Análisis matricial de armaduras con Python

Aprende el método de rigidez
construyendo tu propia calculadora



Eddy Eder Sucapuca Cruz

Análisis matricial de armaduras con Python
Aprende el método de rigidez construyendo tu propia calculadora

Eddy Eder Sucapuca Cruz
Versión preliminar, 2026.

© 2026 Eddy Eder Sucapuca Cruz.

El texto y las figuras de este libro se distribuyen bajo la licencia Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional (CC BY-NC-SA 4.0): `https://creativecommons.org/licenses/by-nc-sa/4.0/deed.es`. Se permite copiarlos, distribuirlos y adaptarlos sin fines comerciales, citando al autor y compartiendo las obras derivadas con la misma licencia.

El código de la calculadora y de los notebooks se distribuye bajo la licencia MIT: puede usarse, modificarse e incorporarse en otros programas, también comerciales, conservando el aviso de autoría.

El código, los notebooks y la versión web del libro están en `https://github.com/eder3232/libro-rigidez-python-armaduras2d`.

Cómo citar este libro:
Sucapuca Cruz, E. E. (2026). *Análisis matricial de armaduras con Python: aprende el método de rigidez construyendo tu propia calculadora.* `https://github.com/eder3232/libro-rigidez-python-armaduras2d`

Compuesto con LaTeX en las tipografías Libertinus y Source Code Pro. Todos los resultados numéricos y los gráficos del libro se generan ejecutando el código que se muestra.

Índice general

Prefacio IX

I Primeros pasos 1

1 Por qué programar el método de rigidez 3
1.1 Los límites del cálculo a mano . . . 4
1.2 Qué hay dentro de un programa comercial . . . 5
1.3 El mapa del método . . . 5
1.4 Tres armaduras para aprender . . . 6
1.5 Lo que necesitamos para empezar . . . 7
Repaso y resumen del capítulo . . . 7
Ejercicios . . . 8

2 Nuestro primer notebook 9
2.1 La geometría de una barra . . . 9
2.2 Nuestro primer notebook . . . 10
2.3 Variables y operaciones . . . 12
2.4 El módulo `math` . . . 13
2.5 Mostrar resultados con `print` y f-strings . . . 14
2.6 La geometría de una barra en Python . . . 15
Repaso y resumen del capítulo . . . 16
Ejercicios . . . 16

II Del modelo a la matriz 19

3 Describir una armadura 21
3.1 Los datos de una armadura . . . 21
3.2 Listas . . . 22
3.3 Diccionarios . . . 23
3.4 El modelo del ejemplo A . . . 24
3.5 Consultar el modelo . . . 25
3.6 Ver la armadura . . . 26

Repaso y resumen del capítulo 29
Ejercicios 29

4 La geometría de todas las barras 31
4.1 El mismo cálculo para cada barra 31
4.2 El bucle `for` 32
4.3 La geometría de todas las barras 33
4.4 Verificación 34
4.5 Dibujar cualquier armadura 34
Repaso y resumen del capítulo 36
Ejercicios 37

5 La rigidez de una barra 39
5.1 La matriz de rigidez de una barra 39
5.2 Funciones 40
5.3 NumPy 42
5.4 La matriz de rigidez en Python 43
5.5 Verificación 44
Repaso y resumen del capítulo 45
Ejercicios 46

6 Los grados de libertad 47
6.1 Los grados de libertad de la armadura 47
6.2 Decisiones con `if` 48
6.3 La numeración de los grados de libertad 49
6.4 Grados de libertad libres y restringidos 50
6.5 De la numeración desde 1 a los índices de Python 51
6.6 Una revisión previa de la estabilidad 52
Repaso y resumen del capítulo 54
Ejercicios 54

7 Ensamblaje de la matriz global 55
7.1 La idea del ensamblaje 55
7.2 Matrices de ceros e índices 56
7.3 El ensamblaje, término a término 57
7.4 Mostrar una matriz con sus grados de libertad 57
7.5 El ensamblaje con `np.ix_` 59
7.6 Verificación 59
Repaso y resumen del capítulo 61
Ejercicios 61

III Resolver y entender 63

8 Cargas y partición del sistema 65
8.1 Lo conocido y lo desconocido . . . 65
8.2 El vector de cargas . . . 66
8.3 Rebanadas . . . 67
8.4 La partición del sistema . . . 68
8.5 Verificación . . . 69
Repaso y resumen del capítulo . . . 70
Ejercicios . . . 70

9 Desplazamientos y reacciones 73
9.1 La solución del sistema . . . 73
9.2 Resolver un sistema de ecuaciones con NumPy . . . 74
9.3 Armaduras inestables: `raise`, `try` y `except` . . . 75
9.4 Desplazamientos y reacciones del ejemplo A . . . 76
9.5 Verificación del equilibrio . . . 77
Repaso y resumen del capítulo . . . 79
Ejercicios . . . 79

10 Fuerzas internas 81
10.1 De los desplazamientos a la fuerza axial . . . 81
10.2 La fuerza axial en Python . . . 82
10.3 Tablas de resultados . . . 83
10.4 Verificación: el equilibrio de cada nodo . . . 85
10.5 Las fuerzas del ejemplo B . . . 87
Repaso y resumen del capítulo . . . 88
Ejercicios . . . 89

11 Visualizar los resultados 91
11.1 Dibujar para entender . . . 91
11.2 Matplotlib con más detalle . . . 92
11.3 Una función para dibujar la armadura . . . 93
11.4 La deformada . . . 94
11.5 El diagrama de fuerzas axiales . . . 95
Repaso y resumen del capítulo . . . 98
Ejercicios . . . 98

12 Mi calculadora de armaduras 101
12.1 El camino recorrido . . . 101
12.2 Una función que lo hace todo . . . 101
12.3 El informe de resultados . . . 103
12.4 El ejemplo C: un puente peatonal . . . 107
12.5 Mi calculadora . . . 111
Repaso y resumen del capítulo . . . 112

Ejercicios 113

A Referencia rápida de Python y NumPy 115
A.1 Notebooks en Google Colab 115
A.2 Números, texto y resultados 115
A.3 Listas y diccionarios 116
A.4 Bucles y decisiones 117
A.5 Funciones y errores 117
A.6 NumPy 118
A.7 Matplotlib 118

B Leer un error de Python 121
B.1 Cómo se lee un mensaje de error 121
B.2 Un nombre mal escrito 122
B.3 Errores de escritura 122
B.4 Datos equivocados 123
B.5 Los errores de la calculadora 124
B.6 Los errores que no avisan 125

C Trabajar en local 127
C.1 Qué necesitamos 127
C.2 El problema de los entornos 127
C.3 Con uv 128
C.4 Con venv y pip 128
C.5 La calculadora como módulo 129

D Validación con OpenSees 131
D.1 OpenSees 131
D.2 El modelo en OpenSees 131
D.3 Ejemplo B 134
D.4 Ejemplo C 138
D.5 Todas las comparaciones de un vistazo 140
D.6 Conclusión 141

E Notación y convenciones de signos 143
E.1 Símbolos 143
E.2 Convenciones de signos 143
E.3 Unidades 144
E.4 Del símbolo al código 145

F Soluciones de los ejercicios 147
Capítulo 1 147
Capítulo 2 148
Capítulo 3 149
Capítulo 4 152

Capítulo 5 153
Capítulo 6 155
Capítulo 7 156
Capítulo 8 157
Capítulo 9 159
Capítulo 10 161
Capítulo 11 162
Capítulo 12 166

Bibliografía 171

Índice alfabético 173

Prefacio

Este libro enseña el método de rigidez para armaduras planas de la manera en que mejor se aprende un método: aplicándolo paso a paso hasta poder usarlo sin ayuda. Cada paso del método se convierte en unas pocas líneas de Python, y al final del libro esas líneas forman una calculadora completa, escrita por nosotros mismos, que analiza cualquier armadura plana: calcula sus desplazamientos, reacciones y fuerzas axiales, verifica el equilibrio y dibuja los resultados.

A quién va dirigido

El libro está pensado para estudiantes de ingeniería civil que cursan análisis estructural, por lo general en el cuarto año de la carrera, y para docentes que quieran acompañar ese curso con programación. Supone que conocemos la estática de armaduras, las nociones de esfuerzo y deformación de resistencia de materiales y el álgebra de matrices. *No* supone ningún conocimiento de programación: aprendemos Python a medida que el método lo necesita, y siempre con ejemplos de armaduras.

Tres formatos, un mismo contenido

El libro existe en tres formatos que se complementan:

El libro impreso o en PDF, para leer con calma, estudiar la teoría y resolver los ejercicios. Las soluciones están en el anexo F.

La versión web, con el mismo texto, el código listo para copiar, las soluciones plegadas debajo de cada ejercicio y enlaces directos a los notebooks.

Los notebooks, archivos de Google Colab con el código de cada capítulo, sus soluciones y sus puntos de control.

Todo está en el repositorio del libro, `https://github.com/eder3232/libro-rigidez-python-armaduras2d`.

Cómo trabajar con el libro

Lo más importante es *escribir el código*, no solo leerlo. Recomendamos trabajar así:

1. Crear un notebook propio en nuestro Google Drive, como se explica en el capítulo 2, y abrirlo con Google Colab. No hace falta instalar nada.
2. Leer cada capítulo con el notebook abierto al lado, y escribir y ejecutar cada fragmento de código a medida que aparece.
3. Comparar lo que obtenemos con la salida que muestra el libro. Si coinciden, seguimos; si no, buscamos la diferencia antes de avanzar.
4. Resolver los ejercicios del capítulo antes de mirar las soluciones.

Los capítulos se apoyan unos en otros: el código de cada uno usa las funciones de los anteriores. Por eso conviene leerlos en orden.

Los puntos de control

Puede pasar que algo no funcione y no encontremos el error, o que queramos saltarnos un capítulo. Para eso, cada capítulo termina con un *punto de control*: un notebook con todo el código del libro hasta ese capítulo, ya escrito y probado. Basta con abrirlo, copiarlo a nuestro Drive y continuar desde ahí. El capítulo 12 ofrece, además, la calculadora completa, organizada para analizar nuestras propias armaduras.

Las cajas del texto

El texto se acompaña de cajas con un papel fijo:

Recuerda: la teoría que ya conocemos, en la notación del libro.

Python: una herramienta del lenguaje que usamos por primera vez.

Cuidado: un error frecuente y cómo evitarlo.

Estrategia de resolución: los pasos para resolver un tipo de problema.

Para saber más: una ampliación que puede omitirse sin perder el hilo.

Problema resuelto: un ejemplo completo, con su enunciado, su solución y su verificación.

Las ecuaciones más importantes aparecen resaltadas con un fondo de color.

El código aparece sobre un fondo claro, con sus colores habituales, y lo que produce al ejecutarse aparece debajo, marcado con una línea vertical a la izquierda. Los fragmentos de código del libro son exactamente los de los notebooks: se extraen automáticamente de un mismo archivo y se ejecutan antes de cada edición, de modo que toda salida impresa es la que produce el código.

Notación y unidades

Las matrices se escriben en negrita mayúscula (**K**, **T**) y los vectores en negrita (**U**, **F**); la notación completa y las convenciones de signos están en el anexo E. Trabajamos siempre en kilonewtons y metros, de modo que el módulo de elasticidad va en $\mathrm{kN/m^2}$ y las áreas en $\mathrm{m^2}$. Los nodos, las barras y los grados de libertad se numeran desde 1, como en cualquier texto de análisis estructural.

> **Para saber más**
>
> En el ámbito hispanohablante, una misma estructura recibe nombres distintos: *armadura* en buena parte de América Latina, *cercha* en Colombia, Venezuela y España, y *reticulado* en el Cono Sur. En este libro usamos «armadura», pero los tres términos significan lo mismo.

Licencia

Este libro es una obra abierta. El texto y las figuras se distribuyen bajo la licencia Creative Commons Atribución-NoComercial-CompartirIgual 4.0 (CC BY-NC-SA 4.0), y el código, bajo la licencia MIT: podemos usarlo, modificarlo e incorporarlo en nuestros propios programas, citando su origen.

Eddy Eder Sucapuca Cruz

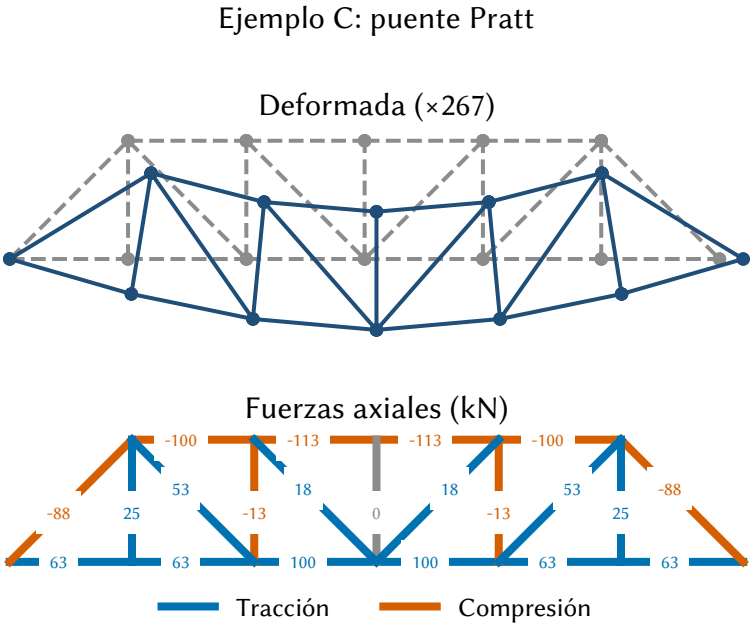
Parte I

Primeros pasos

Capítulo 1

Por qué programar el método de rigidez

Empecemos por el final. El gráfico siguiente muestra el análisis completo de un puente peatonal de 18 metros: arriba, cómo se deforma bajo su carga; abajo, la fuerza axial de cada una de sus 21 barras, en azul las que trabajan en tracción y en naranja las que trabajan en compresión.



No lo dibujó un programa comercial. Lo dibujó una calculadora de unas doscientas líneas de Python, escritas una por una a lo largo de este libro. Al terminarlo, esa calculadora será la nuestra: la habremos escrito, entendido y verificado paso a paso, y podremos usarla para analizar cualquier armadura

plana.

Al terminar este capítulo podremos

- Reconocer los límites del cálculo a mano de armaduras.
- Explicar qué hace por dentro un programa de análisis estructural.
- Describir los pasos del método de rigidez y su orden.
- Conocer los tres ejemplos que recorren el libro.

1.1. Los límites del cálculo a mano

En los cursos de estática aprendimos a analizar armaduras con el método de los nudos y el método de las secciones. Son métodos elegantes y conviene dominarlos: con ellos verificaremos nuestra calculadora a lo largo del libro. Pero tienen dos límites.

El primero es el *tamaño*. El puente del gráfico tiene 12 nodos; el equilibrio de cada nodo da dos ecuaciones, de modo que hay 24 ecuaciones con 24 incógnitas: 21 fuerzas en las barras y 3 reacciones. Resolverlas a mano es posible, pero largo y propenso a errores, y un error en un nudo se arrastra a todos los siguientes. Una armadura de cubierta de una nave industrial puede tener cientos de barras.

El segundo límite es más profundo. Una armadura de b barras, r reacciones y n nodos tiene $b+r$ fuerzas desconocidas y solo $2n$ ecuaciones de equilibrio. Si $b + r > 2n$, la armadura es *hiperestática* (*statically indeterminate*): el equilibrio por sí solo no basta para obtener sus fuerzas, porque estas dependen además de la rigidez de las barras. Es el caso de un cuadro con sus dos diagonales, que estudiaremos como ejemplo B: 6 barras y 3 reacciones frente a 8 ecuaciones. Ningún método basado solo en el equilibrio puede resolverlo.

El *método de rigidez* (*stiffness method*) supera ambos límites. Toma como incógnitas los desplazamientos de los nodos en lugar de las fuerzas, relaciona cada desplazamiento con las fuerzas que produce mediante la rigidez de las barras, y reúne todo en un sistema de ecuaciones lineales,

$$\mathbf{K}\,\mathbf{U} = \mathbf{F},$$

donde $\mathbf{K}$ es la *matriz de rigidez* (*stiffness matrix*) de la armadura, $\mathbf{U}$ el vector de desplazamientos de los nodos y $\mathbf{F}$ el de las fuerzas aplicadas sobre ellos. El método es el mismo para una armadura isostática o hiperestática, de tres barras o de trescientas. Y, sobre todo, es sistemático: cada paso se repite igual para cada barra y cada nodo, lo que lo hace ideal para un computador.

1.2. Qué hay dentro de un programa comercial

Los programas de análisis estructural que se usan en la práctica, como SAP2000, ETABS o Ftool, resuelven las estructuras con el método de rigidez, generalizado a barras que además se flexionan. Cuando dibujamos una armadura en uno de ellos, el programa hace lo mismo que haremos en este libro: calcula la rigidez de cada barra, ensambla la matriz de la estructura, resuelve el sistema y obtiene las fuerzas de cada barra.

Usar un programa sin entender lo que hace por dentro es riesgoso. El programa siempre da un resultado, también cuando los datos están mal: un apoyo olvidado, una carga en la unidad equivocada o un nodo desplazado. Reconocer que un resultado no tiene sentido exige saber de dónde sale. Por eso este libro no enseña a usar un programa, sino a construirlo: al escribir cada paso del método, sabremos qué esperar de él y cómo verificarlo.

> **Para saber más**
>
> El método de rigidez se desarrolló en la industria aeronáutica en la década de 1950, junto con los primeros computadores, y está en la base del método de los elementos finitos. Su desarrollo completo para armaduras, vigas y pórticos está en Kassimali (2012) y en McGuire et al. (2000).

1.3. El mapa del método

La figura 1.1 resume el método de rigidez para armaduras planas y es, a la vez, el mapa de este libro: cada paso es una parte de nuestra calculadora, y cada capítulo construye uno o dos pasos.

En palabras:

1. Describimos la armadura con datos: las coordenadas de sus nodos, las barras que los unen, los apoyos y las cargas.
2. Calculamos la longitud y la orientación de cada barra.
3. Con ellas obtenemos la matriz de rigidez de cada barra, que relaciona los desplazamientos de sus extremos con las fuerzas en ellos.
4. Numeramos los desplazamientos posibles de los nodos, sus *grados de libertad* (*degrees of freedom*).
5. Ensamblamos las matrices de las barras en la matriz de la armadura, y las cargas en el vector de fuerzas.
6. Separamos los grados de libertad libres de los restringidos por los apoyos.

