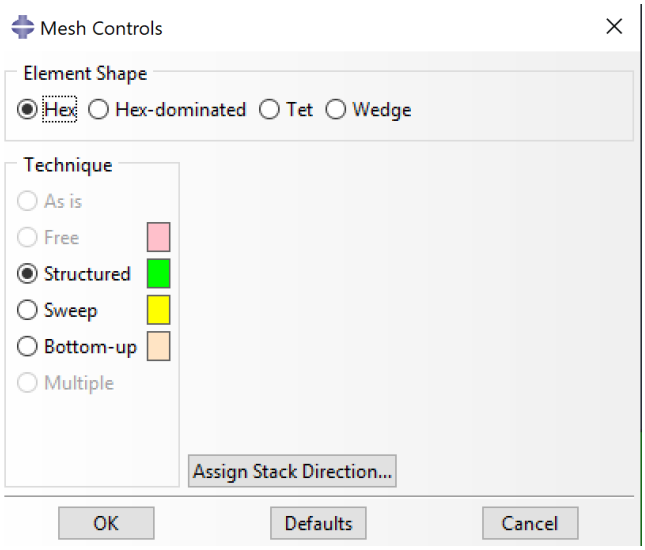


Figura 18: Definir opciones en "Mesh controls"

A continuación se abre en el menú superior *Mesh* → *Element type* y se seleccionan las opciones "3D Stress", "Linear", *"Reduced integration"* lo que dará lugar al elemento C3D8R, suficiente para el cálculo que vamos a realizar. (Fig. **??**).

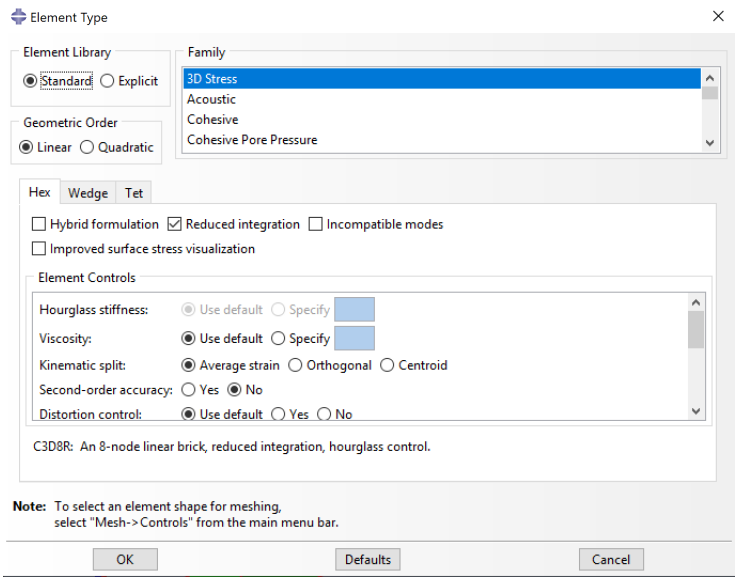


Figura 19: Seleccionar tipo de elemento en "Mesh/element"

El tamaño *"Semilla"* que vamos a emplear es de 1 mm en todas las direcciones del espacio, teniendo en cuenta queremos realizar un mallado de 10 elementos hexaedrícos en la dirección longitudinal de la barra (Fig. **??**).

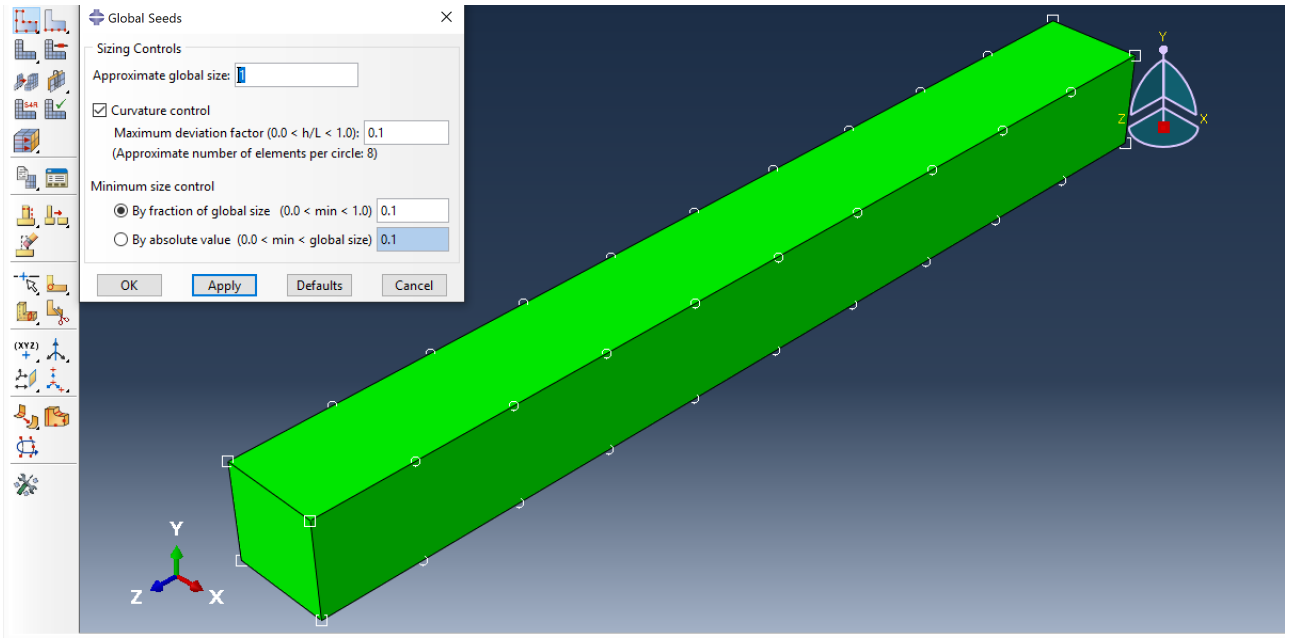


Figura 20: "Global Seeds" empleado

Por tanto, la malla que resulta es la que vemos a continuación

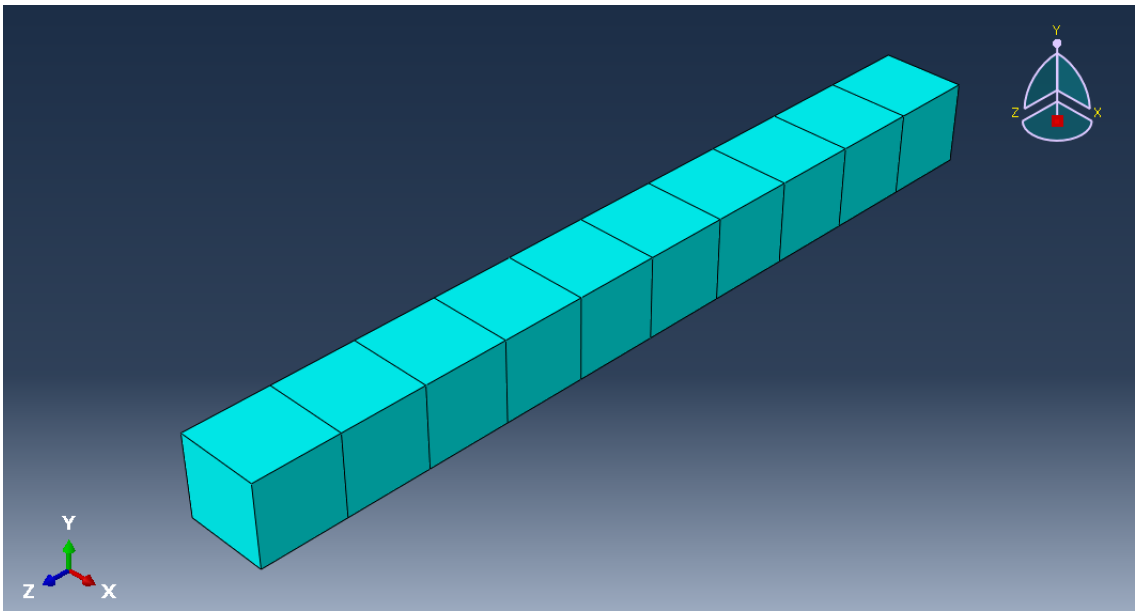


Figura 21: Malla obtenida

3.7. Módulo **Job**

Una vez completado el modelo, se crea un "Job" con las opciones por defecto (figura **??**).

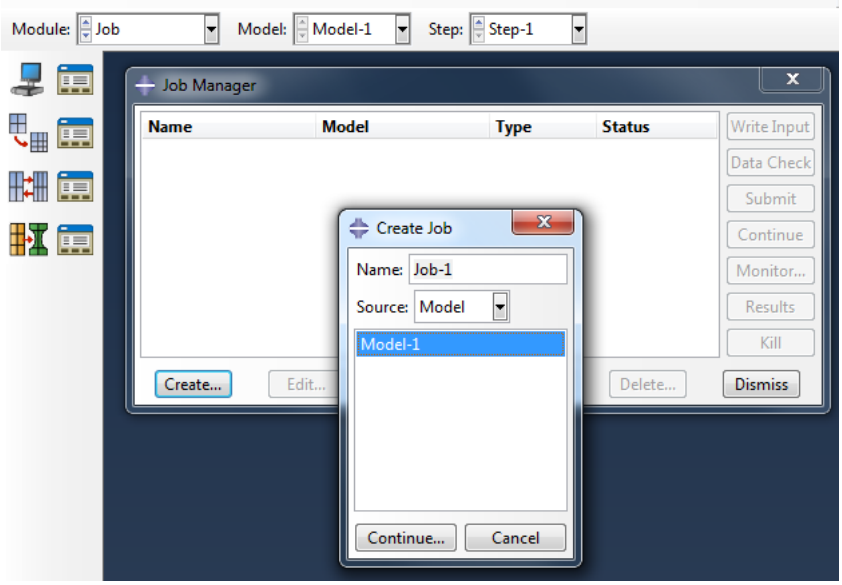


Figura 22: Creación del "Job"