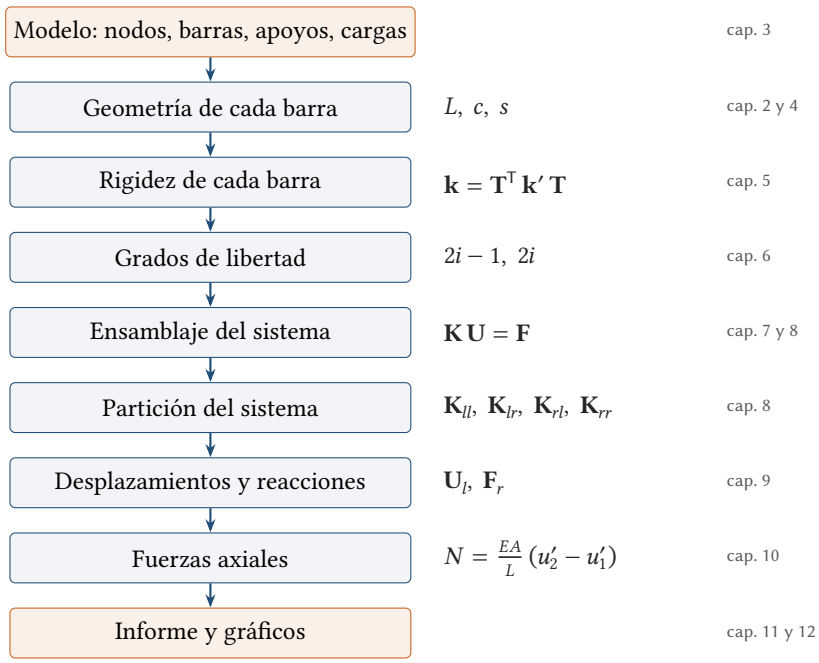


Figura 1.1: Los pasos del método de rigidez, con la ecuación principal de cada uno y el capítulo donde lo programamos.

7. Resolvemos el sistema: obtenemos los desplazamientos de los nodos y, con ellos, las reacciones.
8. De los desplazamientos obtenemos la fuerza axial de cada barra.
9. Presentamos, verificamos y dibujamos los resultados.

> **Recuerda**
>
> Todo el método se basa en tres principios que ya conocemos: el *equilibrio* de cada nodo, la *compatibilidad* (las barras unidas en un nodo se desplazan con él) y la *ley de Hooke* (cada barra se alarga en proporción a su fuerza). Supondremos, como en resistencia de materiales, que el material es elástico lineal y que los desplazamientos son pequeños.

1.4. Tres armaduras para aprender

A lo largo del libro usaremos tres armaduras de acero, con datos realistas ($E = 200\,\text{GPa}$), que se resumen en la tabla 1.1. La columna g es el grado de indeterminación, $g = b + r - 2n$: es cero en una armadura isostática y positivo en una hiperestática.

Tabla 1.1: Los ejemplos del libro.

Ejemplo	Nodos	Barras	g	Para qué sirve
A. Triángulo	3	3	0	Verificar cada paso a mano
B. Cuadro arriostrado	4	6	1	Un caso hiperestático
C. Puente Pratt	12	21	0	Una armadura de verdad

El *ejemplo A* es un triángulo de 6 m de base y 4 m de altura, lo bastante pequeño para comprobar a mano cada número que produzca la calculadora. El *ejemplo B* es el cuadro con dos diagonales del que hablamos antes: nos recordará que el método resuelve lo que el equilibrio por sí solo no puede. Y el *ejemplo C* es el puente del comienzo del capítulo, la prueba final de la calculadora.

Que haya tres ejemplos distintos no es un capricho. Un programa puede dar el resultado correcto con un ejemplo por casualidad, por ejemplo si confunde dos números que en ese ejemplo resultan iguales. Probarlo con armaduras de tamaños y formas distintas hace que esos errores salgan a la luz.

1.5. Lo que necesitamos para empezar

Este libro supone que conocemos la estática de armaduras, las nociones de esfuerzo y deformación de resistencia de materiales, y el álgebra de matrices: producto, transpuesta e inversa. No supone ningún conocimiento de programación. Aprenderemos Python a medida que lo necesitemos, siempre con ejemplos de armaduras, y cada herramienta nueva aparecerá en una caja como esta:

> **Python**
> Las cajas de Python explican las herramientas del lenguaje que usamos por primera vez. El anexo A las reúne todas para consultarlas rápidamente.

Tampoco necesitamos instalar nada: escribiremos nuestro código en notebooks de Google Colab, que funcionan en el navegador. En el capítulo 2 creamos el primero.

Repaso y resumen del capítulo

- El cálculo a mano es largo para armaduras grandes e insuficiente para las hiperestáticas, cuyo grado de indeterminación $g = b + r - 2n$ es positivo.
- El método de rigidez toma como incógnitas los desplazamientos de

los nodos y resuelve el sistema $\mathbf{K}\,\mathbf{U} = \mathbf{F}$. Es el mismo para cualquier armadura y es el que usan los programas comerciales.

- El método tiene nueve pasos (figura 1.1), que programaremos en los capítulos 2 a 12.
- Los ejemplos A, B y C, de tamaños y propiedades distintos, nos permitirán verificar la calculadora.

Ejercicios

Ejercicio 1.1 Calcular el grado de indeterminación de los ejemplos A, B y C, sabiendo que los tres tienen un apoyo articulado y uno de rodillo. ¿Cuáles pueden resolverse con el método de los nudos? Solución: p. 147

Ejercicio 1.2 Cada nodo de una armadura plana puede desplazarse en dos direcciones. ¿Cuántas filas y columnas tiene la matriz $\mathbf{K}$ de cada ejemplo? ¿Cuántos números tiene la del ejemplo C? Solución: p. 147

Ejercicio 1.3 El puente del ejemplo C tiene seis paneles. Para un puente del mismo tipo con p paneles, deducir cuántos nodos y barras tiene, y comprobar que sigue siendo isostático. ¿De qué tamaño sería la matriz $\mathbf{K}$ de un puente de 20 paneles? Solución: p. 147

Capítulo 2

Nuestro primer notebook

Todo análisis de una armadura empieza por su geometría. Antes de hablar de rigideces, cada barra debe quedar descrita por su longitud y por su orientación en el plano. En este capítulo damos el primer paso de nuestra calculadora: obtenemos la geometría de *una* barra, escribiendo nuestras primeras líneas de Python en un notebook de Google Colab.

No necesitamos instalar nada. Basta con una cuenta de Google y un navegador.

Al terminar este capítulo podremos

- Crear un notebook propio en Google Drive y ejecutarlo con Colab.
- Usar variables, operaciones aritméticas y el módulo `math`.
- Mostrar resultados con `print` y f-strings, con unidades y una cantidad adecuada de decimales.
- Calcular la longitud, los cosenos directores y el ángulo de una barra, y verificar el resultado.

2.1. La geometría de una barra

Consideremos una barra que va de un nodo inicial i, de coordenadas (x_i, y_i), a un nodo final j, de coordenadas (x_j, y_j) (figura 2.1). Las proyecciones de la barra sobre los ejes son

$$\Delta x = x_j - x_i, \qquad \Delta y = y_j - y_i, \tag{2.1}$$

y con ellas obtenemos su longitud y sus *cosenos directores* (*direction cosines*):

$$L = \sqrt{\Delta x^2 + \Delta y^2}, \qquad c = \cos\theta = \frac{\Delta x}{L}, \qquad s = \sin\theta = \frac{\Delta y}{L}.$$

El ángulo θ se mide desde el eje x global, en sentido antihorario, hasta la dirección que va del nodo i al nodo j. Por eso el orden de los nodos importa: si intercambiamos i y j, cambian los signos de c y s, y el ángulo aumenta en 180°.

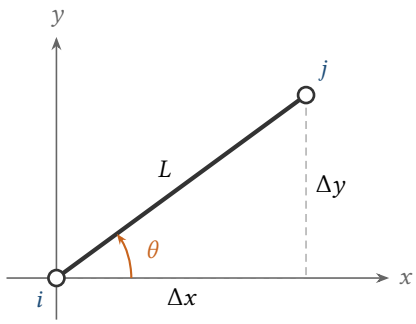


Figura 2.1: Geometría de una barra: proyecciones, longitud y ángulo.

Estas tres cantidades parecen modestas, pero las usaremos en todo el libro: la longitud interviene en la rigidez de la barra, y los cosenos directores en la matriz de transformación del capítulo 5.

2.2. Nuestro primer notebook

Un *notebook* es un documento que mezcla celdas de texto y celdas de código que se pueden ejecutar. Usaremos *Google Colab* (*Google Colaboratory*), un servicio gratuito que ejecuta notebooks de Python en los servidores de Google. Tiene dos ventajas decisivas para nosotros: no requiere instalar nada, y ya incluye las bibliotecas que usa este libro.

Crear el notebook en Google Drive

Para que nuestro trabajo quede guardado y organizado, creamos el notebook dentro de nuestro propio Google Drive:

1. Abrir `https://drive.google.com` con nuestra cuenta de Google.
2. Crear una carpeta para el curso, por ejemplo *Calculadora de armaduras*.
3. Dentro de la carpeta, elegir *Nuevo* → *Más* → *Google Colaboratory*. Si la opción no aparece, elegir *Conectar más aplicaciones*, buscar *Colaboratory* e instalarla.
4. Cambiar el nombre del notebook (arriba a la izquierda), por ejemplo, *Capitulo 02*.

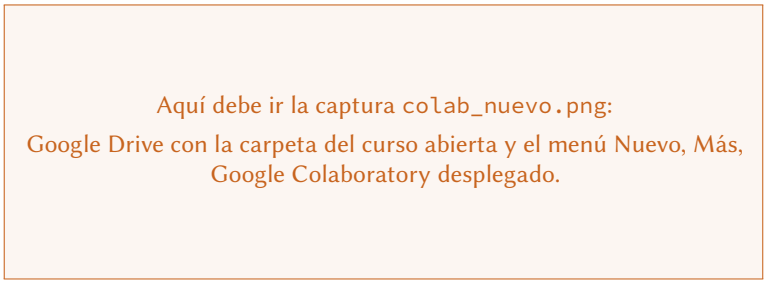


Figura 2.2: Crear un notebook de Colab dentro de nuestra carpeta de Google Drive.

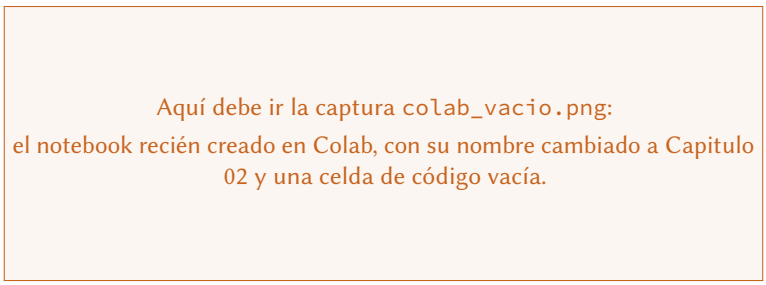


Figura 2.3: Un notebook nuevo, listo para escribir la primera celda.

La figura 2.2 muestra el menú para crear el notebook, y la figura 2.3, el notebook recién creado.

> **Estrategia de resolución**
>
> Recomendamos escribir el código *a mano* en el propio notebook, en lugar de copiarlo y pegarlo. Escribir obliga a leer cada línea, y los pequeños errores que cometamos en el camino enseñan más que el código perfecto. Si algo no funciona, el punto de control al final de cada capítulo permite recuperar el estado correcto y seguir.

Celdas de código

Un notebook nuevo trae una celda de código vacía. Escribimos en ella la siguiente línea y la ejecutamos con el botón de reproducción, a la izquierda de la celda, o con las teclas *Mayús + Enter*:

```
print("Hola, armaduras")
```

```
Hola, armaduras
```

La primera ejecución tarda unos segundos, porque Colab prepara una máquina para nosotros. Debajo de la celda aparece lo que el programa imprime (figura 2.4).

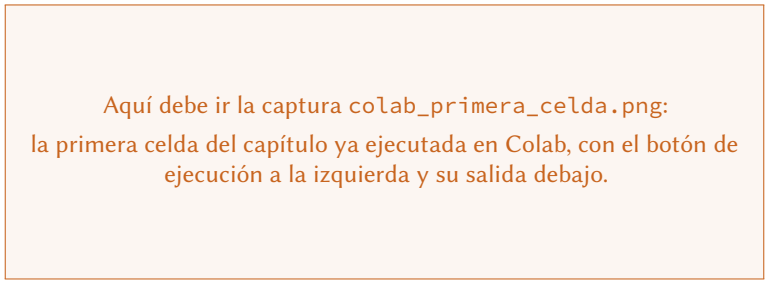


Figura 2.4: La primera celda ejecutada y su resultado.

Una celda también puede contener una operación. Si la última línea de una celda es una expresión, el notebook muestra su valor:

```
2 + 3 * 4
```

```
14
```

Python respeta la precedencia habitual: la multiplicación se evalúa antes que la suma. Para agregar celdas usamos los botones *+ Código* y *+ Texto*; las celdas de texto sirven para anotar qué estamos calculando.

> **Cuidado**
>
> El notebook recuerda lo que se ejecutó, no lo que está escrito. Si cambiamos una celda y no la volvemos a ejecutar, el resto del notebook sigue usando los valores anteriores. Ante un resultado extraño, la opción *Entorno de ejecución → Ejecutar todas* vuelve a ejecutar el notebook en orden, de arriba hacia abajo.

2.3. Variables y operaciones

Una *variable* es un nombre que guarda un valor. Guardemos las coordenadas de los nodos de la barra 3 del ejemplo A, que va del nodo 1, en $(0, 0)$, al nodo 3, en $(3, 4)$:

```
# Coordenadas (en m) del nodo inicial i y del nodo final j
x_i = 0.0
y_i = 0.0
x_j = 3.0
y_j = 4.0
```

El signo `=` no expresa una igualdad matemática: es una *asignación* (*assignment*). Se lee «guardar el valor `0.0` con el nombre `x_i`». Las líneas que empiezan con `#` son comentarios: Python las ignora, pero a nosotros nos recuerdan qué representa cada valor.

> **Python**
> Los nombres de variables pueden contener letras, números y guiones bajos, pero no pueden empezar con un número. En este libro los escribimos en minúsculas, sin tildes ni eñes, y separando las palabras con guiones bajos: `x_i`, `longitud`, `matriz_global`. Python distingue mayúsculas de minúsculas: `L` y `l` son variables distintas.

Con las variables definidas, calculamos las proyecciones de la ecuación (2.1). Al escribir dos expresiones separadas por una coma, el notebook muestra ambas:

```
dx = x_j - x_i
dy = y_j - y_i
dx, dy
```

```
(3.0, 4.0)
```

Las operaciones aritméticas se escriben con `+`, `-`, `*` y `/`, y la potencia con `**`. La suma de los cuadrados de las proyecciones es:

```
dx**2 + dy**2
```

```
25.0
```

Los números muy grandes o muy pequeños se escriben en *notación científica* (*scientific notation*): `2.0e8` significa 2.0×10^8. Así escribiremos el módulo de elasticidad del acero en $\mathrm{kN/m^2}$:

```
E = 2.0e8  # módulo de elasticidad del acero, en kN/m²
A = 0.0020  # área de la sección, en m²
E * A
```

```
400000.0
```

El producto $EA = 400\,000\,\mathrm{kN}$ es la *rigidez axial* de la sección; la retomaremos en el capítulo 5.

2.4. El módulo `math`

Python trae incorporadas solo las operaciones básicas. Las funciones matemáticas, como la raíz cuadrada o las funciones trigonométricas, están en el *módulo* (*module*) `math`, que debemos importar una vez por notebook:

```
import math
```

A partir de ese momento, escribimos las funciones del módulo con el prefijo `math` seguido de un punto. La longitud de la barra es:

```
L = math.sqrt(dx**2 + dy**2)
L
```

```
5.0
```

Y sus cosenos directores:

```
c = dx / L
s = dy / L
c, s
```

```
(0.6, 0.8)
```

Para el ángulo usamos `math.atan2(dy, dx)`, que devuelve el ángulo en radianes, y `math.degrees`, que lo convierte a grados:

```
theta = math.degrees(math.atan2(dy, dx))
theta
```

```
53.13010235415598
```

> **Python**
>
> `math.atan2` recibe *dos* argumentos, primero Δy y después Δx, y por eso conoce el signo de cada uno. Así determina el cuadrante correcto y devuelve ángulos entre $-180°$ y $180°$. En el apartado 2.6 veremos qué ocurre si usamos en su lugar `math.atan(dy / dx)`.

2.5. Mostrar resultados con `print` y f-strings

El ángulo anterior aparece con catorce decimales, una precisión que ninguna medición justifica. Para presentar resultados usamos `print` junto con *f-strings* (*formatted string literals*): textos precedidos por una `f` en los que cada expresión entre llaves se reemplaza por su valor. Después de los dos puntos indicamos el formato; por ejemplo, `:.3f` significa «número con tres decimales»:

```
print(f"Longitud: L = {L:.3f} m")
print(f"Cosenos directores: c = {c:.4f}, s = {s:.4f}")
print(f"Ángulo: θ = {theta:.2f}°")
```

```
Longitud: L = 5.000 m
Cosenos directores: c = 0.6000, s = 0.8000
Ángulo: θ = 53.13°
```

De ahora en adelante, todo resultado que presentemos llevará sus unidades y una cantidad razonable de decimales: milímetros para los desplazamientos, tres cifras significativas para las fuerzas.

2.6. La geometría de una barra en Python

Reunimos todo lo anterior para obtener la geometría completa de una barra. Los pasos son siempre los mismos:

Estrategia de resolución

1. Identificar el nodo inicial i y el nodo final j de la barra.
2. Calcular las proyecciones Δx y Δy.
3. Calcular la longitud L y los cosenos directores c y s.
4. Calcular el ángulo θ con `math.atan2`.
5. Verificar: los signos de c y s deben coincidir con el dibujo, y debe cumplirse $c^2 + s^2 = 1$.

Problema resuelto 2.1 · Barra 2 del ejemplo A

La barra 2 del ejemplo A va del nodo 2, en $(6, 0)$, al nodo 3, en $(3, 4)$, con coordenadas en metros. Determinar su longitud, sus cosenos directores y su ángulo.