Y se envía para calcular mediante "Submit". El *Status* va cambiando de "Submitted" → "Running" → "Completed". (figura **??**) Si no hay mensaje de error el problema está acabado y

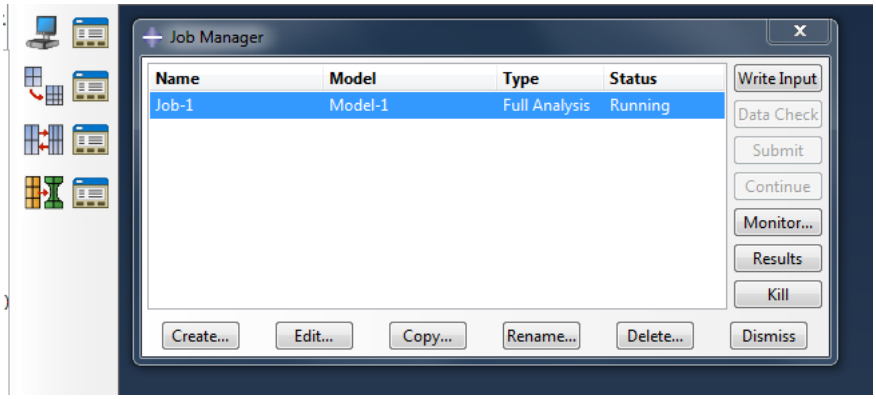


Figura 23: Envío del "Job"

se pasa al módulo de visualizar los resultados.

3.8. Módulo **Visualization**

A modo de comprobación, el primer campo a visualizar es el del desplazamiento en Z. Vemos que el desplazamiento máximo es de 0.073 mm(figura **??**). Vamos a dibujar la distribución del desplazamiento con respecto a la posición en *Z*.

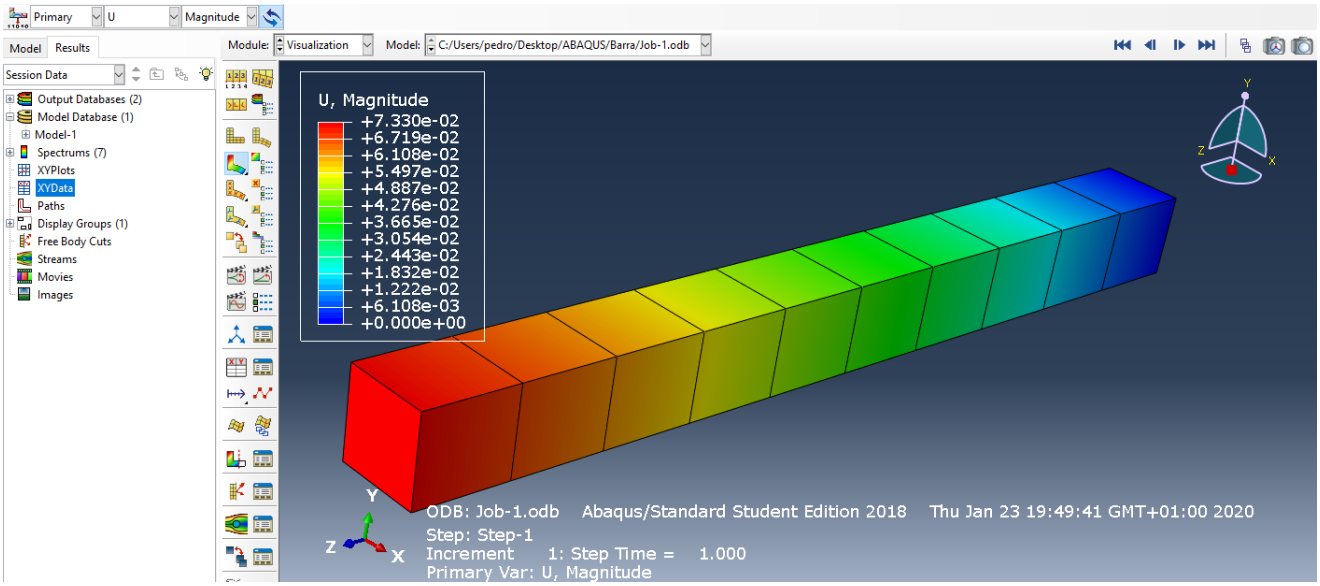


Figura 24: Campo de desplazamientos

Antes de nada hemos de indicar un *Path* que tenga nodos distribuidos en *Z*. Para ello seleccionamos path y con el boton derecho abrimos un desplegable donde debemos seleccionar *create* (figura **??**). Nuestro tipo de path será creado seleccionándolos de una lista de nodos. Nuestro modo de selección será según avanzamos en el eje Z, por lo que seleccionamos *Add After* y debemos cuidar de escoger primero un nodo en $Z = 0$ y el siguiente nodo en $Z = L$. Podemos elegir cualquier arista, pues todas poseen la misma información. Como queda este path definido puede verse en la figura **??**.

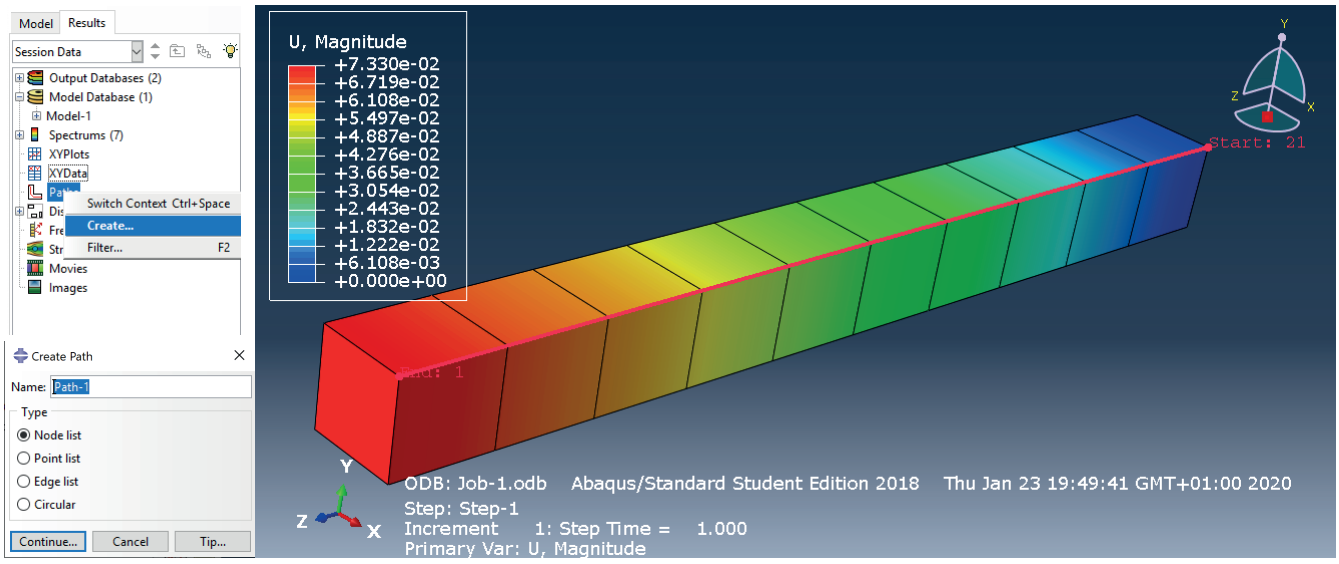


Figura 25: Definición de un path

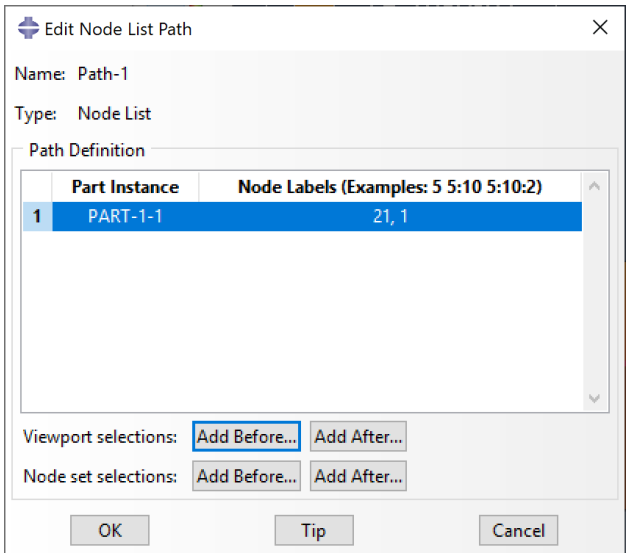


Figura 26: Lista de nodos de un path

Ya es posible crear un gráfico del desplazamiento en Z a lo largo de la barra. Para ello, necesitamos crear un *XY-Data*. En dicho campo, con el boton derecho se nos abre un desplegable donde, al seleccionar *Create*, podemos cread dicho *XY-Data* desde el un Path ya definido. En nuestro caso el campo ha de ser el desplazamiento en *Z, U3*. Con este campo selecionado debemos, en la ventana *XY Data from Path* (figura **??**), selecionar el Path creado anteriormente (en su configuración no deformada) y selecionar la opción de incluir las intersecciones, para añadir así los valores de los nodos intermedios. Finalmente, optamos por elegir, como valor en X de nuestro gráfico, la coordenada Z.