Solución

Las proyecciones son $\Delta x = 3 - 6 = -3\,\text{m}$ y $\Delta y = 4 - 0 = 4\,\text{m}$. En Python:

```
# Barra 2 del ejemplo A: del nodo 2 al nodo 3
x_i, y_i = 6.0, 0.0
x_j, y_j = 3.0, 4.0

dx = x_j - x_i
dy = y_j - y_i
L = math.sqrt(dx**2 + dy**2)
c = dx / L
s = dy / L
theta = math.degrees(math.atan2(dy, dx))

print(f"L = {L:.3f} m")
print(f"c = {c:.4f}, s = {s:.4f}")
print(f"θ = {theta:.2f}°")
```

```
L = 5.000 m
c = -0.6000, s = 0.8000
θ = 126.87°
```

Verificación. La barra sube hacia la izquierda, de modo que c debe ser negativo y s positivo, como obtuvimos. Además, $c^2 + s^2 = 0.36 + 0.64 = 1$.

El ángulo está en el segundo cuadrante, entre 90° y 180°, lo que también coincide con el dibujo.

Cuidado

Si para la misma barra usamos `math.atan(dy / dx)`, en lugar de `math.atan2(dy, dx)`, obtenemos:

```
math.degrees(math.atan(dy / dx))
```

```
-53.13010235415598
```

El cociente $\Delta y/\Delta x$ pierde los signos individuales, y `math.atan` solo devuelve ángulos entre $-90°$ y $90°$. Además, si la barra es vertical, $\Delta x = 0$ y la división produce un error. Por eso usamos siempre `math.atan2`.

Repaso y resumen del capítulo

- La geometría de una barra que va del nodo i al nodo j queda determinada por $L = \sqrt{\Delta x^2 + \Delta y^2}$, $c = \Delta x/L$ y $s = \Delta y/L$, con $\Delta x = x_j - x_i$ y $\Delta y = y_j - y_i$.
- El orden de los nodos define el sentido de la barra: intercambiarlos cambia los signos de c y s.
- Un notebook combina texto y código. Recuerda lo que se ejecutó, no lo que está escrito: ante dudas, *Ejecutar todas*.
- En Python, `=` asigna un valor a una variable; la potencia se escribe `**` y la notación científica `2.0e8`.
- El módulo `math` aporta `math.sqrt`, `math.atan2` y `math.degrees`, entre otras funciones.
- Los resultados se presentan con f-strings, con unidades y un número razonable de decimales: `f"{L:.3f} m"`.

Ejercicios

Los ejercicios con resultado numérico tienen una celda de verificación en el notebook del capítulo: al escribir allí el resultado y ejecutar la celda, se indica si es correcto. Las soluciones completas están en el anexo F y en el notebook de soluciones del capítulo.

Ejercicio 2.1 Calcular la longitud y los cosenos directores de la barra 1 del ejemplo A, que va del nodo 1, en $(0, 0)$, al nodo 2, en $(6, 0)$. ¿Por qué el resultado podía anticiparse sin calcular? Solución: p. 148

Ejercicio 2.2 Una barra del cordón superior de una armadura de cubierta va de $(0, 0)$ a $(4.5, 1.5)$, en metros. Calcular su longitud, sus cosenos directores y su ángulo en grados. Solución: p. 148

Ejercicio 2.3 La barra 3 del ejemplo A es de acero ($E = 200\,\text{GPa}$), tiene un área de $20\,\text{cm}^2$ y una longitud de $5\,\text{m}$. Calcular su rigidez axial EA/L en kN/m. Cuidar la conversión de unidades. Solución: p. 148

Ejercicio 2.4 Una barra va de $(6, 0)$ a $(0, 3)$. Calcular su ángulo con `math.atan(dy / dx)` y con `math.atan2(dy, dx)`. Explicar cuál es el correcto y por qué, apoyándose en un dibujo de la barra. Solución: p. 149

Ejercicio 2.5 Una barra vertical va de $(2, 3)$ a $(2, -1)$. Calcular su longitud, sus cosenos directores y su ángulo. ¿Qué ocurre si se intenta calcular el ángulo con `math.atan(dy / dx)`? Solución: p. 149

> **Punto de control**
>
> Si algo no funcionó o se omitió una parte, el notebook con todo el código hasta este capítulo permite continuar desde aquí:
>
> Repositorio:
> `github.com/eder3232/libro-rigidez-python-armaduras2d`
> Notebook: `notebooks/control_cap02.ipynb`

Parte II

Del modelo a la matriz

Capítulo 3

Describir una armadura

Antes de calcular cualquier cosa, nuestra calculadora necesita conocer la armadura: dónde están sus nodos, qué barras los unen, de qué material y sección son, cómo está apoyada y qué cargas recibe. En el capítulo anterior guardamos cada coordenada en una variable suelta; con tres barras eso ya resulta incómodo, y con cincuenta sería imposible. En este capítulo aprendemos a organizar toda esa información en estructuras de datos de Python, y construimos el modelo completo del ejemplo A.

Al terminar este capítulo podremos

- Identificar los datos que definen una armadura plana.
- Usar listas y diccionarios, y elegir entre ellos.
- Escribir el modelo de una armadura con diccionarios anidados.
- Consultar el modelo y dibujarlo con Matplotlib para verificarlo.

3.1. Los datos de una armadura

Recordemos las hipótesis con las que analizamos armaduras: las barras son rectas y se unen en nodos articulados, las cargas actúan solo en los nodos y el material es elástico lineal. Con ellas, cada barra trabaja únicamente a fuerza axial y la armadura queda definida por cuatro grupos de datos:

Nodos. Las coordenadas (x, y) de cada nodo.

Barras. El nodo inicial y el nodo final de cada barra, su módulo de elasticidad E y el área A de su sección.

Apoyos. En qué direcciones está impedido el desplazamiento de cada nodo apoyado: un apoyo articulado impide ambas; un rodillo, solo una.

Cargas. Las componentes F_x y F_y de las fuerzas aplicadas en los nodos.

La figura 3.1 y la tabla 3.1 reúnen estos datos para el ejemplo A, que nos acompañará en los próximos capítulos.

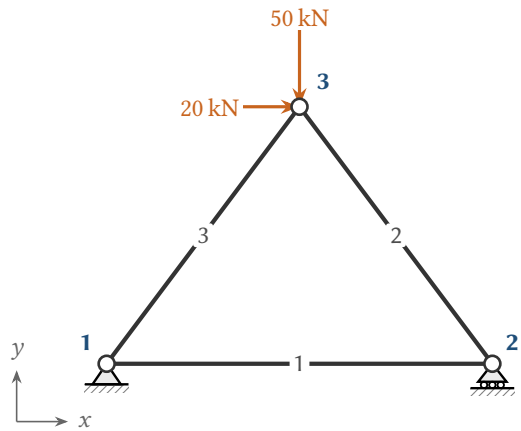


Figura 3.1: Ejemplo A: armadura triangular isostática.

Tabla 3.1: Datos del ejemplo A. Todas las barras son de acero ($E = 200\,\text{GPa}$) con un área de $20\,\text{cm}^2$.

Nodo	Coordenadas (m)	Apoyo	Carga (kN)
1	$(0, 0)$	articulado	—
2	$(6, 0)$	rodillo (impide y)	—
3	$(3, 4)$	—	$F_x = 20$, $F_y = -50$

Barra	Nodo inicial	Nodo final
1	1	2
2	2	3
3	1	3

3.2. Listas

Una *lista* (*list*) es una colección ordenada de valores, escrita entre corchetes y separada por comas. Por ejemplo, las coordenadas del nodo 3:

```
coordenadas_nodo_3 = [3.0, 4.0]
coordenadas_nodo_3
```

```
[3.0, 4.0]
```

Cada elemento ocupa una posición, llamada *índice* (*index*), y se obtiene escribiendo el índice entre corchetes:

```
coordenadas_nodo_3[0], coordenadas_nodo_3[1]
```

```
(3.0, 4.0)
```

> **Cuidado**
>
> En Python, las posiciones de una lista se cuentan *desde cero*: el primer elemento es `coordenadas_nodo_3[0]` y el segundo, `coordenadas_nodo_3[1]`. Es una convención de la mayoría de los lenguajes de programación, pero choca con la numeración de nodos y barras, que en ingeniería empieza en 1.

Ese desfase se vuelve peligroso cuando usamos una lista para guardar propiedades de las barras. Si guardamos las áreas de las barras 1, 2 y 3 en una lista (aquí con un área distinta en la barra 3, para notar la diferencia), el área de la barra 3 está en la posición 2:

```
# Áreas de las barras 1, 2 y 3, en m²
areas = [0.0020, 0.0020, 0.0015]
areas[2]
```

```
0.0015
```

Las listas pueden crecer: el método `append` agrega un elemento al final, y la función `len` cuenta los elementos:

```
areas.append(0.0030)
len(areas), areas
```

```
(4, [0.002, 0.002, 0.0015, 0.003])
```

3.3. Diccionarios

Un *diccionario* (*dictionary*) guarda pares *clave: valor*, entre llaves. A diferencia de la lista, cada valor se obtiene por su clave, no por su posición:

```
nodo_3 = {"x": 3.0, "y": 4.0}
nodo_3["y"]
```

```
4.0
```

Las claves pueden ser textos, como `"x"` e `"y"`, o números. Con números como claves resolvemos el problema de la sección anterior: el área de la barra 3 se obtiene con la clave 3, sin traducciones.

```
# Las claves son los números de barra, empezando en 1
areas = {1: 0.0020, 2: 0.0020, 3: 0.0015}
areas[3]
```

```
0.0015
```

> **Recuerda**
>
> Usamos *listas* para colecciones en las que solo importa el orden, y *diccionarios* cuando cada valor tiene un nombre o un número propio. Los nodos y las barras tienen número propio: por eso los guardaremos en diccionarios cuyas claves son esos números, empezando en 1.

3.4. El modelo del ejemplo A

Un valor de un diccionario puede ser, a su vez, otro diccionario. Con esta idea, los *diccionarios anidados* (*nested dictionaries*), escribimos el modelo completo. Primero los nodos, cada uno con sus coordenadas:

```
# Ejemplo A. Unidades: kN y m
nodos = {
    1: {"x": 0.0, "y": 0.0},
    2: {"x": 6.0, "y": 0.0},
    3: {"x": 3.0, "y": 4.0},
}
```

Después las barras, cada una con sus nodos y sus propiedades. Un diccionario puede ocupar varias líneas: Python sabe que continúa mientras la llave no se haya cerrado.

```
# E en kN/m² (acero) y área en m²
barras = {
    1: {"nodo_inicial": 1, "nodo_final": 2,
        "elasticidad": 2.0e8, "area": 0.0020},
    2: {"nodo_inicial": 2, "nodo_final": 3,
        "elasticidad": 2.0e8, "area": 0.0020},
    3: {"nodo_inicial": 1, "nodo_final": 3,
        "elasticidad": 2.0e8, "area": 0.0020},
}
```

Los apoyos indican, para cada nodo apoyado, si el desplazamiento en x y en y está impedido. Los valores `True` (verdadero) y `False` (falso) son los *booleanos* (*booleans*) de Python:

```
# True: el desplazamiento en esa dirección está impedido
apoyos = {
    1: {"x": True, "y": True},   # articulado
    2: {"x": False, "y": True},  # rodillo
}
```

Por último, las cargas, con su signo según los ejes globales:

```
# Cargas en kN, positivas en el sentido de los ejes
cargas = {
    3: {"x": 20.0, "y": -50.0},
}
```

Estrategia de resolución

Para escribir el modelo de cualquier armadura:

1. Dibujar la armadura, numerar los nodos y las barras desde 1 y elegir el sentido de cada barra (nodo inicial y nodo final).
2. Fijar las unidades (en este libro, kN y m) y convertir todos los datos a ellas.
3. Escribir los cuatro diccionarios: `nodos`, `barras`, `apoyos` y `cargas`.
4. Verificar el modelo consultándolo y dibujándolo.

Separar los apoyos y las cargas de los nodos tiene una ventaja: los nodos sin apoyo o sin carga simplemente no aparecen en esos diccionarios, sin entradas vacías que mantener.

3.5. Consultar el modelo

Para leer un valor anidado encadenamos las claves. Los nodos de la barra 2:

```
barras[2]["nodo_inicial"], barras[2]["nodo_final"]
```

```
(2, 3)
```

Y las coordenadas del nodo inicial de la barra 2, usando el número de nodo como clave del diccionario `nodos`:

```
inicial = barras[2]["nodo_inicial"]
nodos[inicial]["x"], nodos[inicial]["y"]
```

```
(6.0, 0.0)
```

Esta consulta encadenada, de la barra a sus nodos y de los nodos a sus coordenadas, es exactamente la que necesitaremos en el capítulo 4 para calcular la geometría de todas las barras.

El operador **`in`** pregunta si una clave existe en un diccionario. El nodo 3 no tiene apoyo, pero sí carga:

```
3 in apoyos, 3 in cargas
```

```
(False, True)
```

Pedir una clave que no existe, como `apoyos[3]`, produce el error `KeyError` y detiene el programa. El método `get` devuelve en cambio el valor **`None`** («nada»), o el valor que indiquemos:

```
print(apoyos.get(3))
print(apoyos.get(3, "sin apoyo"))
```

```
None
sin apoyo
```

Finalmente, `len` también cuenta las entradas de un diccionario:

```
print(f"Nodos: {len(nodos)}, barras: {len(barras)}")
```

```
Nodos: 3, barras: 3
```

3.6. Ver la armadura

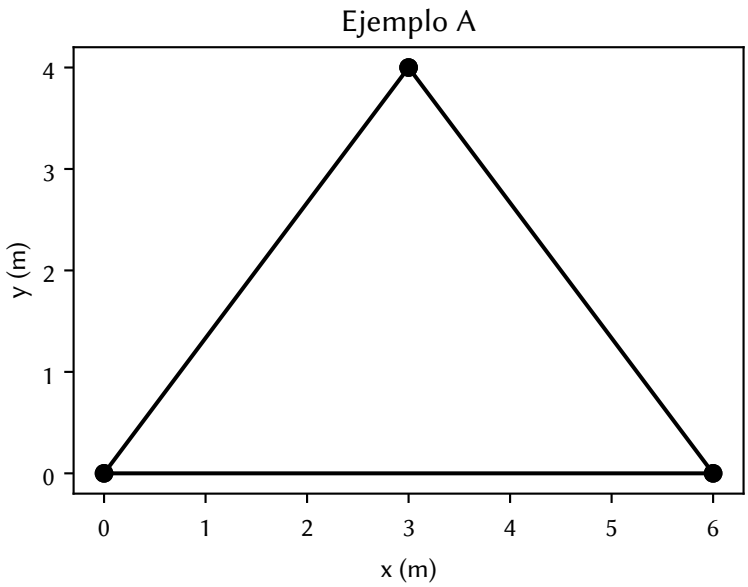
Un error en las coordenadas o en la conectividad es fácil de cometer y difícil de detectar leyendo números. La mejor verificación es *ver* la armadura. Para dibujar usamos *Matplotlib*, la biblioteca de gráficos más usada de Python, que importamos con el nombre corto `plt`:

```
import matplotlib.pyplot as plt
```

`plt.subplots()` crea una figura con unos ejes; `ax.plot` dibuja una línea entre los puntos cuyas abscisas y ordenadas recibe en dos listas, y el texto `"k-o"` pide línea negra (`k`) continua (`-`) con círculos en los extremos (`o`). Dibujamos así cada barra:

```
n1, n2, n3 = nodos[1], nodos[2], nodos[3]

fig, ax = plt.subplots()
# Barra 1: del nodo 1 al nodo 2
ax.plot([n1["x"], n2["x"]], [n1["y"], n2["y"]], "k-o")
# Barra 2: del nodo 2 al nodo 3
ax.plot([n2["x"], n3["x"]], [n2["y"], n3["y"]], "k-o")
# Barra 3: del nodo 1 al nodo 3
ax.plot([n1["x"], n3["x"]], [n1["y"], n3["y"]], "k-o")
ax.set_aspect("equal")
ax.set_xlabel("x (m)")
ax.set_ylabel("y (m)")
ax.set_title("Ejemplo A")
plt.show()
```



La instrucción `ax.set_aspect("equal")` usa la misma escala en ambos ejes, para que la armadura no se vea deformada.

Verificación. El dibujo coincide con la figura 3.1: tres nodos en $(0, 0)$, $(6, 0)$ y $(3, 4)$, unidos por tres barras.

> **Python**
>
> Escribimos una línea `ax.plot` por cada barra. Con tres barras es aceptable; con cincuenta sería tedioso y propenso a errores. En el capítulo 4 aprenderemos a recorrer el diccionario `barras` con un bucle y a dibujar cualquier armadura con unas pocas líneas.

Problema resuelto 3.1 · El modelo del ejemplo B

Escribir el modelo de datos del ejemplo B (figura 3.2): un cuadro de 3 m por 4 m con sus dos diagonales, articulado en el nodo 1 y con un rodillo en el nodo 2. Todas las barras son de acero, con un área de 20 cm^2. Verificar la cantidad de nodos y de barras.

Solución

Seguimos la estrategia de resolución: la numeración ya está en la figura, las unidades son kN y m, y los diccionarios llevan el sufijo `_b` para no reemplazar los del ejemplo A:

```
# Ejemplo B. Unidades: kN y m
nodos_b = {
```

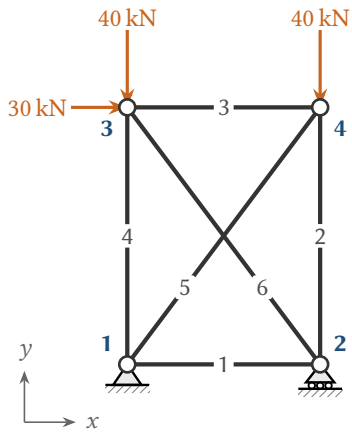


Figura 3.2: Ejemplo B: cuadro arriostrado hiperestático.

```
    1: {"x": 0.0, "y": 0.0},
    2: {"x": 3.0, "y": 0.0},
    3: {"x": 0.0, "y": 4.0},
    4: {"x": 3.0, "y": 4.0},
}

barras_b = {
    1: {"nodo_inicial": 1, "nodo_final": 2,
        "elasticidad": 2.0e8, "area": 0.0020},
    2: {"nodo_inicial": 2, "nodo_final": 4,
        "elasticidad": 2.0e8, "area": 0.0020},
    3: {"nodo_inicial": 3, "nodo_final": 4,
        "elasticidad": 2.0e8, "area": 0.0020},
    4: {"nodo_inicial": 1, "nodo_final": 3,
        "elasticidad": 2.0e8, "area": 0.0020},
    5: {"nodo_inicial": 1, "nodo_final": 4,
        "elasticidad": 2.0e8, "area": 0.0020},
    6: {"nodo_inicial": 2, "nodo_final": 3,
        "elasticidad": 2.0e8, "area": 0.0020},
}

apoyos_b = {
    1: {"x": True, "y": True},    # articulado
    2: {"x": False, "y": True},   # rodillo
}

cargas_b = {
    3: {"x": 30.0, "y": -40.0},
```

```
    4: {"x": 0.0, "y": -40.0},
}
```

Verificación. Contamos los nodos y las barras, y revisamos una de las diagonales:

```
inicial = barras_b[5]["nodo_inicial"]
final = barras_b[5]["nodo_final"]
print(f"Nodos: {len(nodos_b)}, barras: {len(barras_b)}")
print(f"Barra 5: del nodo {inicial} al nodo {final}")
```

```
Nodos: 4, barras: 6
Barra 5: del nodo 1 al nodo 4
```

La cantidad de nodos y de barras coincide con la figura, y la barra 5 es la diagonal que va del nodo 1 al nodo 4. Este modelo nos acompañará, junto con el del ejemplo A, en los próximos capítulos.

Repaso y resumen del capítulo

- Una armadura plana queda definida por sus nodos (coordenadas), sus barras (nodos, E y A), sus apoyos (direcciones impedidas) y sus cargas (componentes en los nodos).
- Una lista es una colección ordenada; sus posiciones se cuentan desde cero. Un diccionario asocia claves con valores.
- El modelo de la calculadora usa cuatro diccionarios: `nodos`, `barras`, `apoyos` y `cargas`, con los números de nodo y de barra como claves, empezando en 1.
- Los valores anidados se leen encadenando claves entre corchetes, de la barra a sus nodos y de los nodos a sus coordenadas.
- `in` pregunta si una clave existe; `get` lee una clave sin producir un error si no existe; `len` cuenta los elementos.
- Todo modelo se verifica dibujándolo con `plt.subplots` y `ax.plot`, con la misma escala en ambos ejes.

Ejercicios

Ejercicio 3.1 Usando los diccionarios del ejemplo A, obtener las coordenadas de los nodos de la barra 3 a partir de `barras[3]`, sin escribir los números de nodo a mano, y calcular la longitud de la barra. Solución: p. 149

Ejercicio 3.2 Una armadura de cubierta tiene los nodos 1 en $(0, 0)$, 2 en $(4, 0)$, 3 en $(8, 0)$ y 4 en $(4, 1.5)$, en metros. Sus barras son 1: 1–2, 2: 2–3, 3: 1–4, 4: 4–3 y 5: 2–4, todas de acero; las cuatro primeras tienen un área de $15\,\text{cm}^2$ y la barra 5, de $10\,\text{cm}^2$. Está articulada en el nodo 1, tiene un rodillo en el nodo 3 y una carga de 30 kN hacia abajo en el nodo 4. Escribir su modelo con los nombres `nodos_cubierta`, `barras_cubierta`, `apoyos_cubierta` y `cargas_cubierta`. Solución: p. 150

Ejercicio 3.3 Predecir, sin ejecutarlo, qué muestran `apoyos.get(3)` y `3 in cargas, 1 in cargas` para el ejemplo A. Comprobar la predicción en el notebook. Solución: p. 150

Ejercicio 3.4 Dibujar la armadura del ejemplo B con Matplotlib y comparar el dibujo con la figura 3.2. Solución: p. 151

> **Punto de control**
>
> Si algo no funcionó o se omitió una parte, el notebook con todo el código hasta este capítulo permite continuar desde aquí:
>
> Repositorio:
> `github.com/eder3232/libro-rigidez-python-armaduras2d`
> Notebook: `notebooks/control_cap03.ipynb`

Capítulo 4

La geometría de todas las barras

En el capítulo 2 calculamos la geometría de una barra, y en el capítulo 3 describimos la armadura completa con diccionarios. Ahora unimos ambas cosas: calculamos la longitud y los cosenos directores de *todas* las barras, sin importar cuántas sean. La herramienta que lo hace posible es el bucle, la instrucción que repite un mismo cálculo para cada elemento de una colección. Con ella, nuestra calculadora deja de depender del tamaño de la armadura.

Al terminar este capítulo podremos

- Repetir un cálculo con el bucle `for`.
- Recorrer un diccionario con `.items()` y acumular resultados.
- Calcular la geometría de todas las barras y guardarla en un diccionario nuevo, sin modificar los datos de entrada.
- Presentar los resultados en una tabla y dibujar cualquier armadura.

4.1. El mismo cálculo para cada barra

La geometría de una barra depende solo de las coordenadas de sus dos nodos:

$$L = \sqrt{\Delta x^2 + \Delta y^2}, \qquad c = \frac{\Delta x}{L}, \qquad s = \frac{\Delta y}{L}.$$

Para el ejemplo A, con tres barras, podríamos copiar tres veces el cálculo del capítulo 2. Pero una armadura de cubierta típica tiene entre 20 y 50 barras, y la de un puente, varios cientos. Copiar el cálculo sería tedioso y, sobre todo, una fuente segura de errores: basta con olvidar cambiar un número en una de las copias.

Lo que necesitamos es escribir el cálculo *una sola vez* y pedirle a Python que lo repita para cada barra del diccionario `barras`.

4.2. El bucle `for`

Un *bucle* (*loop*) **`for`** repite un bloque de instrucciones para cada elemento de una colección. En cada repetición, la variable que sigue a **`for`** toma el valor del elemento siguiente:

```
for numero in [1, 2, 3]:
    print(f"Barra {numero}")
```

```
Barra 1
Barra 2
Barra 3
```

> **Python**
> El bloque que se repite es el que está *sangrado* (desplazado hacia la derecha) debajo de la línea del **`for`**, que termina en dos puntos. Python usa la sangría, normalmente de cuatro espacios, para saber qué instrucciones pertenecen al bucle. Colab la agrega automáticamente al pulsar Enter después de los dos puntos; para salir del bloque basta con borrarla.

Si recorremos un diccionario, la variable toma, una por una, sus *claves*. Con cada clave podemos leer el valor correspondiente:

```
for numero in barras:
    print(numero, barras[numero]["area"])
```

```
1 0.002
2 0.002
3 0.002
```

Casi siempre necesitamos la clave y el valor a la vez. El método `.items()` entrega ambos en cada repetición, y los recibimos en dos variables separadas por una coma:

```
for numero, barra in barras.items():
    i = barra["nodo_inicial"]
    j = barra["nodo_final"]
    print(f"Barra {numero}: del nodo {i} al nodo {j}")
```

```
Barra 1: del nodo 1 al nodo 2
Barra 2: del nodo 2 al nodo 3
Barra 3: del nodo 1 al nodo 3
```

Un uso muy frecuente de los bucles es *acumular* un resultado. Empezamos con un valor inicial y, en cada repetición, le sumamos algo. El operador `+=` suma al valor que la variable ya tenía: `a += b` equivale a `a = a + b`. Así obtenemos, por ejemplo, la longitud total de las barras:

```
longitud_total = 0.0
for numero, barra in barras.items():
    inicial = nodos[barra["nodo_inicial"]]
    final = nodos[barra["nodo_final"]]
    dx = final["x"] - inicial["x"]
    dy = final["y"] - inicial["y"]
    longitud_total += math.sqrt(dx**2 + dy**2)
print(f"Longitud total: {longitud_total:.3f} m")
```

```
Longitud total: 16.000 m
```

4.3. La geometría de todas las barras

Ya tenemos todas las piezas. Recorremos las barras; para cada una, buscamos sus nodos en el diccionario `nodos`, calculamos las proyecciones, la longitud y los cosenos directores, y guardamos el resultado en un diccionario nuevo, `geometria`, con el número de barra como clave:

```
geometria = {}
for numero, barra in barras.items():
    inicial = nodos[barra["nodo_inicial"]]
    final = nodos[barra["nodo_final"]]
    dx = final["x"] - inicial["x"]
    dy = final["y"] - inicial["y"]
    L = math.sqrt(dx**2 + dy**2)
    geometria[numero] = {"longitud": L,
                         "c": dx / L, "s": dy / L}
```

Este código forma parte de nuestra calculadora: es la primera pieza que funciona para *cualquier* armadura descrita con los diccionarios del capítulo 3.

> **Cuidado**
>
> Guardamos los resultados en un diccionario nuevo, en lugar de agregarlos a `barras`. Así los datos de entrada nunca cambian: si algo sale mal, podemos volver a ejecutar el cálculo con la seguridad de partir de los mismos datos. Mezclar datos y resultados en la misma estructura es una causa frecuente de errores difíciles de rastrear.

Para presentar los resultados recorremos `geometria` e imprimimos una fila por barra. En las f-strings, el número antes del punto indica el *ancho* de la columna: `:7.3f` reserva siete caracteres con tres decimales, y `:>5` alinea el número de barra a la derecha en cinco caracteres. Así las columnas quedan alineadas:

```
print("Barra   L (m)       c       s")
for numero, g in geometria.items():
```

```
    L, c, s = g["longitud"], g["c"], g["s"]
    print(f"{numero:>5} {L:7.3f} {c:7.4f} {s:7.4f}")
```

```
Barra   L (m)       c       s
    1   6.000  1.0000  0.0000
    2   5.000 -0.6000  0.8000
    3   5.000  0.6000  0.8000
```

4.4. Verificación

Los valores coinciden con los que calculamos a mano en el capítulo 2: la barra 1 es horizontal ($c = 1$, $s = 0$), y las barras 2 y 3 son las hipotenusas de dos triángulos 3-4-5, con $L = 5$, $s = 0.8$ y $c = \pm 0.6$. El signo de c en la barra 2 es negativo porque va del nodo 2 al nodo 3, hacia la izquierda.

Una verificación que no depende del ejemplo es que, para toda barra, debe cumplirse $c^2 + s^2 = 1$:

```
for numero, g in geometria.items():
    suma = g["c"] ** 2 + g["s"] ** 2
    print(f"Barra {numero}: c² + s² = {suma:.12f}")
```

```
Barra 1: c² + s² = 1.000000000000
Barra 2: c² + s² = 1.000000000000
Barra 3: c² + s² = 1.000000000000
```

Finalmente, comprobamos que los datos de entrada no cambiaron: la barra 1 conserva sus cuatro datos originales y nada más.

```
barras[1]
```

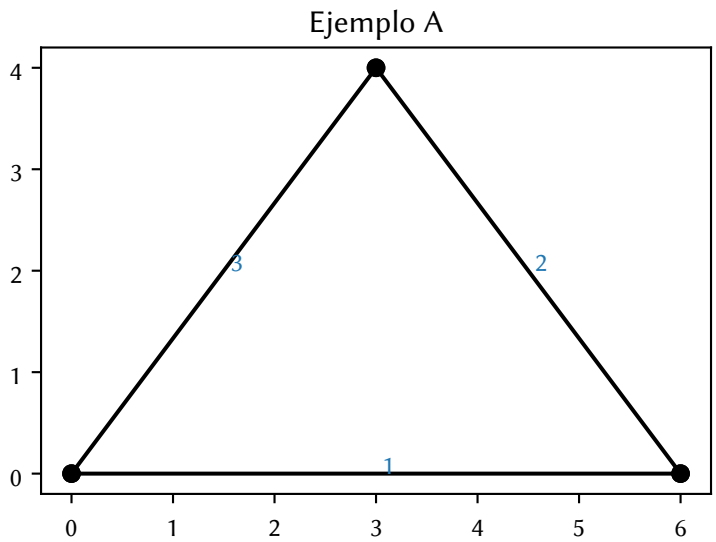
```
{'nodo_inicial': 1, 'nodo_final': 2, 'elasticidad':
 ↪  200000000.0, 'area': 0.002}
```

4.5. Dibujar cualquier armadura

En el capítulo 3 escribimos una línea `ax.plot` por cada barra. Con un bucle, el mismo dibujo funciona para cualquier armadura. Aprovechamos además para escribir el número de cada barra en su punto medio, con `ax.text(x, y, texto)`:

```
fig, ax = plt.subplots()
for numero, barra in barras.items():
    inicial = nodos[barra["nodo_inicial"]]
    final = nodos[barra["nodo_final"]]
    ax.plot([inicial["x"], final["x"]],
```

```
            [inicial["y"], final["y"]], "k-o")
    # Número de la barra en su punto medio
    xm = (inicial["x"] + final["x"]) / 2
    ym = (inicial["y"] + final["y"]) / 2
    ax.text(xm, ym, f" {numero}", color="tab:blue")
ax.set_aspect("equal")
ax.set_title("Ejemplo A")
plt.show()
```



Problema resuelto 4.1 · La geometría del ejemplo B

Calcular la longitud y los cosenos directores de las seis barras del ejemplo B (figura 3.2) y verificar los resultados.

Solución

Aplicamos el mismo bucle a los diccionarios `nodos_b` y `barras_b` del capítulo 3, y guardamos los resultados en `geometria_b`:

```
geometria_b = {}
for numero, barra in barras_b.items():
    inicial = nodos_b[barra["nodo_inicial"]]
    final = nodos_b[barra["nodo_final"]]
    dx = final["x"] - inicial["x"]
    dy = final["y"] - inicial["y"]
    L = math.sqrt(dx**2 + dy**2)
    geometria_b[numero] = {"longitud": L,
                           "c": dx / L, "s": dy / L}
```

```
print("Barra   L (m)       c       s")
for numero, g in geometria_b.items():
    L, c, s = g["longitud"], g["c"], g["s"]
    print(f"{numero:>5} {L:7.3f} {c:7.4f} {s:7.4f}")
```

```
Barra   L (m)       c       s
    1   3.000  1.0000  0.0000
    2   4.000  0.0000  1.0000
    3   3.000  1.0000  0.0000
    4   4.000  0.0000  1.0000
    5   5.000  0.6000  0.8000
    6   5.000 -0.6000  0.8000
```

Verificación. Las barras 1 y 3 son horizontales de 3 m y las barras 2 y 4, verticales de 4 m. Las diagonales 5 y 6 miden 5 m; la diagonal 5 sube hacia la derecha ($c > 0$) y la 6, hacia la izquierda ($c < 0$), como se ve en la figura.

Para resolver el problema copiamos el bucle y cambiamos los nombres de los diccionarios. Funciona, pero es justamente la clase de repetición que queríamos evitar. En el capítulo 5 aprenderemos a darle un nombre a un cálculo, con una *función*, para usarlo con cualquier armadura sin copiarlo.

Repaso y resumen del capítulo

- El bucle `for` repite el bloque sangrado debajo de él para cada elemento de una colección.
- Al recorrer un diccionario, `for` entrega sus claves; `.items()` entrega la clave y el valor a la vez.
- Un acumulador empieza con un valor inicial y crece en cada repetición con `+=`.
- La geometría de todas las barras se guarda en un diccionario nuevo, `geometria`, con el número de barra como clave. Los datos de entrada no se modifican.
- En las f-strings, `:7.3f` fija el ancho y los decimales de una columna, y `:>5` alinea a la derecha.
- Para toda barra debe cumplirse $c^2 + s^2 = 1$.

Ejercicios

Ejercicio 4.1 Modificar el bucle de la geometría para calcular también el ángulo θ de cada barra del ejemplo A, en grados, y mostrarlo para las tres barras. Solución: p. 152

Ejercicio 4.2 El peso específico del acero es $\gamma = 78.5\,\text{kN/m}^3$. Calcular el peso propio total de las barras del ejemplo A, sabiendo que el peso de una barra es $\gamma A L$. Solución: p. 152

Ejercicio 4.3 Calcular la longitud total de las barras del ejemplo B con un acumulador. Solución: p. 152

Ejercicio 4.4 Dibujar la armadura del ejemplo B con un bucle, numerando sus barras, y comparar el dibujo con la figura 3.2. Solución: p. 153

> **Punto de control**
>
> Si algo no funcionó o se omitió una parte, el notebook con todo el código hasta este capítulo permite continuar desde aquí:
>
> Repositorio:
> `github.com/eder3232/libro-rigidez-python-armaduras2d`
> Notebook: `notebooks/control_cap04.ipynb`

Capítulo 5

La rigidez de una barra

Con la geometría de todas las barras resuelta, entramos al corazón del método: la matriz de rigidez de cada barra, que relaciona las fuerzas en sus extremos con los desplazamientos de sus nodos. Para escribirla necesitamos dos herramientas nuevas. Las *funciones*, que nos permiten dar nombre a un cálculo y reutilizarlo sin copiarlo, y *NumPy*, la biblioteca que convierte a Python en una herramienta de cálculo matricial. Al final del capítulo, nuestra calculadora tendrá la matriz de rigidez de cada barra de cualquier armadura.

Al terminar este capítulo podremos

- Definir y usar funciones con parámetros y valor de retorno.
- Crear arreglos de NumPy y operar con ellos: producto por un escalar, producto matricial y transpuesta.
- Construir la matriz de rigidez de una barra en ejes globales, $\mathbf{k} = \mathbf{T}^{\mathsf{T}}\mathbf{k}'\mathbf{T}$.
- Verificar que la matriz es simétrica, singular y que no produce fuerzas ante un movimiento de cuerpo rígido.

5.1. La matriz de rigidez de una barra

Una barra de armadura solo trabaja a fuerza axial. En su propio eje x', los desplazamientos de sus extremos $\mathbf{u}' = [u'_1,\ u'_2]^{\mathsf{T}}$ y las fuerzas axiales en ellos $\mathbf{f}' = [f'_1,\ f'_2]^{\mathsf{T}}$ se relacionan por la ley de Hooke, con la *matriz de rigidez local* (*local stiffness matrix*):

$$\mathbf{f}' = \mathbf{k}'\,\mathbf{u}', \qquad \mathbf{k}' = \frac{EA}{L}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}. \tag{5.1}$$

Los nodos, en cambio, se desplazan en las direcciones x e y globales: cada barra tiene cuatro *grados de libertad* (*degrees of freedom*), $\mathbf{u} = [u_1,\ u_2,\ u_3,\ u_4]^{\mathsf{T}}$ (figura 5.1). El desplazamiento de cada extremo en la dirección de la barra es

la proyección de su desplazamiento global sobre el eje x', lo que se escribe con la *matriz de transformación* (*transformation matrix*):

$$\mathbf{u}' = \mathbf{T}\,\mathbf{u}, \qquad \mathbf{T} = \begin{bmatrix} c & s & 0 & 0 \\ 0 & 0 & c & s \end{bmatrix}. \tag{5.2}$$

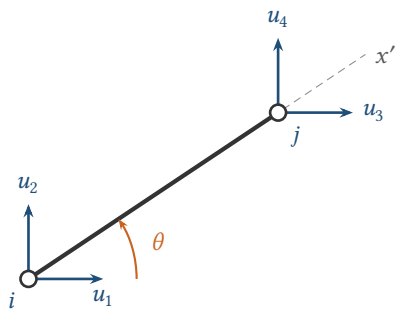


Figura 5.1: Grados de libertad de una barra en ejes globales y su eje local x'.