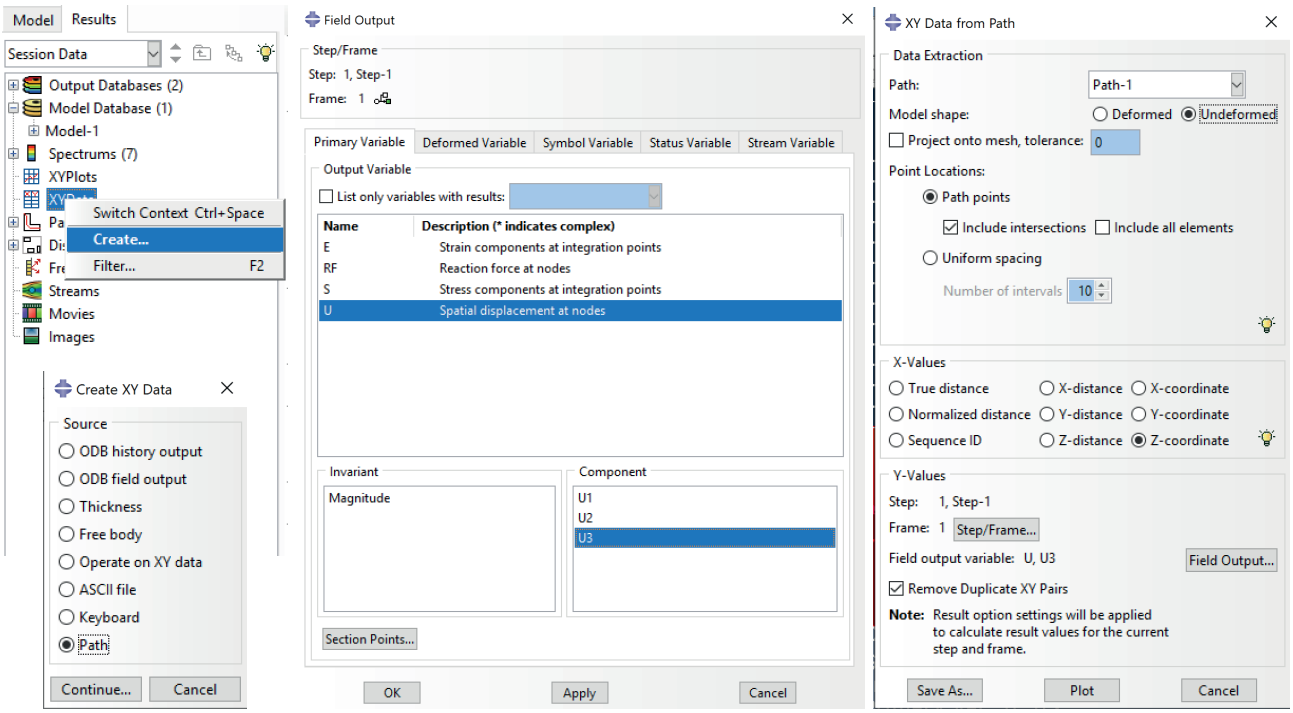


Figura 27: Creación de un XY-data

Si dibujamos dicho gráfico obtenemos una figura similar a la que se puede ver en la figura **??**. Compararemos la forma y valores de esta gráfica con la que obtendremos en en `Python` para ver la correspondencia entre ambos resultados, si bien necesitaremos superponer ambas gráficas.

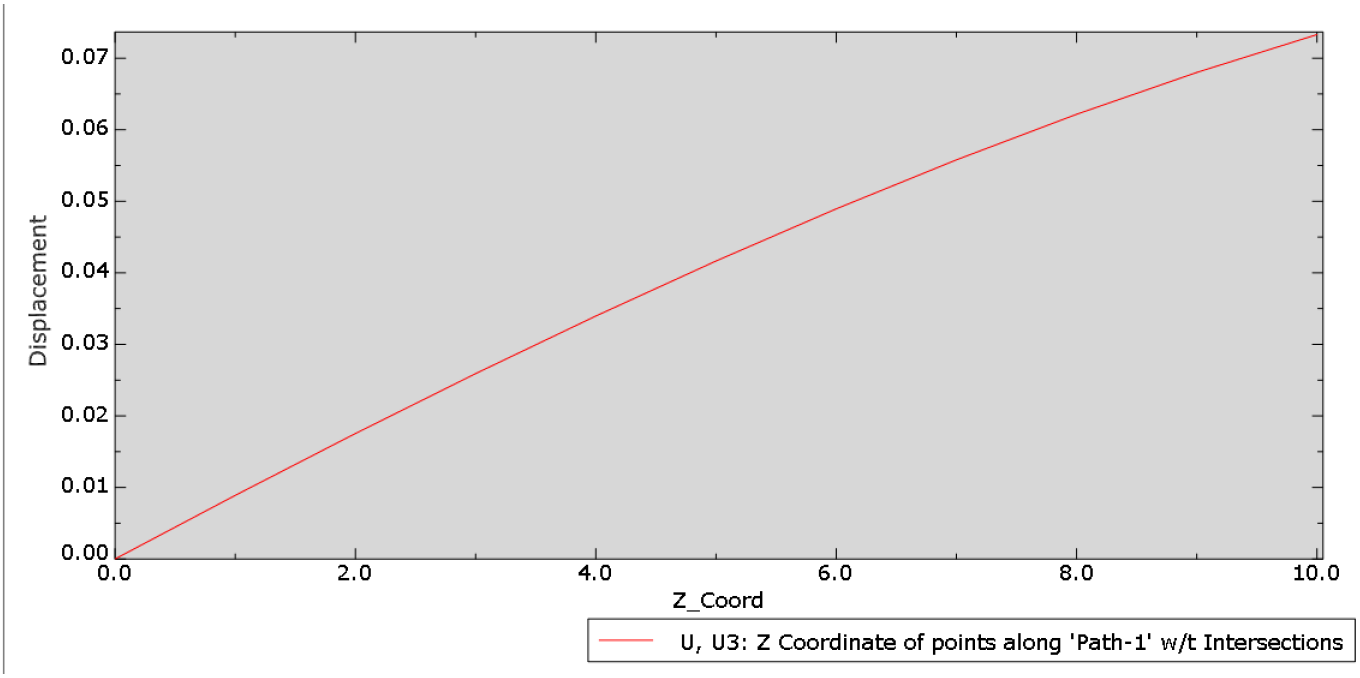


Figura 28: Desplazamiento en en eje Z

Podemos guardar estos datos, para posteriormente exportarlos a `Python` para su comparativa con la solución analítica, haciendo clic en el botón *Save as...* El archivo de las deformaciones lo denominaremos `u3.txt`. Si intentasemos cambiar el campo de dibujo al de las tensiones, en este caso S33 que es la que se corresponde con σ_z, observamos, en la figura **??**. En cambio, aunque la tendencia y valores son cercanos, en los extremos ocurren cosas raras. Esto se debe a que la tensión es un dato correspondiente a los puntos de integración, no a los nodos, que es donde se puede hacer este path. Por tanto, para la comparativa, deberemos extraer los datos de otra manera.

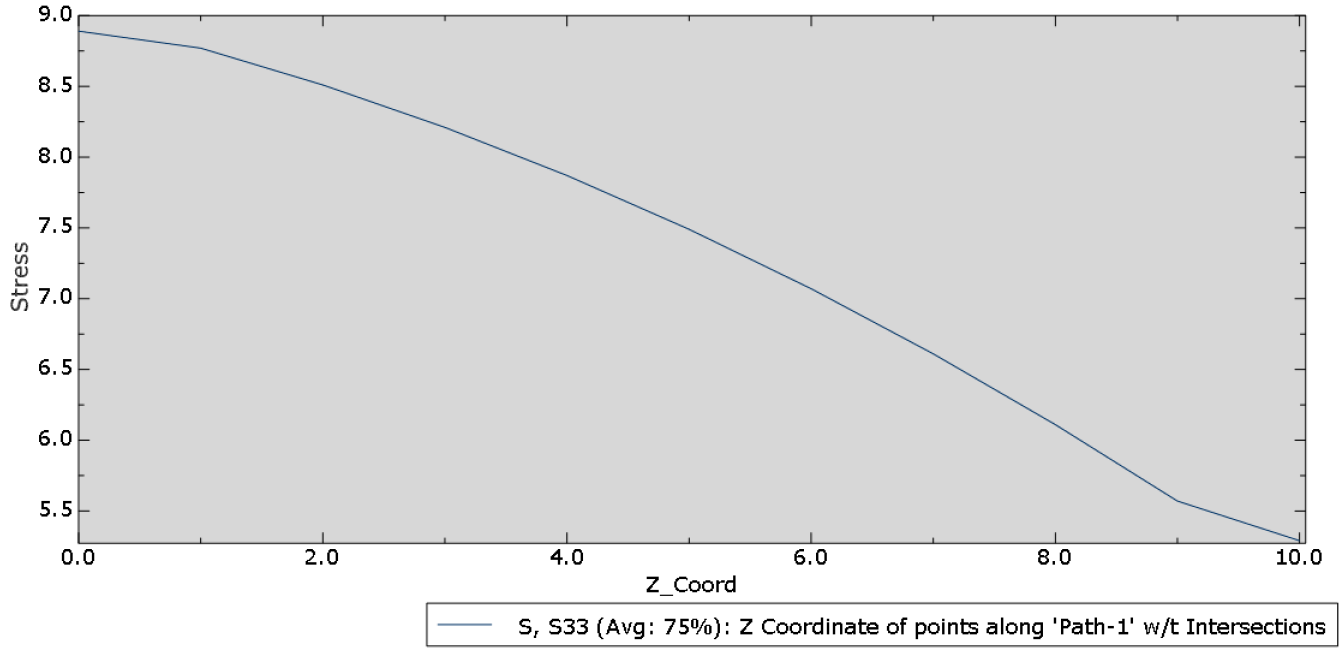
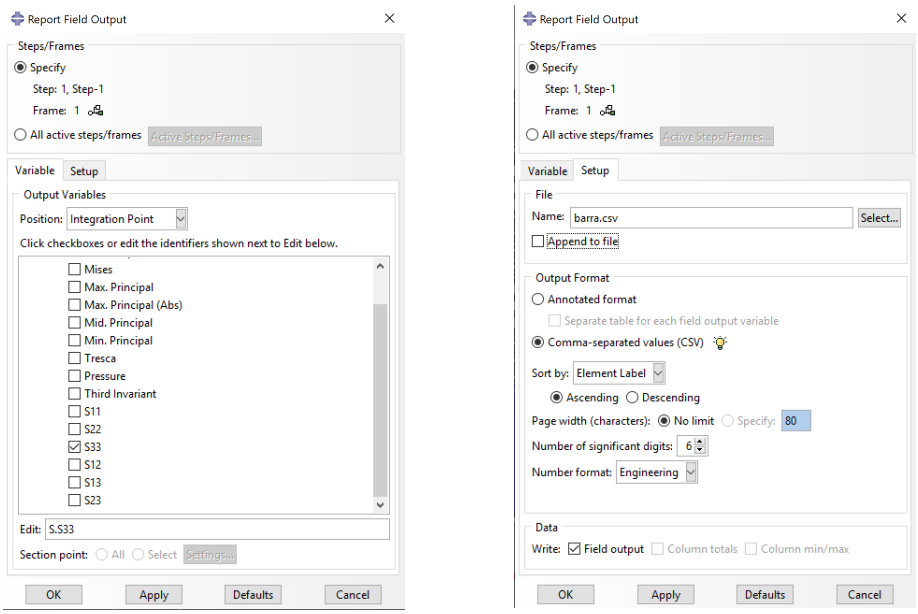