Como las fuerzas se transforman con la transpuesta, $\mathbf{f} = \mathbf{T}^\mathsf{T}\mathbf{f}'$, llegamos a la matriz de rigidez de la barra en ejes globales:

$$\mathbf{k} = \mathbf{T}^\mathsf{T}\,\mathbf{k}'\,\mathbf{T} = \frac{EA}{L}\begin{bmatrix} c^2 & cs & -c^2 & -cs \\ cs & s^2 & -cs & -s^2 \\ -c^2 & -cs & c^2 & cs \\ -cs & -s^2 & cs & s^2 \end{bmatrix}$$

> **Para saber más**
>
> La deducción completa, a partir del equilibrio de la barra y de la compatibilidad de desplazamientos, está en Kassimali (2012, cap. 3).

Podríamos programar directamente la forma desarrollada, término a término. Preferimos, sin embargo, programar el producto $\mathbf{T}^\mathsf{T}\mathbf{k}'\mathbf{T}$: es más corto, se parece más a la teoría y, como veremos en los próximos capítulos, las mismas matrices $\mathbf{T}$ y $\mathbf{k}'$ nos servirán para calcular las fuerzas internas.

5.2. Funciones

Una *función* (*function*) es un cálculo con nombre. Se define con `def`, seguido del nombre, de los *parámetros* (*parameters*) entre paréntesis y de dos puntos. El cuerpo va sangrado, como en un bucle, y la instrucción `return` indica el resultado:

```
def rigidez_axial(E, A, L):
    """Rigidez axial EA/L de una barra, en kN/m."""
    return E * A / L


rigidez_axial(2.0e8, 0.0020, 5.0)
```

```
80000.0
```

Definir la función no calcula nada: solo le enseña a Python cómo hacerlo. El cálculo ocurre cuando la *llamamos*, escribiendo su nombre con los valores de los parámetros, los *argumentos* (*arguments*), entre paréntesis. El texto entre triples comillas, debajo de la primera línea, es la *documentación* (*docstring*) de la función: explica qué hace y en qué unidades.

> **Python**
>
> Las variables creadas dentro de una función, como el cálculo intermedio que haga, son *locales*: existen solo mientras la función se ejecuta y no interfieren con las variables del resto del notebook. Una función se comunica con el exterior únicamente a través de sus parámetros y de su `return`.

Con una función resolvemos el problema que dejamos pendiente en el capítulo 4: el bucle de la geometría estaba escrito para los diccionarios `nodos` y `barras`, y para el ejemplo B tuvimos que copiarlo. Convertido en función, recibe los diccionarios como parámetros:

```
def calcular_geometria(nodos, barras):
    """Longitud y cosenos directores de cada barra.

    Devuelve un diccionario {número de barra:
    {"longitud": L, "c": c, "s": s}}.
    """
    geometria = {}
    for numero, barra in barras.items():
        inicial = nodos[barra["nodo_inicial"]]
        final = nodos[barra["nodo_final"]]
        dx = final["x"] - inicial["x"]
        dy = final["y"] - inicial["y"]
        L = math.sqrt(dx**2 + dy**2)
        geometria[numero] = {"longitud": L,
                             "c": dx / L, "s": dy / L}
    return geometria
```

La misma función sirve ahora para cualquier armadura:

```
geometria = calcular_geometria(nodos, barras)
geometria_b = calcular_geometria(nodos_b, barras_b)
geometria_b[6]
```

```
{'longitud': 5.0, 'c': -0.6, 's': 0.8}
```

5.3. NumPy

Las listas de Python no saben multiplicar matrices. Para eso usamos *NumPy*, la biblioteca de cálculo numérico sobre la que se apoya casi todo el Python científico. Se importa con el nombre corto `np`, y con `np.set_printoptions` pedimos que muestre los resultados con un decimal y sin notación científica:

```
import numpy as np

np.set_printoptions(precision=1, suppress=True)
```

El objeto central de NumPy es el *arreglo* (*array*), que creamos con `np.array` a partir de una lista de filas. Su atributo `shape` indica sus dimensiones:

```
k_local = np.array([[1, -1],
                    [-1, 1]])
k_local.shape
```

```
(2, 2)
```

Las operaciones con un número se aplican a cada elemento. Así obtenemos $\mathbf{k}'$ para una barra con $EA/L = 80\,000\,\text{kN/m}$:

```
80000 * k_local
```

```
array([[ 80000, -80000],
       [-80000,  80000]])
```

El *producto matricial* se escribe con el operador `@`, y la transpuesta de un arreglo `T`, con `T.T`. Comprobemos la ecuación (5.2) con la barra 3 del ejemplo A, cuyo nodo final se desplaza 3 mm en x y 4 mm en y:

```
# Barra 3 del ejemplo A: c = 0.6, s = 0.8
T = np.array([[0.6, 0.8, 0.0, 0.0],
              [0.0, 0.0, 0.6, 0.8]])
# El nodo final se mueve 3 mm en x y 4 mm en y
u = np.array([0.0, 0.0, 3.0, 4.0])
print(T.shape, u.shape, T.T.shape)
T @ u
```

```
(2, 4) (4,) (4, 2)
array([0., 5.])
```

El nodo final se aleja 5 mm del inicial en la dirección de la barra: el desplazamiento $(3, 4)$ es paralelo a ella, y su módulo es 5.

> **Cuidado**
>
> En NumPy, `*` multiplica elemento por elemento y `@` hace el producto matricial. Confundirlos es uno de los errores más frecuentes: `A * B` no es el producto de matrices de álgebra lineal. Además, las dimensiones deben ser compatibles: `T @ u` funciona porque `T` es de 2×4 y `u` tiene 4 elementos.

5.4. La matriz de rigidez en Python

Traducimos la teoría del apartado 5.1 a tres funciones, una por cada matriz. Cada una es corta y hace una sola cosa, lo que facilita leerlas y verificarlas por separado:

```
def rigidez_local(E, A, L):
    """Matriz de rigidez de la barra en sus ejes (2x2)."""
    return E * A / L * np.array([[1.0, -1.0],
                                 [-1.0, 1.0]])


def matriz_transformacion(c, s):
    """Matriz T (2 x 4) tal que u' = T u."""
    return np.array([[c, s, 0.0, 0.0],
                     [0.0, 0.0, c, s]])


def rigidez_barra(E, A, L, c, s):
    """Matriz de rigidez de la barra en ejes globales."""
    T = matriz_transformacion(c, s)
    return T.T @ rigidez_local(E, A, L) @ T
```

La última línea es casi idéntica a la ecuación clave: $\mathbf{T}^{\mathrm{T}}\mathbf{k}'\mathbf{T}$. Probemos con la barra 1 del ejemplo A, que es horizontal:

```
g = geometria[1]
k1 = rigidez_barra(2.0e8, 0.0020, g["longitud"],
                   g["c"], g["s"])
k1
```

```
array([[ 66666.7,      0. , -66666.7,      0. ],
       [     0. ,      0. ,      0. ,      0. ],
       [-66666.7,      0. ,  66666.7,      0. ],
       [     0. ,      0. ,      0. ,      0. ]])
```

Con $c = 1$ y $s = 0$, solo sobreviven los términos en c^2, que valen $EA/L = 2.0 \times 10^8 \cdot 0.0020/6 = 66\,666.7\ \mathrm{kN/m}$: una barra horizontal no ofrece rigidez a los desplazamientos verticales de sus nodos.

Para obtener las matrices de todas las barras, envolvemos el bucle en una función más, que guarda cada matriz en un diccionario con el número de barra como clave:

```
def calcular_rigideces(barras, geometria):
    """Matriz de rigidez global (4 x 4) de cada barra."""
    rigideces = {}
    for numero, barra in barras.items():
        g = geometria[numero]
        rigideces[numero] = rigidez_barra(
            barra["elasticidad"], barra["area"],
            g["longitud"], g["c"], g["s"])
    return rigideces


geometria = calcular_geometria(nodos, barras)
rigideces = calcular_rigideces(barras, geometria)
```

Las funciones `calcular_geometria` y `calcular_rigideces` son las dos primeras etapas de nuestra calculadora. A partir de ahora, los datos fluyen de una a otra: los diccionarios del modelo entran a la primera, y su resultado, junto con las barras, entra a la segunda.

5.5. Verificación

Toda matriz de rigidez de una barra debe cumplir tres propiedades, que podemos comprobar con NumPy sin hacer ningún cálculo a mano:

Simetría. $\mathbf{k} = \mathbf{k}^{\mathsf{T}}$. La función `np.allclose` compara dos arreglos admitiendo las pequeñas diferencias del redondeo.

Singularidad. Una barra suelta puede moverse sin deformarse, así que $\mathbf{k}$ no se puede invertir. Su *rango* (*rank*), la cantidad de filas independientes, es 1: todas las filas son múltiplos de la misma.

Cuerpo rígido. Si la barra se traslada sin deformarse, no aparecen fuerzas: $\mathbf{k}\,\mathbf{u} = \mathbf{0}$.

```
for numero, k in rigideces.items():
    simetrica = np.allclose(k, k.T)
    rango = np.linalg.matrix_rank(k)
    print(f"Barra {numero}: simétrica = {simetrica}, "
          f"rango = {rango}")
```

```
Barra 1: simétrica = True, rango = 1
Barra 2: simétrica = True, rango = 1
Barra 3: simétrica = True, rango = 1
```

```
# Traslación de la barra 2 como cuerpo rígido: 1 mm en x
traslacion = np.array([0.001, 0.0, 0.001, 0.0])
rigideces[2] @ traslacion
```

```
array([0., 0., 0., 0.])
```

La singularidad no es un defecto: es la expresión matemática de que una barra sola no es una estructura. En el capítulo 8 veremos cómo los apoyos la eliminan.

Problema resuelto 5.1 · La barra 2 del ejemplo A

Calcular a mano los términos de la matriz de rigidez global de la barra 2 del ejemplo A y compararlos con el resultado de `rigidez_barra`.

Solución

La barra 2 tiene $L = 5\,\text{m}$, $c = -0.6$ y $s = 0.8$, de modo que $EA/L = 2.0 \times 10^8 \cdot 0.0020/5 = 80\,000\,\text{kN/m}$. Los tres términos distintos de la ecuación clave son:

```
g = geometria[2]
EA_L = 2.0e8 * 0.0020 / g["longitud"]
c, s = g["c"], g["s"]
print(f"EA/L = {EA_L:.0f} kN/m, c = {c:.1f}, s = {s:.1f}")
print(f"EA/L·c²  = {EA_L * c**2:9.1f}")
print(f"EA/L·cs  = {EA_L * c * s:9.1f}")
print(f"EA/L·s²  = {EA_L * s**2:9.1f}")
rigideces[2]
```

```
EA/L = 80000 kN/m, c = -0.6, s = 0.8
EA/L·c²  =   28800.0
EA/L·cs  =  -38400.0
EA/L·s²  =   51200.0
array([[ 28800., -38400., -28800.,  38400.],
       [-38400.,  51200.,  38400., -51200.],
       [-28800.,  38400.,  28800., -38400.],
       [ 38400., -51200., -38400.,  51200.]])
```

Verificación. Cada término de la matriz coincide, con su signo, con el patrón de la ecuación clave: c^2 y s^2 en la diagonal, y cs negativo porque la barra sube hacia la izquierda.

Repaso y resumen del capítulo

- La matriz de rigidez de una barra en ejes globales es $\mathbf{k} = \mathbf{T}^{\mathsf{T}}\mathbf{k}'\mathbf{T}$, con $\mathbf{k}'$ de 2×2 y $\mathbf{T}$ de 2×4.

- Una función se define con `def`, recibe parámetros y devuelve un resultado con `return`. Sus variables internas son locales.
- Con funciones, el mismo cálculo sirve para cualquier armadura: basta con llamar a `calcular_geometria` con otros diccionarios.
- NumPy aporta los arreglos: `np.array`, `shape`, el producto matricial `@` y la transpuesta `.T`. El operador `*` multiplica elemento por elemento.
- La matriz de una barra es simétrica, de rango 1 y no produce fuerzas ante un movimiento de cuerpo rígido.

Ejercicios

Ejercicio 5.1 Escribir una función `peso_barra(A, L, gamma=78.5)` que devuelva el peso propio de una barra, en kN, y usarla para calcular el peso del ejemplo A. El valor después del signo igual es un *valor por omisión*: se usa si al llamar la función no se indica otro. Solución: p. 153

Ejercicio 5.2 Mover el nodo final de la barra 3 del ejemplo A 1 mm en la dirección perpendicular a la barra, es decir, en la dirección $(-s, c)$, y calcular las fuerzas que aparecen con $\mathbf{k}\,\mathbf{u}$. Explicar el resultado. Solución: p. 154

Ejercicio 5.3 Calcular la matriz de rigidez global de una barra vertical de 3 m, de acero y con 10 cm^2 de área, que va del nodo $(0, 0)$ al nodo $(0, 3)$. ¿Qué términos son distintos de cero y por qué? Solución: p. 154

Ejercicio 5.4 Calcular las matrices de rigidez de las seis barras del ejemplo B con las funciones del capítulo y verificar que todas son simétricas y de rango 1. Solución: p. 155

> **Punto de control**
>
> Si algo no funcionó o se omitió una parte, el notebook con todo el código hasta este capítulo permite continuar desde aquí:
>
> Repositorio:
> `github.com/eder3232/libro-rigidez-python-armaduras2d`
> Notebook: `notebooks/control_cap05.ipynb`

Capítulo 6

Los grados de libertad

Cada barra tiene ya su matriz de rigidez de 4×4, escrita en función de los desplazamientos de sus dos nodos. Para unirlas en una sola matriz de la armadura necesitamos un sistema de referencia común: una numeración de todos los desplazamientos posibles de la estructura, sus *grados de libertad*. En este capítulo los numeramos, los separamos en libres y restringidos según los apoyos y hacemos una primera revisión de la estabilidad. Por el camino, enfrentamos una diferencia incómoda entre la ingeniería y Python: nosotros contamos desde 1, y Python, desde 0.

Al terminar este capítulo podremos

- Numerar los grados de libertad de una armadura desde 1, nodo por nodo.
- Tomar decisiones en el código con `if`, `elif` y `else`.
- Clasificar los grados de libertad en libres y restringidos.
- Traducir la numeración desde 1 a los índices de NumPy en un único lugar del código.
- Calcular el grado de indeterminación de una armadura.

6.1. Los grados de libertad de la armadura

En una armadura plana, cada nodo puede desplazarse en dos direcciones, x e y. Una armadura de n nodos tiene, por lo tanto, $2n$ grados de libertad. Los numeramos nodo por nodo, siempre en el mismo orden: primero x y después y. Así, el nodo i tiene los grados de libertad

$$\mathrm{gdl}_x = 2i - 1, \qquad \mathrm{gdl}_y = 2i.$$

Cada grado de libertad es de uno de dos tipos (figura 6.1):

Libre. El nodo puede desplazarse en esa dirección. El desplazamiento es una

incógnita y la fuerza, la carga aplicada, es un dato.

Restringido. Un apoyo impide el desplazamiento. El desplazamiento es un dato (nulo, salvo asentamientos) y la fuerza, la reacción, es una incógnita.

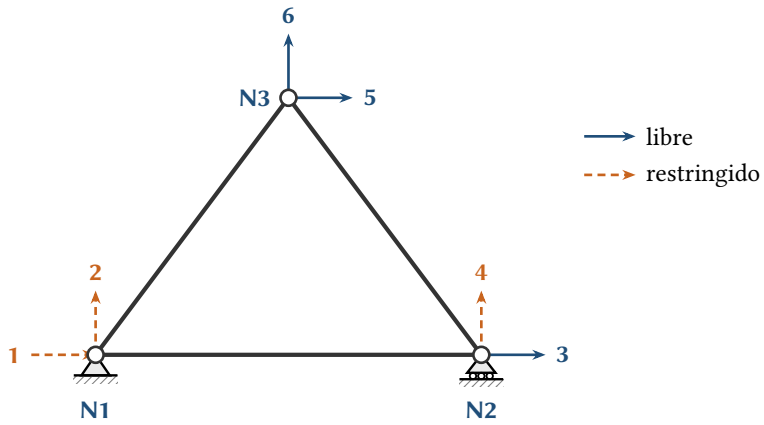


Figura 6.1: Grados de libertad del ejemplo A. El nodo 1 (articulado) tiene ambos restringidos; el nodo 2 (rodillo), solo el vertical.

Para cada barra necesitamos además saber a qué grados de libertad de la armadura corresponden sus cuatro grados de libertad locales: los dos de su nodo inicial y los dos de su nodo final. Esta lista, que algunos textos llaman *vector de correspondencia* (*code numbers*), es la que nos permitirá colocar cada matriz de 4×4 en el lugar correcto de la matriz de la armadura en el capítulo 7.

6.2. Decisiones con `if`

Para clasificar un grado de libertad debemos preguntar si el nodo tiene un apoyo en esa dirección y actuar según la respuesta. Las preguntas se escriben con *comparaciones* (*comparisons*): `>` y `<`, `>=` y `<=`, `==` (igual a, con dos signos) y `!=` (distinto de). Su resultado es un booleano:

```
area = barras[1]["area"]
print(area > 0.001, len(barras) == 3, len(nodos) != 3)
```

```
True True False
```

> **Cuidado**
> Un solo signo igual asigna un valor; dos signos iguales comparan. Escribir **`if`**

> `grado = 0:` en lugar de `if grado == 0:` es un error de sintaxis frecuente al empezar.

La instrucción `if` ejecuta un bloque solo si la condición es verdadera, y `else`, el bloque alternativo. Como en los bucles y las funciones, los bloques van sangrados debajo de una línea que termina en dos puntos:

```
apoyo = apoyos[2]
if apoyo["x"]:
    print("El nodo 2 no puede moverse en x")
else:
    print("El nodo 2 puede moverse en x")
```

```
El nodo 2 puede moverse en x
```

Cuando hay más de dos casos, `elif` (contracción de *else if*) agrega condiciones intermedias. Lo usaremos en el apartado 6.6.