Figura 29: Tension σ_z a lo largo del eje Z

Para ello lo realizamos de una manera distinta. Debemos escoger, en la pestaña “Report/”



(a) Variable (b) Setup

Figura 30: Field Output

la opción *Field Output*. Se nos abrira una ventana donde, en la pestaña *Variable* deberemos seleccionar el Field S33 en los puntos de integración (figura **??**), mientras que en la pestaña Setup debemos denominar al archivo `s33.csv` (figura **??**).

4. Ejemplo de programación elementos finitos en Python

A continuación se adjunta un *Notebook* de *Python* con el seguimiento de la creación de códigos y aplicación al caso de la barra 1D.

El archivo `*.ipynb` puede descargarse del siguiente repositorio, o bien abrir directamente en cualquier de los programas que a continuación se mencionarán:

`Repositorio Git-Hub`

El *Notebook* puede correrse el programa `Jupyter`, que pertenece a la suite `Anaconda`, suite en abierto para plataformas *Windows, Mac y Linux*. Es necesario instalarse dicha suite y cargar el *Notebook* bien abriendo el archivo `*.ipynb`, o bien desde el repositorio de Git-Hub antes mencionado.

Por otro lado, también nos encontramos soluciones que nos permiten correr el script sin necesidad de instalar ningún programa, son soluciones *en la nube*. Entre otras, las más conocidas, son:

- `Colab de Google` Es necesario tener cuenta de Gmail.
- `MyBinder` Solo desde repositorio 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

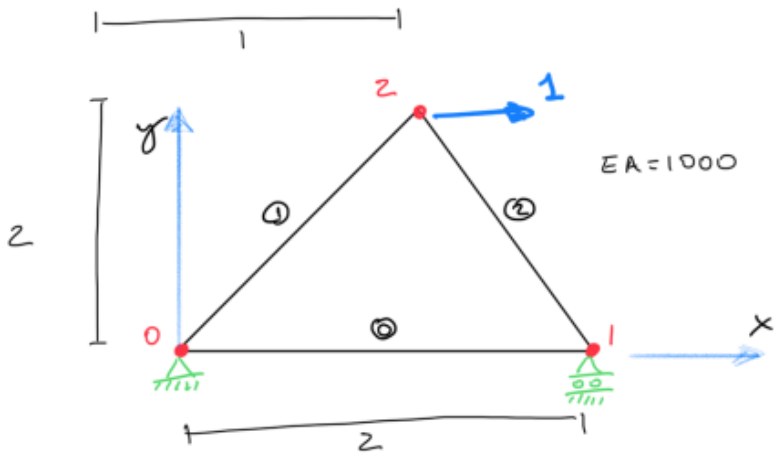
El código se implementará en un notebook de jupyter que permite la mezcla de texto, gráficos y código de forma sencilla mediante el uso de celdas de tipo *markdown* o tipo *code*.

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.

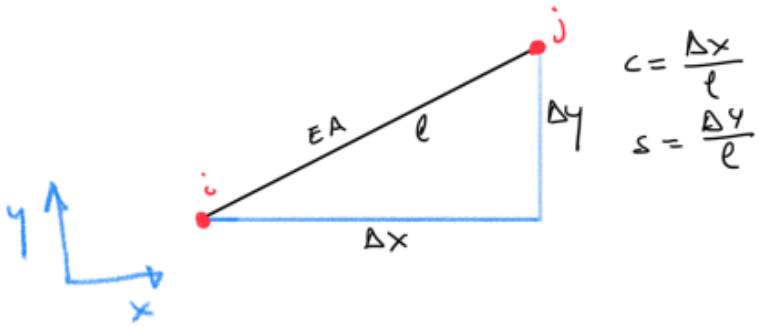
Como los grados de libertad considerados son dos por nodo (u según x,v según y) las dimensiones de las matrices serán, siendo n el número de nodos.

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)
```

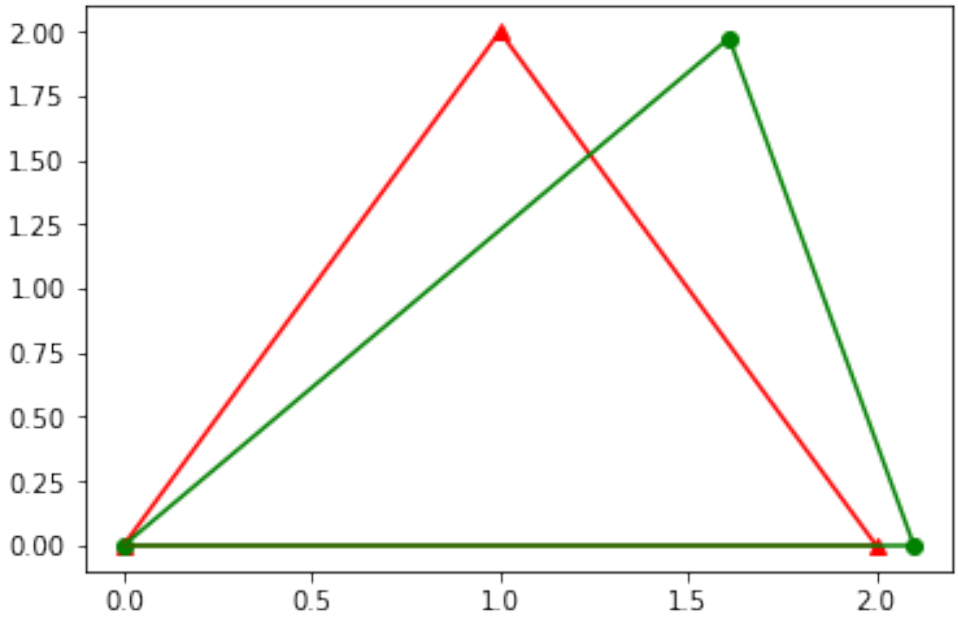
```
[[ 1.00000000e-20  1.00000000e-20]
 [ 1.00000000e-03 -1.00000000e-20]
 [ 6.09016994e-03 -2.50000000e-04]]
```

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.

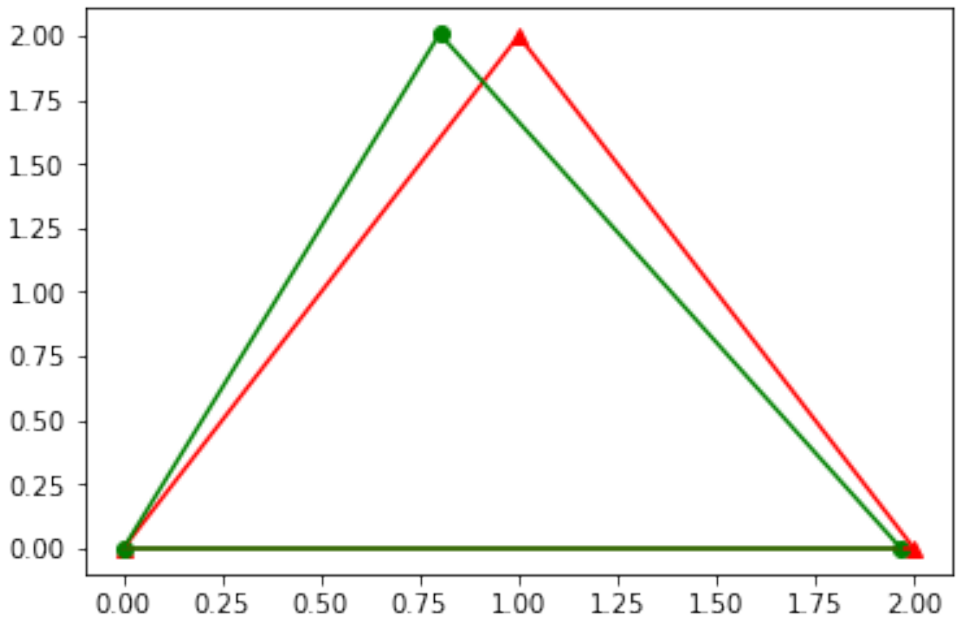
[39]:
```
d2 = f_solve(x,elementos,fuerzas2,cc2)
print(d2)
```

```
[[-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.

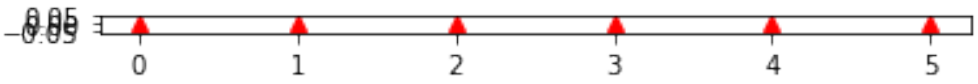
[41]:
```
xf = lambda u,v: u
yf = lambda u,v: v
f1  =lambda u: np.linspace(u[0],u[1],u[2])
f2 = lambda u,v : np.meshgrid(f1(u),f1(v))
f3 = lambda xu,u,v: xu(*f2(u,v)).reshape(u[2]*v[2],1)
f4 = lambda xu,yu,u,v : np.hstack((f3(xu,u,v),f3(yu,u,v)))
faux = lambda patterns,ntot: map(lambda x: [x[0],x[1],ntot],patterns)
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^')
```

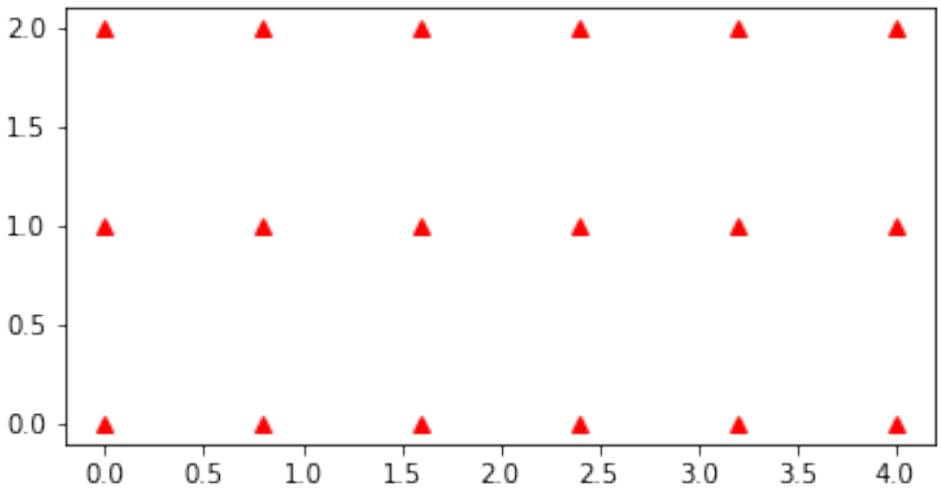
[43]:
```
[<matplotlib.lines.Line2D at 0xa78f5390>]
```



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>]
```



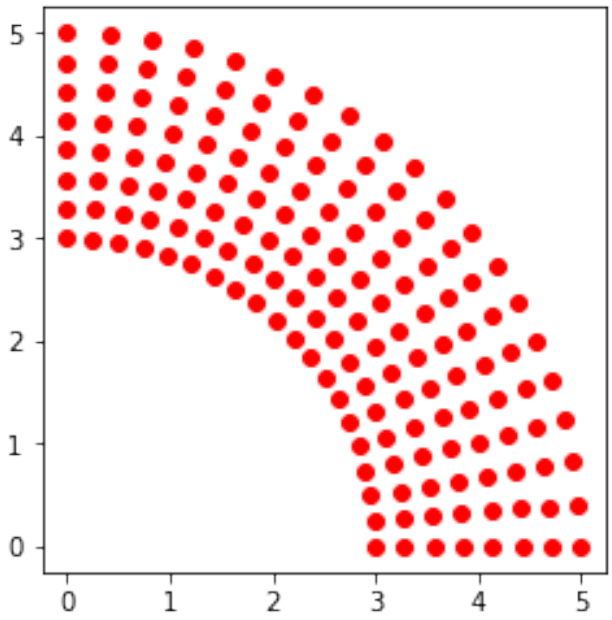
[45]:
```
print(xl2)
```

```
[[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]]
```

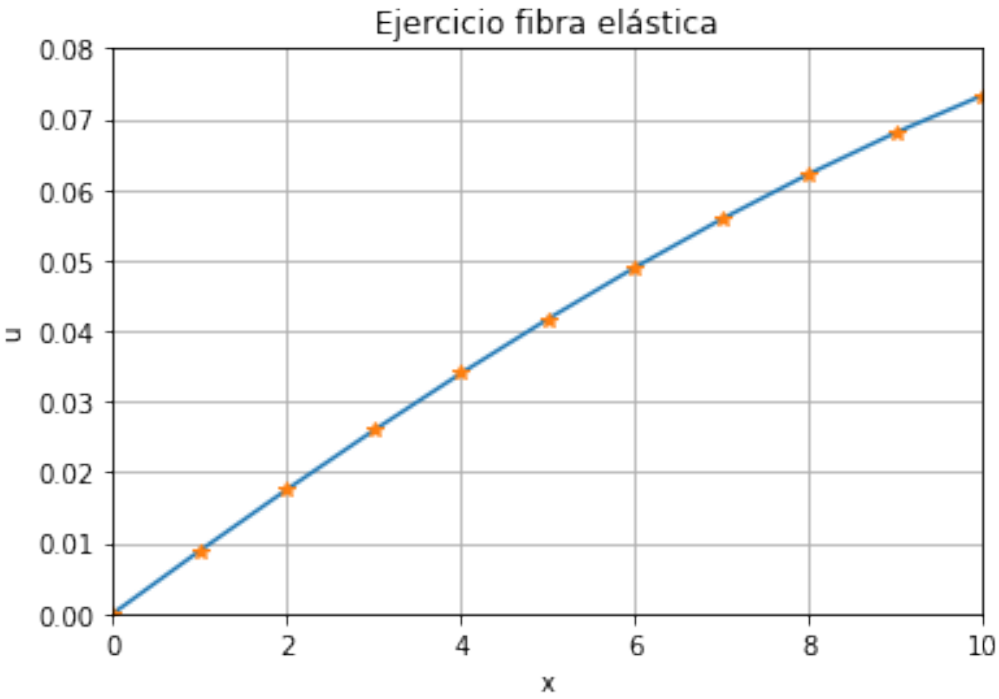
Leemos los resultados de abaqus que estarán preparados en un fichero con dos columnas de números separados por comas. Utilizaremos el módulo pandas para la lectura.

[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>]
```

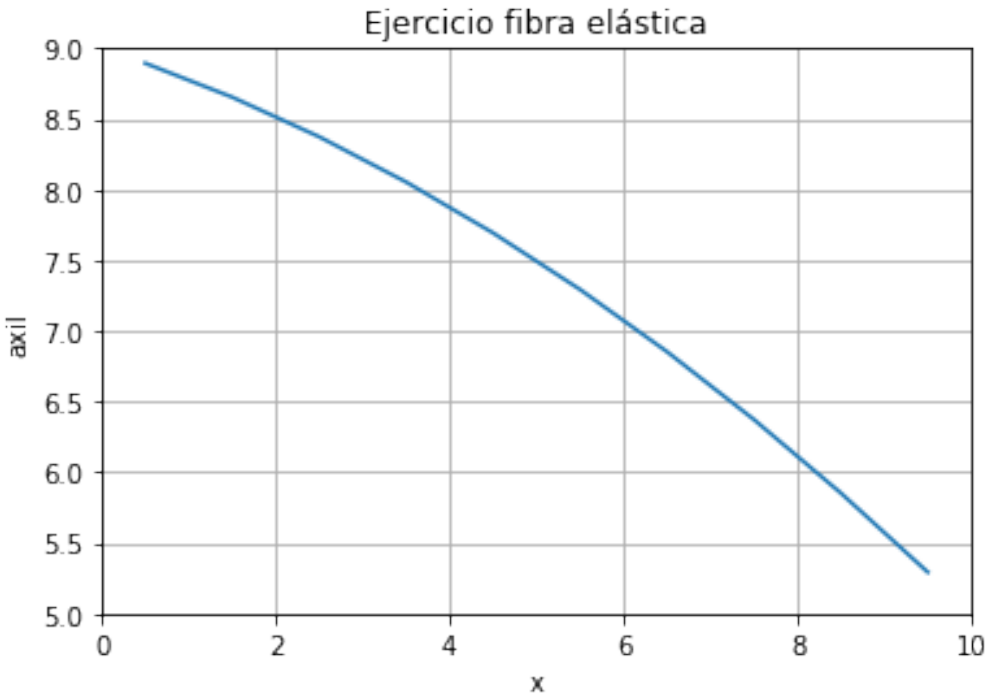


Se puede hacer lo mismo entre los axiles y la tensión S33 obtenida en **abaqus** dado que al ser la sección de 1×1 coinciden numéricamente. Las tensiones se representarán en el punto geométrico en el que han sido obtenidas, es decir, el punto de Gauss del elemento.

[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]
```

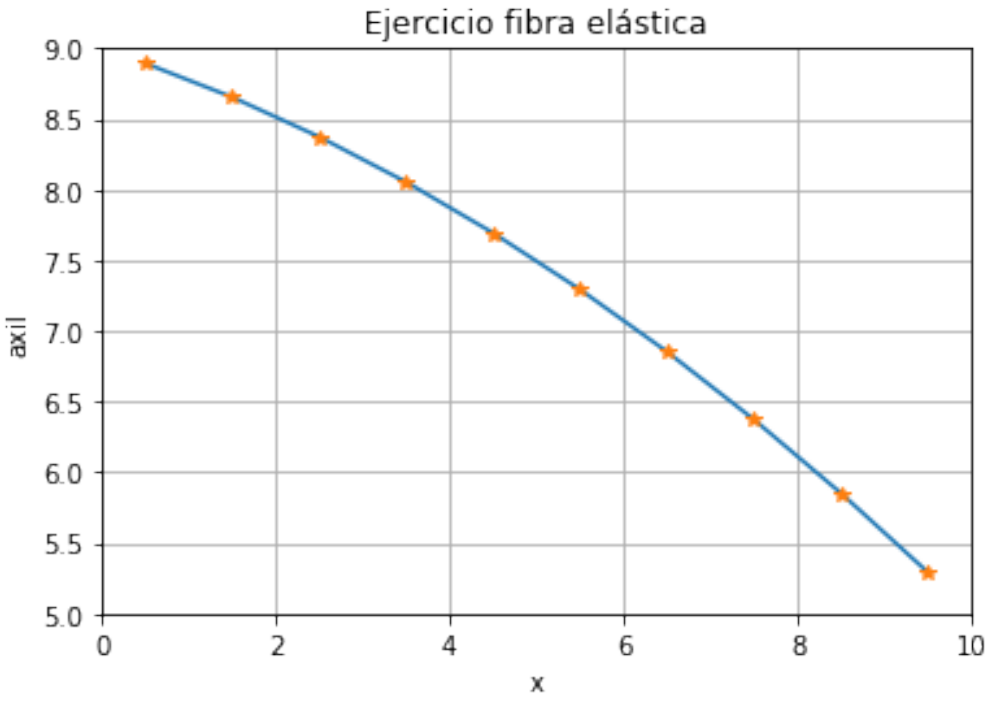
[65]:
```
array([5.29, 5.85, 6.37, 6.85, 7.29, 7.69, 8.05, 8.37, 8.65, 8.89])
```

[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>]
```