6.3. La numeración de los grados de libertad

La ecuación clave se traduce en una función de una línea. Una función puede devolver varios valores separados por comas, y quien la llama los recibe en varias variables:

```
def gdl_del_nodo(nodo):
    """Grados de libertad (x, y) del nodo, desde 1."""
    return 2 * nodo - 1, 2 * nodo
```

```
gdl_x, gdl_y = gdl_del_nodo(3)
print(f"Nodo 3: x → {gdl_x}, y → {gdl_y}")
```

```
Nodo 3: x → 5, y → 6
```

Con ella obtenemos el vector de correspondencia de una barra: los grados de libertad de su nodo inicial seguidos de los de su nodo final, en el mismo orden que $u_1, \ldots, u_4$ en la figura 5.1.

```
def gdl_de_barra(barra):
    """Grados de libertad de los extremos de una barra."""
    xi, yi = gdl_del_nodo(barra["nodo_inicial"])
    xj, yj = gdl_del_nodo(barra["nodo_final"])
    return [xi, yi, xj, yj]
```

```
for numero, barra in barras.items():
    print(f"Barra {numero}: {gdl_de_barra(barra)}")
```

```
Barra 1: [1, 2, 3, 4]
Barra 2: [3, 4, 5, 6]
Barra 3: [1, 2, 5, 6]
```

Verificación. La barra 2 va del nodo 2 al nodo 3, cuyos grados de libertad son (3, 4) y (5, 6), como se ve en la figura 6.1.

> **Cuidado**
>
> La fórmula $2i-1, 2i$ supone que los nodos están numerados de forma consecutiva desde 1, sin saltos. Si un modelo tuviera los nodos 1, 2 y 5, el nodo 5 recibiría los grados de libertad 9 y 10, y la armadura parecería tener más grados de libertad de los que tiene. En este libro numeramos siempre los nodos de forma consecutiva.

6.4. Grados de libertad libres y restringidos

Para clasificar, recorremos los nodos y preguntamos, en cada dirección, si hay un apoyo. Un nodo sin apoyo no aparece en el diccionario `apoyos`: el método `get`, que vimos en el capítulo 3, nos permite tratarlo como un nodo «sin restricciones». Cada grado de libertad se agrega con `append` a la lista que le corresponde:

```python
def clasificar_gdl(nodos, apoyos):
    """Grados de libertad libres y restringidos."""
    libres = []
    restringidos = []
    sin_apoyo = {"x": False, "y": False}
    for nodo in nodos:
        gdl_x, gdl_y = gdl_del_nodo(nodo)
        apoyo = apoyos.get(nodo, sin_apoyo)
        if apoyo["x"]:
            restringidos.append(gdl_x)
        else:
            libres.append(gdl_x)
        if apoyo["y"]:
            restringidos.append(gdl_y)
        else:
            libres.append(gdl_y)
    return libres, restringidos
```

```python
libres, restringidos = clasificar_gdl(nodos, apoyos)
```

```python
print(f"Libres:       {libres}")
print(f"Restringidos: {restringidos}")
```

```
Libres:       [3, 5, 6]
Restringidos: [1, 2, 4]
```

Verificación. El nodo 1 aporta sus dos grados de libertad, 1 y 2, a los restringidos, y el rodillo del nodo 2, el vertical (4). Los libres son el horizontal del nodo 2 y los dos del nodo 3. La suma, $3 + 3 = 6 = 2n$, da todos los grados de libertad.

6.5. De la numeración desde 1 a los índices de Python

En el capítulo 3 vimos que Python cuenta las posiciones desde cero. En los próximos capítulos guardaremos fuerzas, desplazamientos y rigideces en arreglos de NumPy, cuya primera posición es la 0: el grado de libertad 1 estará en la posición 0, el 2 en la posición 1, y así sucesivamente.

Podríamos restar 1 cada vez que accedemos a un arreglo, pero sería peligroso: bastaría con olvidarlo una vez para que el programa use el grado de libertad equivocado, sin dar ningún error. Por eso concentramos la traducción en *una sola función*, que convierte una lista de grados de libertad en la lista de índices correspondiente:

```
def indices(gdl):
    """Traduce grados de libertad (desde 1) a índices
    de NumPy (desde 0). Es el único lugar del libro donde
    se hace esta traducción."""
    resultado = []
    for g in gdl:
        resultado.append(g - 1)
    return resultado
```

> **Recuerda**
>
> En todo el libro, los números que lee una persona (nodos, barras y grados de libertad) empiezan en 1. La función `indices` es el único lugar donde se traduce a la numeración de Python: en ningún otro lugar del código aparece un `- 1` con ese propósito.

NumPy permite además seleccionar varios elementos de un arreglo a la vez, indicando la lista de sus índices entre corchetes. Por ejemplo, los valores que ocupan los grados de libertad libres:

```
# Un valor por cada grado de libertad del ejemplo A
valores = np.array([10.0, 20.0, 30.0, 40.0, 50.0, 60.0])
print(indices(libres))
valores[indices(libres)]
```

```
[2, 4, 5]
array([30., 50., 60.])
```

Los grados de libertad libres 3, 5 y 6 ocupan las posiciones 2, 4 y 5, y seleccionan los valores 30, 50 y 60. Este mecanismo es la base del ensamblaje y de la partición de los próximos capítulos.

> **Para saber más**
>
> Varios textos, como Kassimali (2012), *renumeran* los grados de libertad para que los libres ocupen los primeros números y los restringidos, los últimos. Así, la matriz de la armadura queda ordenada en bloques. Con las listas `libres` y `restringidos` y la selección por índices de NumPy, no necesitamos renumerar: obtenemos los mismos bloques sin cambiar la numeración natural, que es más fácil de seguir en los dibujos. El ejercicio 6.4 muestra la correspondencia entre ambas convenciones.

6.6. Una revisión previa de la estabilidad

Una armadura de b barras y r reacciones tiene $b + r$ fuerzas desconocidas, y el equilibrio de sus n nodos aporta $2n$ ecuaciones. La diferencia es el *grado de indeterminación* (*degree of indeterminacy*):

$$g = b + r - 2n.$$

Si $g < 0$, faltan barras o apoyos y la armadura es inestable; si $g = 0$, es estáticamente determinada (isostática); si $g > 0$, es hiperestática de grado g. Como cada grado de libertad restringido corresponde a una reacción, r es la cantidad de grados de libertad restringidos:

```
def indeterminacion(nodos, barras, restringidos):
    """b + r - 2n: incógnitas menos ecuaciones."""
    b, r, n = len(barras), len(restringidos), len(nodos)
    return b + r - 2 * n
```

Para interpretar el resultado usamos **`if`**, **`elif`** y **`else`**:

```
grado = indeterminacion(nodos, barras, restringidos)
if grado < 0:
    print(f"Inestable: faltan {-grado} barras o apoyos")
elif grado == 0:
    print("Estáticamente determinada (isostática)")
else:
    print(f"Hiperestática de grado {grado}")
```

```
Estáticamente determinada (isostática)
```

Cuidado

La condición $g \geq 0$ es *necesaria* pero no suficiente: una armadura puede tener suficientes barras y apoyos, pero mal dispuestos, como tres reacciones paralelas o un tramo sin diagonales. El método de rigidez detecta esos casos de forma automática, porque la matriz del sistema resulta singular; lo veremos en el capítulo 9.

Problema resuelto 6.1 · Los grados de libertad del ejemplo B

Obtener los vectores de correspondencia de las barras del ejemplo B (figura 3.2), clasificar sus grados de libertad y calcular su grado de indeterminación.

Solución

Las funciones del capítulo sirven sin cambios para cualquier armadura:

```
for numero, barra in barras_b.items():
    print(f"Barra {numero}: {gdl_de_barra(barra)}")

libres_b, restringidos_b = clasificar_gdl(
    nodos_b, apoyos_b)
print(f"Libres:       {libres_b}")
print(f"Restringidos: {restringidos_b}")

grado_b = indeterminacion(nodos_b, barras_b,
                          restringidos_b)
print(f"Grado de indeterminación: {grado_b}")
```

```
Barra 1: [1, 2, 3, 4]
Barra 2: [3, 4, 7, 8]
Barra 3: [5, 6, 7, 8]
Barra 4: [1, 2, 5, 6]
Barra 5: [1, 2, 7, 8]
Barra 6: [3, 4, 5, 6]
Libres:       [3, 5, 6, 7, 8]
Restringidos: [1, 2, 4]
Grado de indeterminación: 1
```

Verificación. El ejemplo B tiene 4 nodos y 8 grados de libertad: 3 restringidos (el apoyo articulado del nodo 1 y el rodillo del nodo 2) y 5 libres. Con 6 barras, $g = 6 + 3 - 8 = 1$: una barra más de las necesarias, la segunda diagonal. Por eso el método de los nudos no basta para resolverlo, y el método de rigidez sí.

Repaso y resumen del capítulo

- Una armadura plana de n nodos tiene $2n$ grados de libertad; el nodo i tiene los grados de libertad $2i-1$ (x) y $2i$ (y).
- En un grado de libertad libre, el desplazamiento es la incógnita; en uno restringido, la incógnita es la reacción.
- El vector de correspondencia de una barra son los grados de libertad de sus dos nodos, en el orden $u_1, \ldots, u_4$.
- `if`, `elif` y `else` ejecutan un bloque u otro según una condición; las comparaciones usan `==`, `!=`, `<`, `>`, `<=` y `>=`.
- La función `indices` es el único lugar donde la numeración desde 1 se traduce a los índices de NumPy, que empiezan en 0.
- El grado de indeterminación $g = b + r - 2n$ clasifica la armadura; $g \geq 0$ es necesario, pero no suficiente, para la estabilidad.

Ejercicios

Ejercicio 6.1 Obtener el vector de correspondencia de la barra 6 del ejemplo B y comprobarlo en la figura 3.2. Solución: p. 155

Ejercicio 6.2 Si el rodillo del nodo 2 del ejemplo A se reemplaza por un apoyo articulado, ¿cuáles son los grados de libertad libres y restringidos? Calcular el nuevo grado de indeterminación. Solución: p. 155

Ejercicio 6.3 Calcular el grado de indeterminación del ejemplo A sin su barra 1 e interpretar el resultado. ¿Qué movimiento puede hacer la armadura? Solución: p. 156

Ejercicio 6.4 Construir la correspondencia entre la numeración natural del ejemplo A y la de los textos que renumeran primero los grados de libertad libres. La función `enumerate(lista, 1)` recorre una lista y entrega, además de cada elemento, su posición contando desde 1. Solución: p. 156

> **Punto de control**
>
> Si algo no funcionó o se omitió una parte, el notebook con todo el código hasta este capítulo permite continuar desde aquí:
>
> Repositorio:
> `github.com/eder3232/libro-rigidez-python-armaduras2d`
> Notebook: `notebooks/control_cap06.ipynb`

Capítulo 7

Ensamblaje de la matriz global

Tenemos la matriz de rigidez de cada barra y sabemos a qué grados de libertad de la armadura corresponde cada una de sus filas y columnas. Falta el paso que da nombre al método: *ensamblar*, es decir, sumar las rigideces de todas las barras en una sola matriz, la *matriz de rigidez global* (*global stiffness matrix*) de la armadura. Es una operación sencilla, que a mano resulta tediosa y propensa a errores, y que en Python se escribe en pocas líneas.

Al terminar este capítulo podremos

- Explicar por qué la matriz global es la suma de las matrices de las barras, colocadas según sus grados de libertad.
- Crear matrices de ceros y acceder a sus elementos con índices.
- Ensamblar la matriz global con un doble bucle y con `np.ix_`.
- Mostrar una matriz con sus grados de libertad como encabezados.
- Verificar que la matriz global es simétrica y singular.

7.1. La idea del ensamblaje

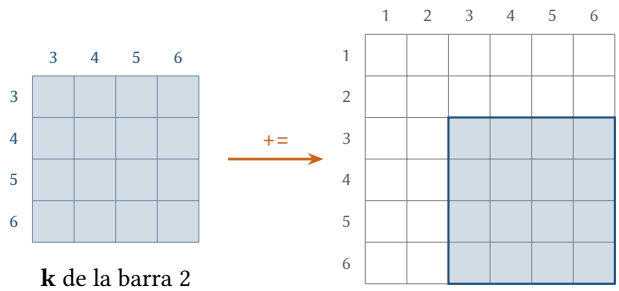
Pensemos en el equilibrio de un nodo en la dirección de uno de sus grados de libertad, digamos el i. La fuerza externa F_i que actúa en esa dirección debe ser igual a la suma de las fuerzas que ejercen sobre el nodo todas las barras que llegan a él. Y la fuerza de cada barra depende, a través de su matriz $\mathbf{k}$, de los desplazamientos de sus extremos. Al escribir el equilibrio de todos los grados de libertad obtenemos el sistema

$$\mathbf{F} = \mathbf{K}\,\mathbf{U},$$

en el que cada término de $\mathbf{K}$ es la suma de los aportes de las barras que conectan esos grados de libertad. Si g_1, g_2, g_3, g_4 es el vector de correspondencia de una barra, su término k_{ab} se suma en la fila g_a y la columna g_b de la matriz global:

$$K_{g_a g_b} \leftarrow K_{g_a g_b} + k_{ab}, \qquad a, b = 1, \ldots, 4.$$

La figura 7.1 muestra la idea para la barra 2 del ejemplo A, cuyo vector de correspondencia es $[3, 4, 5, 6]$: su matriz de 4×4 se suma en el bloque de filas y columnas 3 a 6 de la matriz de 6×6. Las demás barras se suman de la misma forma, cada una en sus propias filas y columnas, y los términos que comparten se acumulan.



k de la barra 2

K de la armadura

Figura 7.1: Ensamblaje de la barra 2 del ejemplo A: cada término de su matriz se suma en la fila y la columna de la matriz global que indica su vector de correspondencia.

> **Recuerda**
>
> El término K_{ij} es la fuerza que aparece en el grado de libertad i cuando el grado de libertad j se desplaza una unidad y todos los demás se mantienen fijos. Por eso solo es distinto de cero si alguna barra une los nodos de i y de j.

7.2. Matrices de ceros e índices

La matriz global empieza llena de ceros, y las barras van sumando sus aportes. `np.zeros` crea un arreglo de ceros con las dimensiones que se indiquen entre paréntesis:

```
n_gdl = 2 * len(nodos)
K = np.zeros((n_gdl, n_gdl))
K.shape
```

```
(6, 6)
```

Para leer o modificar un término de una matriz se indican su fila y su columna, separadas por una coma. Los obtenemos siempre con la función `indices` del capítulo 6, a partir de los grados de libertad. El operador `+=` acumula: cada suma se agrega a lo que el término ya tenía.

```
# Término de la fila del gdl 5 y la columna del gdl 6
fila, columna = indices([5, 6])
K[fila, columna] += 10.0
K[fila, columna] += 2.5
print(fila, columna, K[fila, columna])
```

```
4 5 12.5
```

Para recorrer los cuatro grados de libertad de una barra usamos `range(4)`, que produce los números 0, 1, 2 y 3. En este caso los necesitamos desde cero, porque son posiciones dentro de la matriz **k** de NumPy:

```
for a in range(4):
    print(a, end=" ")
```

```
0 1 2 3
```

7.3. El ensamblaje, término a término

La ecuación clave se traduce de forma literal con dos bucles, uno dentro del otro: para cada barra, recorremos sus filas *a* y, para cada fila, sus columnas *b*. En total, 16 sumas por barra.

```
K = np.zeros((n_gdl, n_gdl))
for numero, barra in barras.items():
    g = gdl_de_barra(barra)
    k = rigideces[numero]
    for a in range(4):
        for b in range(4):
            fila, columna = indices([g[a], g[b]])
            K[fila, columna] += k[a, b]
```

> **Python**
>
> Un bucle dentro de otro se llama *anidado*. El bucle interior se ejecuta completo en cada repetición del exterior: por cada valor de `a`, `b` toma los valores 0, 1, 2 y 3. Cada nivel agrega un nivel de sangría.

7.4. Mostrar una matriz con sus grados de libertad

Al imprimir un arreglo de NumPy, sus filas y columnas no tienen rótulos, y no es fácil saber qué término corresponde a qué grado de libertad. Escribimos una función que imprime la matriz con los grados de libertad, numerados desde 1, como encabezados de filas y columnas:

```
def mostrar_matriz(matriz, filas, columnas=None,
                   decimales=1, ancho=7):
    """Imprime una matriz con los grados de libertad
    (desde 1) como encabezados de filas y columnas."""
    if columnas is None:
        columnas = filas
    encabezado = " gdl"
    for g in columnas:
        encabezado += f"{g:>{ancho}}"
    print(encabezado)
    for i, g in enumerate(filas):
        linea = f"{g:>4}"
        for valor in matriz[i]:
            # Sumar 0.0 evita mostrar -0.0
            valor = round(valor, decimales) + 0.0
            linea += f"{valor:>{ancho}.{decimales}f}"
        print(linea)
```

Si no se indican las `columnas`, se usan los mismos grados de libertad que para las filas, que es el caso de las matrices cuadradas como $\mathbf{K}$. En el capítulo 8 mostraremos también bloques no cuadrados.

> **Python**
>
> **None** es el valor «nada» de Python: lo usamos como valor por omisión de `columnas` para saber si quien llama a la función la indicó. Antes de mostrar cada valor lo redondeamos con `round`; un número negativo muy pequeño, como -10^{-15}, se redondea a `-0.0`, y sumarle `0.0` lo convierte en `0.0`, para que la tabla no muestre ceros con signo. El operador `+=` también funciona con textos: los *concatena*, es decir, los une uno tras otro. Así construimos cada línea de la tabla antes de imprimirla. En el formato `{valor:>{ancho}.{decimales}f}`, el ancho y los decimales se toman de las variables `ancho` y `decimales`: un formato puede contener, a su vez, valores entre llaves.