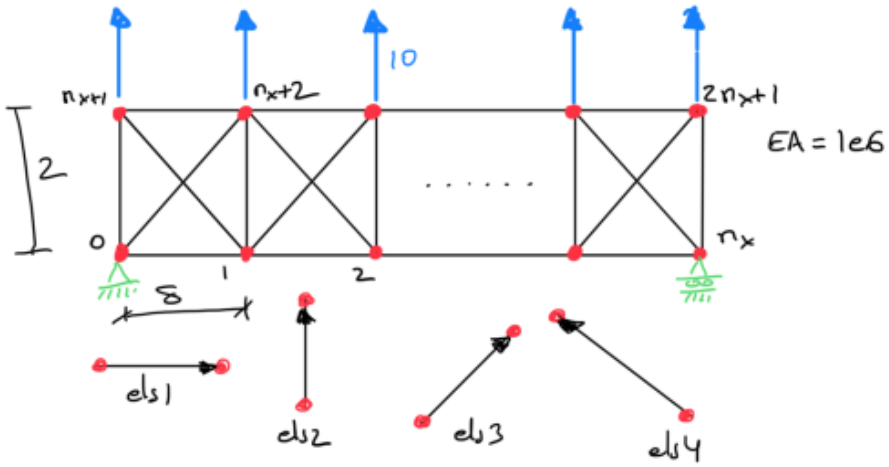


6 Ejemplos

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)
```

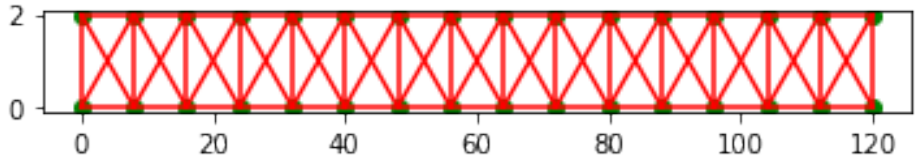
[69]:
```
els1 = genelem([[1,1,1],[1,1,nx]],[0,1],2*nx,x,fc(1e6))
```

[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
```

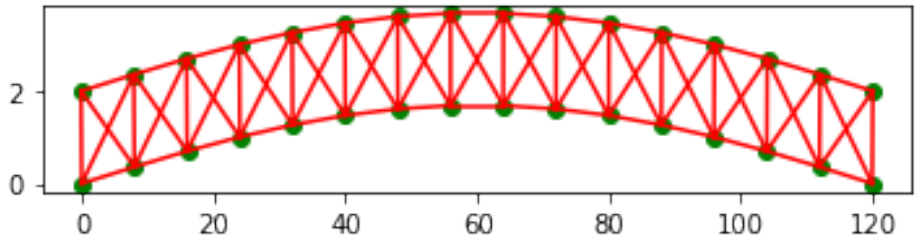
[75]:
```
[[16, 1, 10],
 [17, 1, 10],
 [18, 1, 10],
 [19, 1, 10],
 [20, 1, 10],
 [21, 1, 10],
 [22, 1, 10],
 [23, 1, 10],
 [24, 1, 10],
 [25, 1, 10],
 [26, 1, 10],
 [27, 1, 10],
 [28, 1, 10],
 [29, 1, 10],
 [30, 1, 10],
 [31, 1, 10]]
```

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()
```

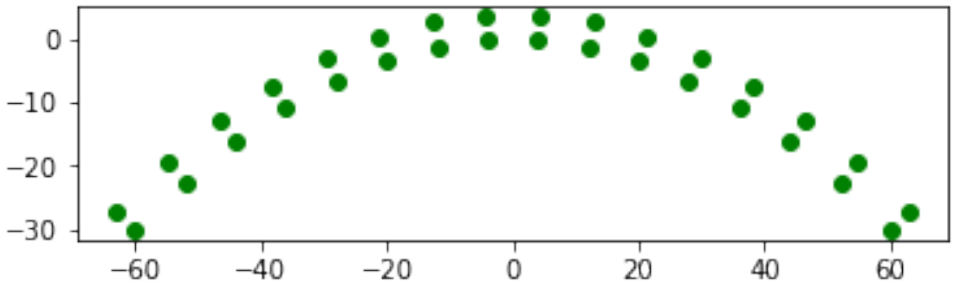
6.2 Arco parabólico de celosía

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.

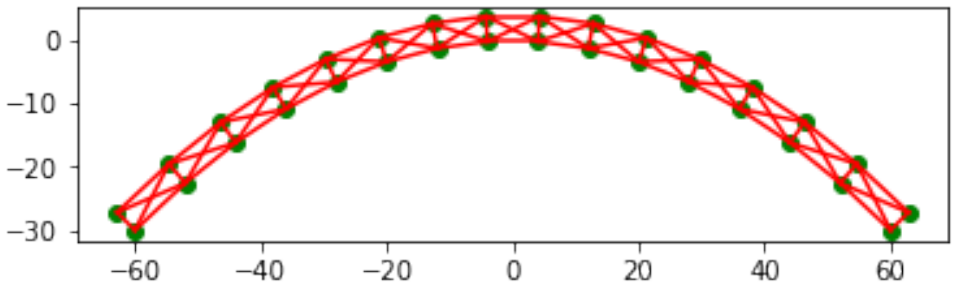
[81]:
```
els1 = genelem([[1,1,1],[1,1,nx]],[0,1],2*nx,x,fc(1e6))
```

[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);
```

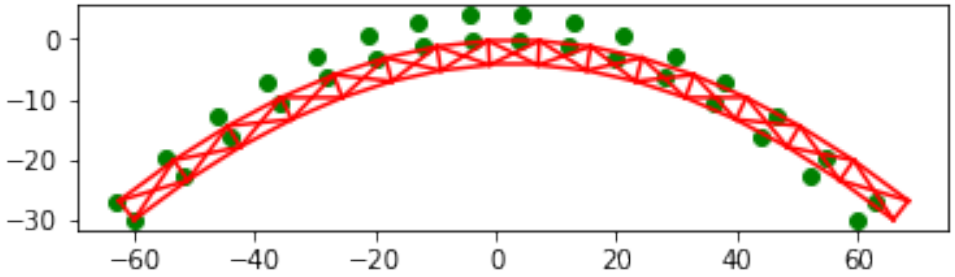


[86]:
```
cc = [[0,0],[0,1],[nx,1]]
fuerzas = genfuerzas([[1,0,1]],[nx+1,1],nx+1,x,fc(-10))
```

[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);
```



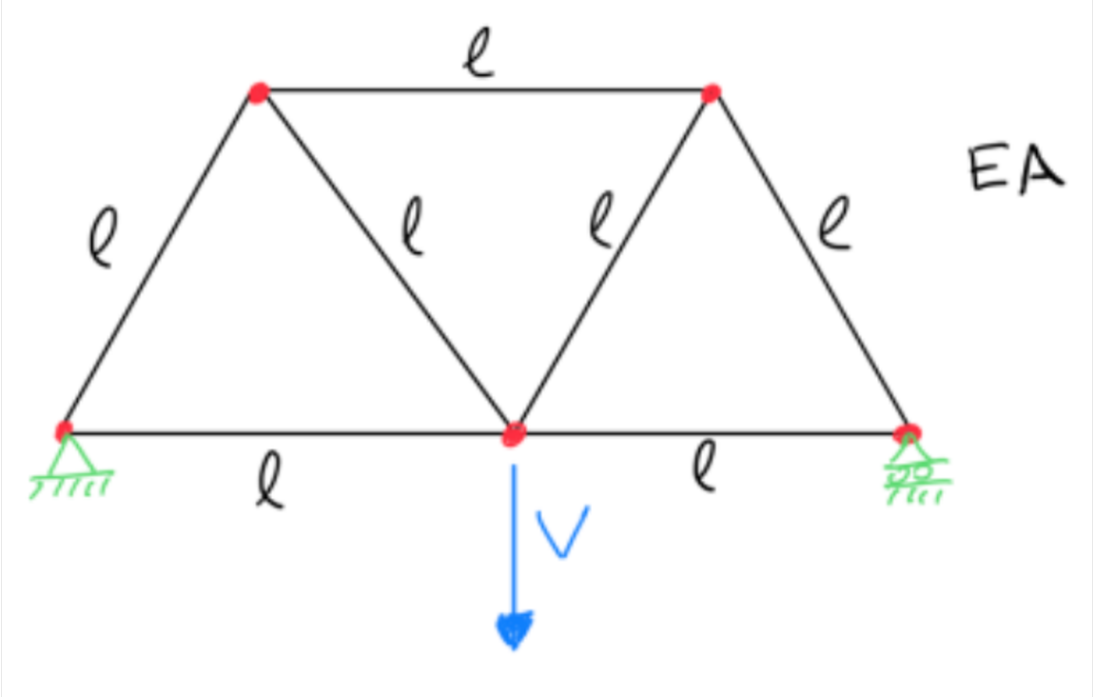
Aparte de los movimientos y de los axiles que ya hemos visto como se obtienen también podemos obtener las reacciones, que serán las fuerzas asociadas a las restricciones. En este caso, por ejemplo,las reacciones verticales están asociadas a los nodos `u[0]` y `u[nx]`.
Como el apoyo derecho es un carrito 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

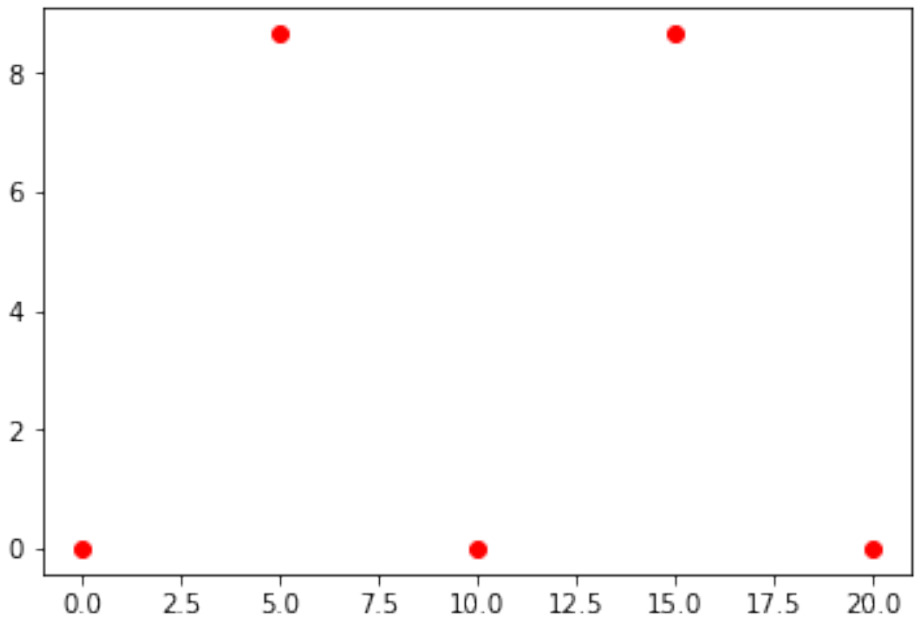
Y obtener el movimiento horizontal del apoyo derecho.

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>]
```