Las rigideces de una armadura de acero son del orden de decenas de miles de kN/m. Para leerlas con comodidad las mostramos en kN/mm, dividiendo por 1000:

```
todos = list(range(1, n_gdl + 1))
print("K en kN/mm:")
mostrar_matriz(K / 1000, todos)
```

```
K en kN/mm:
 gdl      1      2      3      4      5      6
   1   95.5   38.4  -66.7    0.0  -28.8  -38.4
   2   38.4   51.2    0.0    0.0  -38.4  -51.2
   3  -66.7    0.0   95.5  -38.4  -28.8   38.4
   4    0.0    0.0  -38.4   51.2   38.4  -51.2
   5  -28.8  -38.4  -28.8   38.4   57.6    0.0
```

```
    6  -38.4  -51.2   38.4  -51.2    0.0  102.4
```

7.5. El ensamblaje con np.ix_

NumPy permite seleccionar de una vez un bloque de filas y columnas de una matriz. La función `np.ix_(filas, columnas)` indica que queremos todas las combinaciones de esas filas y esas columnas: con los cuatro índices de una barra, el bloque de 4×4 que le corresponde en la matriz global.

```
g = indices(gdl_de_barra(barras[2]))
print(g)
K[np.ix_(g, g)].shape
```

```
[2, 3, 4, 5]
(4, 4)
```

Con esta selección, las 16 sumas del doble bucle se escriben en una sola línea, y el ensamblaje completo cabe en una función corta. Es la versión que usará nuestra calculadora:

```
def ensamblar_rigidez(nodos, barras, rigideces):
    """Matriz de rigidez global de la armadura."""
    n_gdl = 2 * len(nodos)
    K = np.zeros((n_gdl, n_gdl))
    for numero, barra in barras.items():
        g = indices(gdl_de_barra(barra))
        K[np.ix_(g, g)] += rigideces[numero]
    return K


K_global = ensamblar_rigidez(nodos, barras, rigideces)
```

Ambas versiones deben dar el mismo resultado:

```
np.allclose(K_global, K)
```

```
True
```

7.6. Verificación

La matriz global hereda las propiedades de las matrices de las barras. Debe ser simétrica y, como la armadura todavía no tiene apoyos, singular: puede trasladarse en x, en y y girar sin deformarse. Esos tres movimientos de cuerpo rígido hacen que su rango sea $2n - 3$:

```
simetrica = np.allclose(K_global, K_global.T)
rango = np.linalg.matrix_rank(K_global)
print(f"Simétrica: {simetrica}")
print(f"Rango: {rango} de {n_gdl}")
```

```
Simétrica: True
Rango: 3 de 6
```

En consecuencia, una traslación de toda la armadura no produce fuerzas:

```
# Toda la armadura se traslada 1 mm en x
traslacion_x = np.array([1.0, 0.0, 1.0, 0.0, 1.0, 0.0])
K_global @ traslacion_x
```

```
array([0., 0., 0., 0., 0., 0.])
```

Comprobemos también dos términos a mano. Al grado de libertad 5 (nodo 3, dirección x) llegan las barras 2 y 3, cada una con $EA/Lc^2 = 80 \cdot 0.36 = 28.8\,\text{kN/mm}$: su suma, $57.6\,\text{kN/mm}$, es el término de la fila y columna 5. Al grado de libertad 3 (nodo 2, dirección x) llegan las barras 1 y 2: $66.7 + 28.8 = 95.5\,\text{kN/mm}$.

Problema resuelto 7.1 · La matriz global del ejemplo B

Ensamblar la matriz de rigidez global del ejemplo B y verificar sus propiedades.

Solución

Encadenamos las funciones de los capítulos anteriores. Como la matriz es de 8×8, la mostramos sin decimales y con columnas más angostas:

```
geometria_b = calcular_geometria(nodos_b, barras_b)
rigideces_b = calcular_rigideces(barras_b, geometria_b)
K_b = ensamblar_rigidez(nodos_b, barras_b, rigideces_b)

print("K del ejemplo B en kN/mm:")
mostrar_matriz(K_b / 1000, list(range(1, 9)),
               decimales=0, ancho=6)
print(f"Simétrica: {np.allclose(K_b, K_b.T)}")
print(f"Rango: {np.linalg.matrix_rank(K_b)} de 8")
```

```
K del ejemplo B en kN/mm:
 gdl      1      2      3      4      5      6      7      8
   1    162     38   -133      0      0      0    -29    -38
   2     38    151      0      0      0   -100    -38    -51
   3   -133      0    162    -38    -29     38      0      0
   4      0      0    -38    151     38    -51      0   -100
   5      0      0    -29     38    162    -38   -133      0
```

```
   6     0  -100    38   -51   -38   151     0     0
   7   -29   -38     0     0  -133     0   162    38
   8   -38   -51     0  -100     0     0    38   151
Simétrica: True
Rango: 5 de 8
```

Verificación. La matriz es simétrica y su rango es $5 = 2 \cdot 4 - 3$, como corresponde a una armadura libre en el plano. En el término de la fila y columna 1 (nodo 1, dirección x) se suman la barra 1, horizontal, con $EA/L = 133.3\,\text{kN/mm}$, y la diagonal 5, con $80 \cdot 0.36 = 28.8\,\text{kN/mm}$: $162\,\text{kN/mm}$. La barra vertical 4 no aporta nada en x.

Repaso y resumen del capítulo

- La matriz de rigidez global es la suma de las matrices de las barras: el término k_{ab} de una barra se suma en la fila g_a y la columna g_b de $\mathbf{K}$.
- `np.zeros` crea una matriz de ceros; sus términos se leen y se modifican con `matriz[fila, columna]`, y `+=` acumula.
- `range(4)` produce 0, 1, 2 y 3; los bucles pueden anidarse.
- `np.ix_(filas, columnas)` selecciona un bloque de una matriz y permite ensamblar cada barra en una línea.
- `mostrar_matriz` imprime una matriz con sus grados de libertad numerados desde 1.
- La matriz global es simétrica y, sin apoyos, singular: su rango es $2n - 3$.

Ejercicios

Ejercicio 7.1 Calcular a mano el término de la fila y columna 6 de la matriz global del ejemplo A, en kN/mm, sumando los aportes de las barras que llegan al nodo 3. Comprobarlo con el programa. Solución: p. 156

Ejercicio 7.2 Ensamblar una matriz global del ejemplo A que contenga solo la barra 2 y mostrarla. Comparar el resultado con la figura 7.1. Solución: p. 157

Ejercicio 7.3 Verificar que una traslación vertical de toda la armadura del ejemplo A tampoco produce fuerzas. Solución: p. 157

Ejercicio 7.4 Ensamblar la matriz global del ejemplo A recorriendo las barras en orden inverso (3, 2, 1) y comprobar que el resultado no cambia. ¿Por qué?
Solución: p. 157

Punto de control

Si algo no funcionó o se omitió una parte, el notebook con todo el código hasta este capítulo permite continuar desde aquí:

Repositorio:
`github.com/eder3232/libro-rigidez-python-armaduras2d`
Notebook: `notebooks/control_cap07.ipynb`

Parte III

Resolver y entender

Capítulo 8

Cargas y partición del sistema

La matriz global $\mathbf{K}$ relaciona todas las fuerzas nodales con todos los desplazamientos, $\mathbf{F} = \mathbf{K}\,\mathbf{U}$. Pero no podemos resolver ese sistema tal como está: $\mathbf{K}$ es singular, y en cada grado de libertad conocemos o la fuerza o el desplazamiento, nunca ambos. Antes de resolver debemos ordenar lo que sabemos y lo que no. En este capítulo ensamblamos el vector de cargas y dividimos el sistema en bloques de grados de libertad libres y restringidos. Al hacerlo, veremos cómo los apoyos eliminan la singularidad.

Al terminar este capítulo podremos

- Ensamblar el vector de cargas nodales de la armadura.
- Seleccionar partes de un arreglo con rebanadas y sumar sus elementos.
- Dividir la matriz global en los bloques $\mathbf{K}_{ll}$, $\mathbf{K}_{lr}$, $\mathbf{K}_{rl}$ y $\mathbf{K}_{rr}$.
- Comprobar que $\mathbf{K}_{ll}$ ya no es singular cuando la armadura es estable.

8.1. Lo conocido y lo desconocido

Ordenemos los grados de libertad en dos grupos, los libres (l) y los restringidos (r), y escribamos el sistema por bloques:

$$\begin{bmatrix}\mathbf{F}_l\\ \mathbf{F}_r\end{bmatrix} = \begin{bmatrix}\mathbf{K}_{ll} & \mathbf{K}_{lr}\\ \mathbf{K}_{rl} & \mathbf{K}_{rr}\end{bmatrix}\begin{bmatrix}\mathbf{U}_l\\ \mathbf{U}_r\end{bmatrix}$$

En los grados de libertad libres conocemos las cargas $\mathbf{F}_l$ y buscamos los desplazamientos $\mathbf{U}_l$. En los restringidos conocemos los desplazamientos $\mathbf{U}_r$, que son nulos salvo que haya asentamientos, y buscamos las fuerzas $\mathbf{F}_r$, que contienen las reacciones. El sistema por bloques equivale a dos ecuaciones:

$$\mathbf{F}_l = \mathbf{K}_{ll}\,\mathbf{U}_l + \mathbf{K}_{lr}\,\mathbf{U}_r, \tag{8.1}$$

$$\mathbf{F}_r = \mathbf{K}_{rl}\,\mathbf{U}_l + \mathbf{K}_{rr}\,\mathbf{U}_r. \tag{8.2}$$

La primera tiene una sola incógnita, $\mathbf{U}_l$: la resolveremos en el capítulo 9. Con $\mathbf{U}_l$ conocido, la segunda da directamente las reacciones.

La figura 8.1 muestra los cuatro bloques para el ejemplo A. El dibujo ordena los grados de libertad poniendo primero los libres para que los bloques se vean contiguos; en el programa no hace falta reordenar nada, porque cada bloque se selecciona con las listas `libres` y `restringidos`.

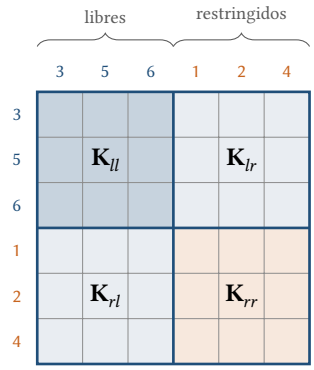


Figura 8.1: Partición de la matriz global del ejemplo A en bloques de grados de libertad libres (3, 5, 6) y restringidos (1, 2, 4).

8.2. El vector de cargas

El vector de cargas **F** tiene un término por cada grado de libertad. Se ensambla como la matriz: empieza en ceros y cada carga se suma en los grados de libertad de su nodo. Usamos `+=` para que, si un mismo nodo recibiera varias cargas, se acumulen.

```
def ensamblar_cargas(nodos, cargas):
    """Vector de cargas nodales F de la armadura."""
    F = np.zeros(2 * len(nodos))
    for nodo, carga in cargas.items():
        fila_x, fila_y = indices(gdl_del_nodo(nodo))
        F[fila_x] += carga["x"]
        F[fila_y] += carga["y"]
    return F


F = ensamblar_cargas(nodos, cargas)
```

Como con las matrices, escribimos una función para mostrar un vector junto con sus grados de libertad:

```
def mostrar_vector(vector, gdl, decimales=2, ancho=10):
```

```
    """Imprime un vector con sus grados de libertad."""
    for i, g in enumerate(gdl):
        valor = round(vector[i], decimales) + 0.0
        print(f"{g:>4}{valor:>{ancho}.{decimales}f}")
```

```
print(" gdl  F (kN)")
mostrar_vector(F, list(range(1, len(F) + 1)))
```

```
 gdl  F (kN)
   1      0.00
   2      0.00
   3      0.00
   4      0.00
   5     20.00
   6    -50.00
```

> **Cuidado**
>
> Las cargas que actúan directamente sobre un grado de libertad restringido quedan en $\mathbf{F}_r$ y no producen desplazamientos: las toma el apoyo. En ese caso, la reacción es la fuerza total en el apoyo menos la carga aplicada. Lo tendremos en cuenta al calcular las reacciones en el capítulo 9.

8.3. Rebanadas

Además de seleccionar elementos con una lista de índices, NumPy permite tomar *rebanadas* (*slices*): tramos regulares de un arreglo. La notación es `[inicio:fin:paso]`, donde el fin no se incluye y cualquiera de los tres valores puede omitirse. Como los grados de libertad en x ocupan las posiciones pares (0, 2, 4, …) y los de y las impares, las rebanadas `[0::2]` y `[1::2]` separan las componentes:

```
print(F[0:2])   # gdl 1 y 2: el nodo 1
print(F[0::2])  # gdl 1, 3, 5: las componentes en x
print(F[1::2])  # gdl 2, 4, 6: las componentes en y
```

```
[0. 0.]
[ 0.  0. 20.]
[  0.   0. -50.]
```

Con el método `sum` obtenemos la resultante de las cargas en cada dirección, una verificación rápida de que el vector se ensambló bien:

```
print(f"Suma de cargas en x: {F[0::2].sum():7.2f} kN")
print(f"Suma de cargas en y: {F[1::2].sum():7.2f} kN")
```

```
Suma de cargas en x:   20.00 kN
Suma de cargas en y:  -50.00 kN
```

8.4. La partición del sistema

Los cuatro bloques se obtienen con `np.ix_`, combinando los índices de los grados de libertad libres y restringidos. Una función los devuelve juntos, y quien la llama los recibe en cuatro variables, en el mismo orden:

```
def particionar(K, libres, restringidos):
    """Bloques Kll, Klr, Krl y Krr de la matriz K."""
    l = indices(libres)
    r = indices(restringidos)
    K_ll = K[np.ix_(l, l)]
    K_lr = K[np.ix_(l, r)]
    K_rl = K[np.ix_(r, l)]
    K_rr = K[np.ix_(r, r)]
    return K_ll, K_lr, K_rl, K_rr


K_ll, K_lr, K_rl, K_rr = particionar(K_global, libres,
                                     restringidos)
F_l = F[indices(libres)]
```

La selección de $\mathbf{F}_l$ es la misma que hicimos en el capítulo 6 con un vector de prueba. Veamos los bloques del ejemplo A; para $\mathbf{K}_{lr}$, cuyas filas y columnas son distintas, indicamos ambas listas a `mostrar_matriz`:

```
print("K_ll en kN/mm:")
mostrar_matriz(K_ll / 1000, libres)
print("K_lr en kN/mm:")
mostrar_matriz(K_lr / 1000, libres, restringidos)
print(" gdl  F_l (kN)")
mostrar_vector(F_l, libres)
```

```
K_ll en kN/mm:
 gdl      3      5      6
   3   95.5  -28.8   38.4
   5  -28.8   57.6    0.0
   6   38.4    0.0  102.4
K_lr en kN/mm:
 gdl      1      2      4
   3  -66.7    0.0  -38.4
   5  -28.8  -38.4   38.4
   6  -38.4  -51.2  -51.2
 gdl  F_l (kN)
   3      0.00
   5     20.00
   6    -50.00
```

8.5. Verificación

La matriz global era singular porque la armadura podía moverse como cuerpo rígido. Al separar los grados de libertad restringidos, impedimos esos movimientos: si la armadura está bien apoyada, $\mathbf{K}_{ll}$ tiene rango completo y puede resolverse. Además, como $\mathbf{K}$ es simétrica, $\mathbf{K}_{lr} = \mathbf{K}_{rl}^{\mathsf{T}}$.

```
print(f"K_ll: {K_ll.shape}, rango "
      f"{np.linalg.matrix_rank(K_ll)}")
print(f"K_lr es la transpuesta de K_rl: "
      f"{np.allclose(K_lr, K_rl.T)}")
```

```
K_ll: (3, 3), rango 3
K_lr es la transpuesta de K_rl: True
```

Recuerda

El rango completo de $\mathbf{K}_{ll}$ es la verificación definitiva de la estabilidad. A diferencia de la cuenta $g = b + r - 2n$ del capítulo 6, detecta también las armaduras con barras o apoyos mal dispuestos, porque refleja si existe algún movimiento posible sin deformar las barras.

Problema resuelto 8.1 · El ejemplo B

Ensamblar el vector de cargas del ejemplo B, particionar su sistema y comprobar que $\mathbf{K}_{ll}$ tiene rango completo.

Solución

Calculamos primero la matriz global del ejemplo B con las funciones de los capítulos anteriores, y después aplicamos las de este capítulo:

```
geometria_b = calcular_geometria(nodos_b, barras_b)
rigideces_b = calcular_rigideces(barras_b, geometria_b)
K_b = ensamblar_rigidez(nodos_b, barras_b, rigideces_b)
F_b = ensamblar_cargas(nodos_b, cargas_b)
libres_b, restringidos_b = clasificar_gdl(nodos_b,
                                          apoyos_b)
K_ll_b, K_lr_b, K_rl_b, K_rr_b = particionar(
    K_b, libres_b, restringidos_b)
F_l_b = F_b[indices(libres_b)]

print("K_ll en kN/mm:")
mostrar_matriz(K_ll_b / 1000, libres_b, decimales=0,
               ancho=6)
print(f"Rango de K_ll: {np.linalg.matrix_rank(K_ll_b)}")
print(" gdl  F_l (kN)")
mostrar_vector(F_l_b, libres_b)
```

```
K_ll en kN/mm:
 gdl     3     5     6     7     8
   3   162   -29    38     0     0
   5   -29   162   -38  -133     0
   6    38   -38   151     0     0
   7     0  -133     0   162    38
   8     0     0     0    38   151
Rango de K_ll: 5
 gdl  F_l (kN)
   3      0.00
   5     30.00
   6    -40.00
   7      0.00
   8    -40.00
```

Verificación. $\mathbf{K}_{ll}$ es de 5×5 y su rango es 5: la armadura está bien apoyada. En $\mathbf{F}_l$ aparecen las cargas de los nodos 3 y 4 en sus grados de libertad (5 y 6 para el nodo 3, 8 para el nodo 4), y ceros en los demás.

Repaso y resumen del capítulo

- El sistema $\mathbf{F} = \mathbf{K}\,\mathbf{U}$ se divide en bloques de grados de libertad libres y restringidos. En los libres se conocen las cargas; en los restringidos, los desplazamientos.
- $\mathbf{F}_l = \mathbf{K}_{ll}\mathbf{U}_l + \mathbf{K}_{lr}\mathbf{U}_r$ da los desplazamientos, y $\mathbf{F}_r = \mathbf{K}_{rl}\mathbf{U}_l + \mathbf{K}_{rr}\mathbf{U}_r$, las reacciones.
- El vector de cargas se ensambla sumando cada carga en los grados de libertad de su nodo.
- Las rebanadas `[inicio:fin:paso]` seleccionan tramos regulares de un arreglo; `sum` suma sus elementos.
- `particionar` devuelve los cuatro bloques de $\mathbf{K}$ seleccionados con `np.ix_`.
- Si la armadura es estable, $\mathbf{K}_{ll}$ tiene rango completo.