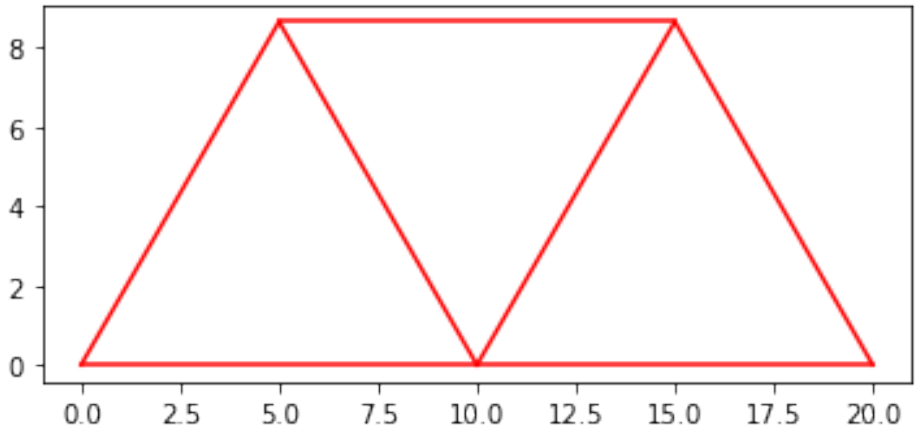


Y a continuación los elementos.

[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
```

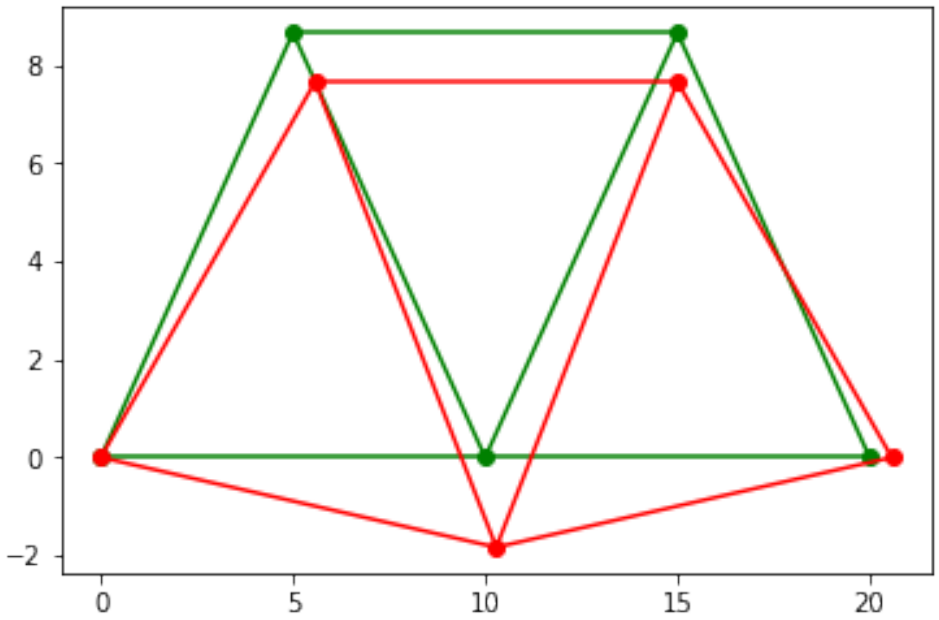
que si comparamos con la solución.

[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]
```

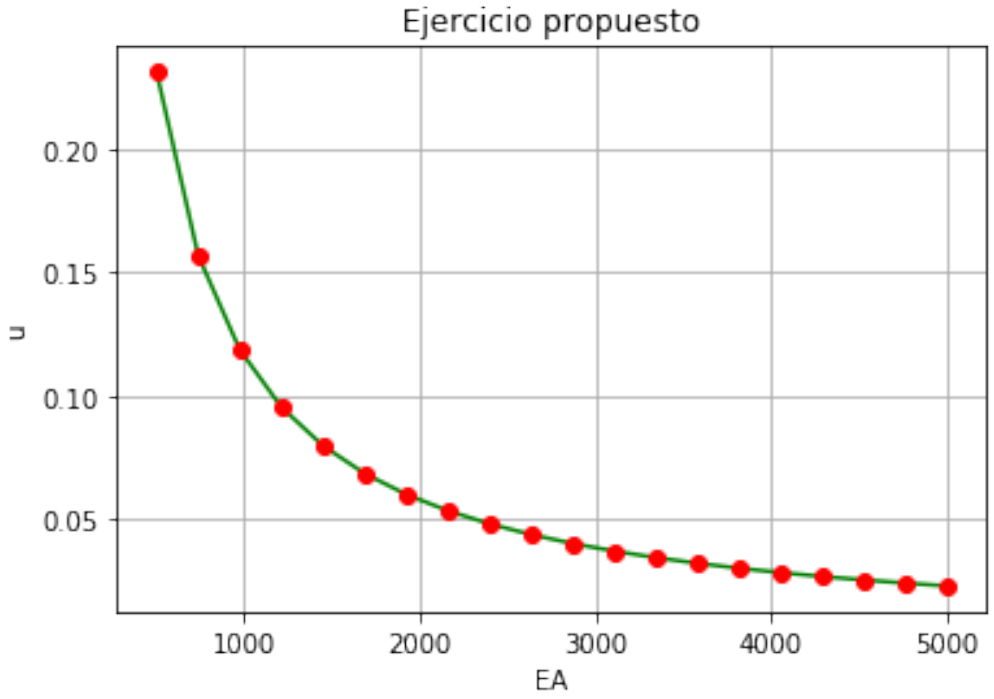
[106]:
```
0.1154700538379252
```

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. Simulación con MatLab / Octave

A continuación se expone la misma solución realizada en `MatLab`, para todo aquel que domine el lenguaje de programación de esta herramienta y quiera comprobar dicho problema también con este programa.

Código MatLab / Octave

La organización del código en `Matlab` / `Octave` es libre, pero, puesto que es la primera aproximación que el alumno hace a un código de elementos finitos, el seguir los esquemas propuestos a continuación puede resultar muy interesante para tener unas pautas.

- Preproceso

  A continuación se describe el preproceso, donde material, geometría, condiciones de contorno y mallado son definidas. Se debe también, según el tipo de problema y los grados de libertad que tenga el problema, definir el número de grados de libertad totales.

Código 1: Esquema Matlab *(a)*

```
% Definición de parámetros particulares de cada alumno:
Nmat=1124                       % número de matrícula del alumno
m=floor(Nmat/1000)              % cifra de millares
c=floor((Nmat-m*1000)/100)      % cifra de centenas
d=floor((Nmat-m*1000-c*100)/10) % cifra de decenas
u=Nmat-m*1000-c*100-d*10        % cifra de unidades

% A: PREPROCESO
%-----------------------------------------------------------------

% 1. Geometría
A=1;               % Area
L=10;              % Longitud de la barra

% 2. Material
E=1000;            % Módulo elástico

% 3. Condiciones de contorno en carga
P=5;                 % dato de fuerza aplicada en x=L
q0=d/10;             % Fuerza volumetrica en x=0
qf=q0+u/10;          % Fuerza volumétrica en x=L

% 4. Malla
Nele=11-c;           % número de elementos
Nnod=Nele+1;         % número de nodos
q=linspace(q0,qf,Nnod);
h=L/Nele;            % tamaño de cada elemento

% 5. Grados de Libertad del problema
% Dim = GDL totales = grados de libertad nodales x nº nodos
gdl=1;
Dim=Nnod*gdl;
```

- Construcción de matrices elementales y globales

  La primera parte del *solver* trata de construir, a partir de las matrices elementales, y tras el ensamblado, las matrices globales de rigidez y carga. La matriz de rigidez elemental es

igual para todos los elementos, al tratarse de elementos del mismo tamaño y propiedades constantes. En cambio, la de fuerza volumétrica, en este caso varía, por ser una carga que varía a lo largo de la barra. Finalmente, en el último nodo, se ha de aplicar la carga externa de 5 N.

Código 2: Esquema Matlab *(b)*

```
% B: CONSTRUCCIÓN de MATRICES y VECTORES globales
%-------------------------------------------------------------------
% 0. Inicialización
K=zeros(Dim);
F=zeros(Dim,1);
Fext=zeros(Dim,1);

% 1. Matriz de rigidez elemental (común a todos los elementos)
k=(E*A/h)*[1,-1;-1,1];

% 2. Ensamblaje de la matriz global
for i=1:Nele
    K(i:i+1,i:i+1)=K(i:i+1,i:i+1)+k;        % ensambla rigidez
end

% 3. Ensamblaje del vector de fuerzas volumetricas global
for i=1:Nele
    % vector elemental de cargas distribuidas
    fvol=(h/6)*[2*q(i)+q(i+1);q(i)+2*q(i+1)];
    % ensambla cargas
    F(i:i+1)=F(i:i+1)+fvol;
end

% 4. Suma del vector de fuerzas externas
Fext(Nnod)=Fext(Nnod)+P;
F=F+Fext;           % suma cargas en contorno y cargas distribuidas
```

- Aplicación de condiciones de contorno y resolución

  Para el núcleo central del *solver* es necesario aplicar las condiciones de contorno, reduciendo el sistema matricial y, finalmente, invirtiendo esta matriz reducida. Se comprueba el equilibrio a posteriori.

Código 3: Esquema Matlab *(c)*

```
% C: APLICACIÓN de las CONDICIONES de CONTORNO y RESOLUCIÓN
%-------------------------------------------------------------------

% 1. Reducción de la matriz de rigidez
Kg=K(2:Nnod,2:Nnod);

% 2. Reducción del vector de fuerzas
Fg=F(2:Nnod);

% 3. Resuelve sistema reducido para desplazamientos
ug=Kg\Fg;

% 4. Desplazamientos y reacciones. Equilibrio: Sum fuerzas = 0
un=[0;ug];
fint=K*un;
r_0=fint(1)-F(1);
```

```
Eq=sum(fint);
if abs(Eq)>1e-10
    Disp('Equilibrio no alcanzado');
end
```

- Postproceso

  Dentro del postproceso nos interesa calcular las deformaciones y a través de ellas las tensiones a nivel elemental. Una vez calculada la solución analítica tanto de tensiones como de desplazamientos (ecuaciones (8)), podemos realizar una comparativa con nuestros resultados, a nivel nodal de los desplazamientos y a nivel elemental de las tensiones.

Código 4: Esquema Matlab *(d)*

```
% D: POSTPROCESO
%-----------------------------------------------------------------

% 1. Deformaciones y Tensiones
epsilon=zeros(Nele,1);
for i=1:Nele
    epsilon(i)=(un(i+1)-un(i))/h;
end
sigma=E*epsilon;

% 2. Solución analítica
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. Gráfico

% 3.1. Dibuja la solucion del desplazamiento en x
figure
xn=linspace(0,L,Nele+1);
plot(xn,un,'-+',xc,uc,'-','LineWidth',2,'MarkerSize',10);
% hold
title(['Ejercicio de fibra elástica 1D MEF, Nmat=' num2str(Nmat)]);
legend({'Solución MEF','Solución analitica'},'Location','Northwest');
xlabel('Coordenada x (mm)');
ylabel('Desplazamiento u (mm)');

% 3.2. Dibuja la solucion de la tensión en 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(['Ejercicio de fibra elástica 1D MEF, Nmat=' num2str(Nmat)]);
legend({'Solución MEF','Solución analitica'},'Location','Northeast');
xlabel('Coordenada x (mm)');
ylabel('Tensión \sigma (N/mm^2)');

pause;
```

A continuación, en la figura **??**, podemos ver la solución obtenida con el código que hemos desarrollado en `MatLab` / `Octave` y la solución analítica a partir de las ecuaciones (8). Como vemos, la solución de los desplazamientos ha sido obtenida en los nodos, por lo que a la localización de cada nodo se corresponderá un valor del desplazamiento. En cambio, las tensiones son obtenidas en los puntos de integración de los elementos, en este caso un punto por elemento en el medio del mismo. Por ello debemos dibujar el valor de las tensiones en la localización de los puntos de integración. Además hemos añadido la función *stairs* para resaltar que el valor de las tensiones es constante en cada elemento.

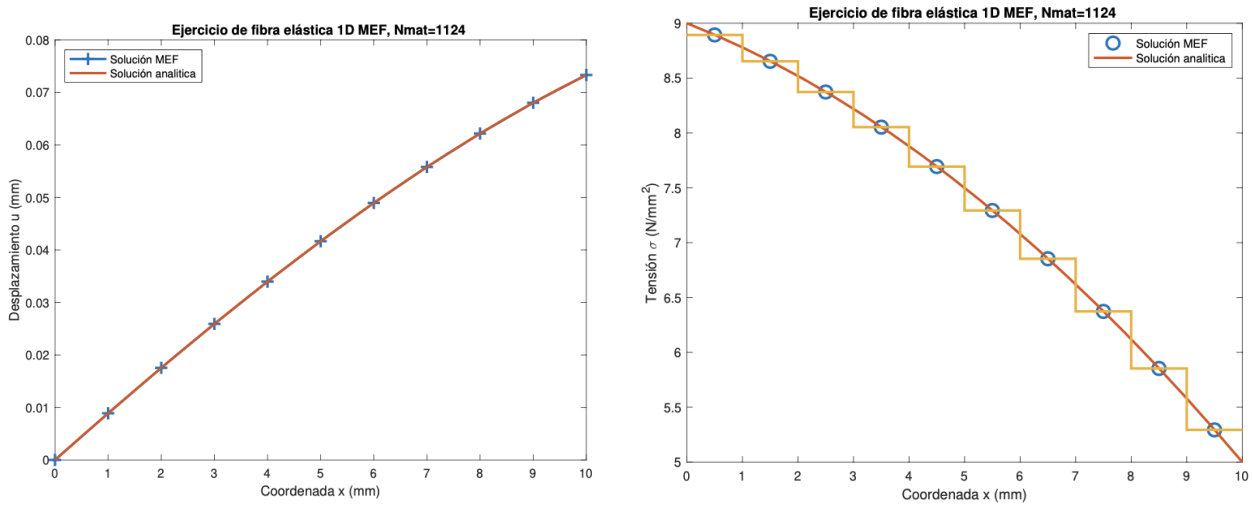


Figura 31: Comparación entre la solución obtenida con MatLab y la solución analítica.

- Comparativa con solución analítica en `MatLab` / `Octave`

  En primer lugar debemos dejar preparado el código de `MatLab` / `Octave` para introducir los datos procedentes de *Abaqus*. A continuación se presenta el código, similar al de postproceso empleado anteriormente.

Código 5: Esquema Matlab-Abaqus

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

xn_ab=[

    ];

un_ab=[

    ];

xe_ab=[

    ];

sigma_ab=[

    ];

% % 1. Dibuja la solucion del desplazamiento en x
figure
```

```
plot(xn_ab,un_ab,'-+',xc,uc,'-','LineWidth',2,'MarkerSize',10)
%hold
title(['Ejercicio de fibra elástica 1D MEF, Nmat=' num2str(Nmat)])
legend({'Solución MEF Abaqus','Solución analitica'},'Location','Northwest')
xlabel('Coordenada x (mm)')
ylabel('Desplazamiento u (mm)')

% 2. Dibuja la solucion de la tensión en 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(['Ejercicio de fibra elástica 1D MEF, Nmat=' num2str(Nmat)])
legend({'Solución MEF Abaqus','Solución analitica'},'Location','Northeast')
xlabel('Coordenada x (mm)')
ylabel('Tensión \sigma (N/mm^2)')
```

Como vemos, dejamos huecos para los vectores que provengan de los resultados de *Abaqus*. Los vectores los rellenamos con los datos que extraigamos de Abaqus. El resultado de la comparativa se ve en la figura **??**. Se puede comprobar que el resultado es idéntico al obtenido con el programa 1D en Matlab, figura **??** y el realizado en Python y como anteriormente también coinciden de forma exacta los valores discretos en los nodos y en el centro de los elementos con la solución analítica.

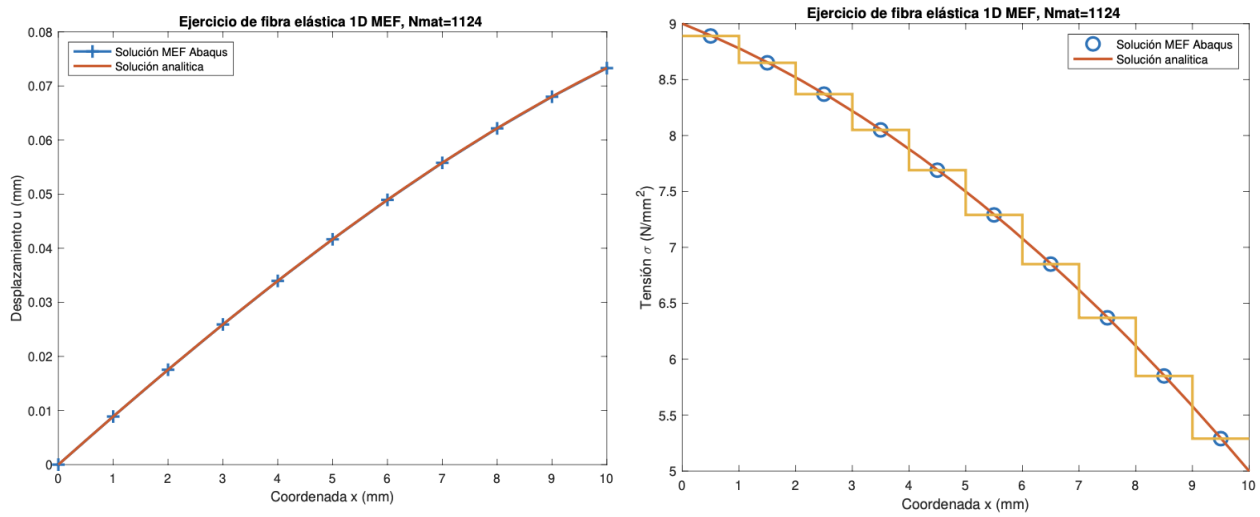


Figura 32: Comparación entre la solución obtenida con *Abaqus* y la solución analítica.