Ejercicios

Ejercicio 8.1 Calcular la resultante de las cargas del ejemplo B en cada dirección y compararla con la figura 3.2. Solución: p. 157

Ejercicio 8.2 Agregar al ejemplo A una carga horizontal de 15 kN en el nodo 2, sin modificar el diccionario `cargas` original. Mostrar los vectores $\mathbf{F}$ y $\mathbf{F}_l$. Solución: p. 158

Ejercicio 8.3 Ensamblar la matriz global del ejemplo A sin la barra 1, particionarla con los mismos grados de libertad y calcular el rango de $\mathbf{K}_{ll}$. Relacionar el resultado con el ejercicio 6.3. Solución: p. 158

Ejercicio 8.4 Construir la matriz global del ejemplo A ordenada como en la figura 8.1, con los grados de libertad libres primero, y mostrarla. Dos listas se unen en una sola con el operador `+`. Solución: p. 159

Punto de control

Si algo no funcionó o se omitió una parte, el notebook con todo el código hasta este capítulo permite continuar desde aquí:

Repositorio:
`github.com/eder3232/libro-rigidez-python-armaduras2d`
Notebook: `notebooks/control_cap08.ipynb`

Capítulo 9

Desplazamientos y reacciones

Llegamos al momento que justifica todo el trabajo anterior: resolver el sistema. Con los bloques del capítulo 8, la primera ecuación nos da los desplazamientos de los nodos libres y la segunda, las reacciones en los apoyos. NumPy resuelve el sistema en una línea; nuestro trabajo está en hacerlo bien, en detectar a tiempo una armadura inestable y en comprobar que el resultado respeta el equilibrio.

Al terminar este capítulo podremos

- Resolver el sistema de ecuaciones con `np.linalg.solve` y explicar por qué no conviene invertir la matriz.
- Detectar una armadura inestable y comunicarlo con una excepción.
- Calcular las reacciones en los apoyos.
- Verificar el equilibrio global de la armadura, admitiendo los errores de redondeo.

9.1. La solución del sistema

La ecuación (8.1) tiene una sola incógnita, $\mathbf{U}_l$. Pasando al lado izquierdo el término conocido obtenemos un sistema de ecuaciones lineales:

$$\mathbf{K}_{ll}\,\mathbf{U}_l = \mathbf{F}_l - \mathbf{K}_{lr}\,\mathbf{U}_r$$

Con los desplazamientos de los nodos libres conocidos, la ecuación (8.2) da las fuerzas en los grados de libertad restringidos. Como esas fuerzas incluyen las cargas aplicadas directamente sobre los apoyos, las reacciones son

$$\mathbf{R} = \mathbf{K}_{rl}\,\mathbf{U}_l + \mathbf{K}_{rr}\,\mathbf{U}_r - \mathbf{F}_r, \tag{9.1}$$

donde $\mathbf{F}_r$ contiene solo esas cargas aplicadas, casi siempre nulas. Mientras los apoyos no se desplacen, $\mathbf{U}_r = \mathbf{0}$; en el capítulo 8 dejamos el término $\mathbf{K}_{lr}\mathbf{U}_r$

para poder incorporar más adelante los asentamientos de apoyo.

9.2. Resolver un sistema de ecuaciones con NumPy

La función `np.linalg.solve(A, b)` resuelve el sistema $\mathbf{Ax} = \mathbf{b}$. Los desplazamientos resultan en metros, porque usamos kN y m; los mostramos en milímetros:

```
# Apoyos sin asentamientos: U_r es nulo
U_r = np.zeros(len(restringidos))
lado_derecho = F_l - K_lr @ U_r
U_l = np.linalg.solve(K_ll, lado_derecho)
U_l * 1000  # en mm
```

```
array([ 0.4,  0.6, -0.7])
```

Siempre conviene comprobar que la solución satisface el sistema. El *residuo* (*residual*) $\mathbf{K}_{ll}\mathbf{U}_l - (\mathbf{F}_l - \mathbf{K}_{lr}\mathbf{U}_r)$ debe ser nulo:

```
residuo = K_ll @ U_l - lado_derecho
print(f"Mayor residuo: {np.abs(residuo).max():.1e} kN")
print(np.allclose(K_ll @ U_l, lado_derecho))
```

```
Mayor residuo: 7.1e-15 kN
True
```

El residuo no es exactamente cero sino del orden de 10^{-15}. Las computadoras guardan los números reales con unas 16 cifras significativas, y cada operación redondea la última. Por eso los resultados numéricos se comparan con una *tolerancia* (*tolerance*), como hace `np.allclose`, y nunca exigiendo igualdad exacta.

> **Cuidado**
>
> En los libros de texto la solución se escribe $\mathbf{U}_l = \mathbf{K}_{ll}^{-1}(\mathbf{F}_l - \mathbf{K}_{lr}\mathbf{U}_r)$, y NumPy tiene la función `np.linalg.inv` para invertir una matriz. El resultado es el mismo:
>
> ```
> U_l_con_inv = np.linalg.inv(K_ll) @ lado_derecho
> np.allclose(U_l_con_inv, U_l)
> ```
>
> ```
> True
> ```
>
> Pero calcular la inversa para después multiplicarla es más lento y acumula más errores de redondeo que resolver el sistema directamente, porque `solve` hace solo las operaciones necesarias para esa $\mathbf{b}$. En una armadura de pocas barras la diferencia no se nota; en una de miles de grados de libertad, sí. La regla es simple: para resolver un sistema, `solve`; la inversa, solo si de verdad se necesita la matriz inversa.

9.3. Armaduras inestables: raise, try y except

En el capítulo 8 vimos que si la armadura es inestable, $\mathbf{K}_{ll}$ es singular. Resolver un sistema singular no tiene sentido, y no podemos confiar en que NumPy siempre lo detecte: por el redondeo, una matriz singular puede parecer «casi» invertible y producir desplazamientos enormes y sin significado. Por eso nuestra función comprueba el rango antes de resolver y, si es insuficiente, se detiene con un mensaje claro:

```
def resolver_desplazamientos(K_ll, K_lr, F_l, U_r):
    """Desplazamientos de los gdl libres, U_l."""
    if np.linalg.matrix_rank(K_ll) < len(K_ll):
        raise ValueError("La armadura es inestable: "
                         "K_ll es singular.")
    return np.linalg.solve(K_ll, F_l - K_lr @ U_r)
```

La instrucción **raise** produce una *excepción* (*exception*): detiene la función y comunica qué salió mal. `ValueError` es el tipo de excepción que Python usa para «un dato con un valor inadecuado». Quien llama a la función puede *atrapar* la excepción con **try** y **except** y decidir qué hacer, en lugar de que el programa se detenga. Probemos con el ejemplo A sin su barra 1, el mecanismo de los ejercicios 6.3 y 8.3:

```
barras_sin_1 = {2: barras[2], 3: barras[3]}
K_sin_1 = ensamblar_rigidez(nodos, barras_sin_1,
                            rigideces)
K_ll_sin_1, K_lr_sin_1, _, _ = particionar(
    K_sin_1, libres, restringidos)
try:
    resolver_desplazamientos(K_ll_sin_1, K_lr_sin_1,
                             F_l, U_r)
except ValueError as error:
    print(f"No se puede resolver. {error}")
```

```
No se puede resolver. La armadura es inestable: K_ll es
↪  singular.
```

> **Python**
>
> El bloque de **try** se ejecuta normalmente; si en él se produce una excepción del tipo indicado en **except**, Python salta al bloque de **except**, y **as** `error` guarda el mensaje. Si no hay excepción, el bloque de **except** no se ejecuta. El guion bajo _ es un nombre convencional para los valores que se reciben pero no se van a usar.

9.4. Desplazamientos y reacciones del ejemplo A

Resolvemos ahora el ejemplo A con la función, y armamos el vector completo de desplazamientos, con los valores calculados en los grados de libertad libres y los impuestos en los restringidos. La asignación con una lista de índices coloca cada valor en su lugar:

```
U_r = np.zeros(len(restringidos))
U_l = resolver_desplazamientos(K_ll, K_lr, F_l, U_r)

# Vector completo de desplazamientos
U = np.zeros(2 * len(nodos))
U[indices(libres)] = U_l
U[indices(restringidos)] = U_r
```

```
print(" gdl    U (mm)")
mostrar_vector(U * 1000, list(range(1, len(U) + 1)),
               decimales=3)
```

```
 gdl    U (mm)
   1     0.000
   2     0.000
   3     0.431
   4     0.000
   5     0.563
   6    -0.650
```

Los resultados tienen sentido físico: el nodo 3 se desplaza 0.563 mm hacia la derecha y 0.650 mm hacia abajo, en la dirección de la carga; el rodillo del nodo 2 se desliza 0.431 mm hacia la derecha, porque la barra 1 se alarga; y el nodo 1 no se mueve. Son desplazamientos pequeños, del orden de los que esperamos en una armadura de acero de estas dimensiones.

Las reacciones se calculan con la ecuación (9.1):

```
def calcular_reacciones(K_rl, K_rr, U_l, U_r, F_r):
    """Reacciones en los grados de libertad restringidos.

    F_r son las cargas aplicadas directamente sobre los
    apoyos (normalmente ceros)."""
    return K_rl @ U_l + K_rr @ U_r - F_r


F_r = F[indices(restringidos)]
R = calcular_reacciones(K_rl, K_rr, U_l, U_r, F_r)
```

```
print(" gdl    R (kN)")
mostrar_vector(R, restringidos)
```

```
 gdl    R (kN)
   1    -20.00
   2     11.67
   4     38.33
```

Como el ejemplo A es isostático, podemos comprobarlas con la estática: la reacción horizontal del nodo 1 equilibra la carga de 20 kN, y tomando momentos respecto del nodo 1, $6R_{2y} = 20 \cdot 4 + 50 \cdot 3$, de donde $R_{2y} = 38.33$ kN (ejercicio 9.1).

9.5. Verificación del equilibrio

Una armadura en equilibrio cumple, en conjunto, las tres ecuaciones de la estática: la suma de las fuerzas en x, la suma en y y la suma de momentos de las cargas y las reacciones deben ser nulas. Es la verificación más importante del método de rigidez, porque no depende de cómo calculamos los resultados. El método `copy` crea una copia del vector de cargas, para sumarle las reacciones sin modificar el original:

```python
def comprobar_equilibrio(nodos, F, R, restringidos):
    """Resultantes de cargas más reacciones: Fx, Fy y el
    momento respecto del origen. Deben ser nulas."""
    total = F.copy()
    total[indices(restringidos)] += R
    momento = 0.0
    for nodo in nodos:
        ix, iy = indices(gdl_del_nodo(nodo))
        x, y = nodos[nodo]["x"], nodos[nodo]["y"]
        momento += x * total[iy] - y * total[ix]
    return total[0::2].sum(), total[1::2].sum(), momento
```

```python
suma_x, suma_y, momento = comprobar_equilibrio(
    nodos, F, R, restringidos)
print(f"Suma de fuerzas en x: {suma_x:.2e} kN")
print(f"Suma de fuerzas en y: {suma_y:.2e} kN")
print(f"Suma de momentos:     {momento:.2e} kN·m")
```

```
Suma de fuerzas en x: 3.55e-15 kN
Suma de fuerzas en y: 1.42e-14 kN
Suma de momentos:     5.68e-14 kN·m
```

Las tres sumas son del orden de 10^{-14}, es decir, nulas dentro del redondeo: la armadura está en equilibrio.

Problema resuelto 9.1 · El ejemplo B

Calcular los desplazamientos y las reacciones del ejemplo B y verificar el equilibrio global.

Solución

Encadenamos todas las funciones de la calculadora:

```
geometria_b = calcular_geometria(nodos_b, barras_b)
rigideces_b = calcular_rigideces(barras_b, geometria_b)
K_b = ensamblar_rigidez(nodos_b, barras_b, rigideces_b)
F_b = ensamblar_cargas(nodos_b, cargas_b)
libres_b, restringidos_b = clasificar_gdl(nodos_b,
                                          apoyos_b)
K_ll_b, K_lr_b, K_rl_b, K_rr_b = particionar(
    K_b, libres_b, restringidos_b)

U_r_b = np.zeros(len(restringidos_b))
U_l_b = resolver_desplazamientos(
    K_ll_b, K_lr_b, F_b[indices(libres_b)], U_r_b)
R_b = calcular_reacciones(
    K_rl_b, K_rr_b, U_l_b, U_r_b,
    F_b[indices(restringidos_b)])

print(" gdl    U (mm)")
mostrar_vector(U_l_b * 1000, libres_b, decimales=3)
print(" gdl    R (kN)")
mostrar_vector(R_b, restringidos_b)
```

```
  gdl    U (mm)
    3     0.179
    5     0.900
    6    -0.081
    7     0.854
    8    -0.481
  gdl    R (kN)
    1    -30.00
    2      0.00
    4     80.00
```

```
suma_x, suma_y, momento = comprobar_equilibrio(
    nodos_b, F_b, R_b, restringidos_b)
print(f"ΣFx = {suma_x:.1e}, ΣFy = {suma_y:.1e}, "
      f"ΣM = {momento:.1e}")
```

```
ΣFx = -1.8e-14, ΣFy = 0.0e+00, ΣM = 5.7e-14
```

Verificación. Las sumas son nulas dentro del redondeo. Llama la atención que las reacciones puedan comprobarse con la estática, aunque la armadura sea hiperestática: con un apoyo articulado y un rodillo hay solo tres reacciones, de modo que la armadura es isostática *externamente*. La hiperestaticidad es *interna*: sobra una barra, y la estática no basta para repartir las fuerzas entre las barras. Eso lo resolverá el capítulo 10.

Repaso y resumen del capítulo

- Los desplazamientos libres se obtienen de $\mathbf{K}_{ll}\mathbf{U}_l = \mathbf{F}_l - \mathbf{K}_{lr}\mathbf{U}_r$, y las reacciones, de $\mathbf{R} = \mathbf{K}_{rl}\mathbf{U}_l + \mathbf{K}_{rr}\mathbf{U}_r - \mathbf{F}_r$.
- `np.linalg.solve` resuelve un sistema lineal; invertir la matriz es más lento y menos preciso.
- Los resultados numéricos se comparan con tolerancias: el redondeo deja residuos del orden de 10^{-15}.
- `raise` produce una excepción; `try` y `except` la atrapan. La calculadora rechaza las armaduras inestables antes de resolver.
- El equilibrio global (fuerzas en x e y y momentos) es la verificación más importante de los resultados.

Ejercicios

Ejercicio 9.1 Calcular a mano, con las ecuaciones de la estática, las reacciones del ejemplo A, y compararlas con las del programa. Solución: p. 159

Ejercicio 9.2 Resolver el ejemplo A con dos apoyos articulados (como en el ejercicio 6.2). ¿Pueden comprobarse ahora las reacciones con la estática? ¿Por qué? Solución: p. 159

Ejercicio 9.3 Duplicar las cargas del ejemplo A y comprobar que los desplazamientos se duplican. ¿Qué hipótesis del método garantiza este resultado? Solución: p. 160

Ejercicio 9.4 Calcular el mayor desplazamiento, en valor absoluto, de los nodos del ejemplo B. La función `np.abs` da el valor absoluto de cada elemento de un arreglo, y el método `max`, el mayor de ellos. Solución: p. 160

Punto de control

Si algo no funcionó o se omitió una parte, el notebook con todo el código hasta este capítulo permite continuar desde aquí:

Repositorio:
`github.com/eder3232/libro-rigidez-python-armaduras2d`
Notebook: `notebooks/control_cap09.ipynb`

Capítulo 10

Fuerzas internas

Los desplazamientos y las reacciones son resultados importantes, pero lo que el ingeniero necesita para diseñar es la fuerza en cada barra: cuánto se estira o se comprime, y con qué esfuerzo trabaja su sección. En este capítulo obtenemos la fuerza axial de cada barra a partir de los desplazamientos de sus nodos, la presentamos en una tabla clara y la verificamos de dos formas independientes. Con este paso, nuestra calculadora resuelve el análisis completo de una armadura.

Al terminar este capítulo podremos

- Calcular la fuerza axial de una barra a partir de los desplazamientos de sus nodos.
- Clasificar cada barra en tracción o compresión y calcular su esfuerzo.
- Presentar resultados en tablas con columnas alineadas.
- Verificar las fuerzas con el equilibrio de cada nodo y con el método de los nudos.

10.1. De los desplazamientos a la fuerza axial

Para cada barra, los desplazamientos de sus extremos en ejes globales, $\mathbf{u}$, se leen del vector $\mathbf{U}$ en los grados de libertad de su vector de correspondencia. Con la matriz de transformación los llevamos al eje de la barra, $\mathbf{u'} = \mathbf{T}\,\mathbf{u}$, y con la rigidez local obtenemos las fuerzas en sus extremos, $\mathbf{f'} = \mathbf{k'}\,\mathbf{u'}$: las mismas matrices del capítulo 5, ahora recorridas en sentido inverso.

La fuerza en el extremo final, f'_2, actúa en la dirección del eje x': si es positiva, la barra tira de sus nodos y está en tracción (figura 10.2). Por eso tomamos $N = f'_2$. Desarrollando el producto obtenemos una expresión que muestra el sentido físico del cálculo:

$$N = \frac{EA}{L}(u'_2 - u'_1) = \frac{EA}{L}\left(-c\,u_1 - s\,u_2 + c\,u_3 + s\,u_4\right)$$

La fuerza axial es la rigidez axial por el alargamiento de la barra, la diferencia entre los desplazamientos de sus extremos en la dirección de la barra: la ley de Hooke. La figura 10.1 lo muestra gráficamente. Cada nodo se desplaza en cualquier dirección, pero solo la componente de su desplazamiento a lo largo de la barra la estira o la acorta; la componente perpendicular solo la hace girar.

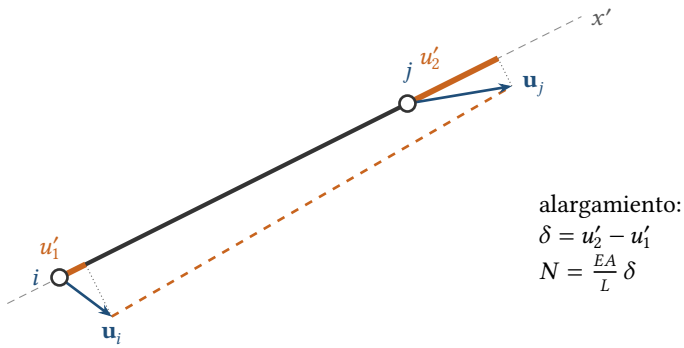


Figura 10.1: Los desplazamientos de los extremos (exagerados), proyectados sobre el eje de la barra, dan u'_1 y u'_2. Su diferencia es el alargamiento.

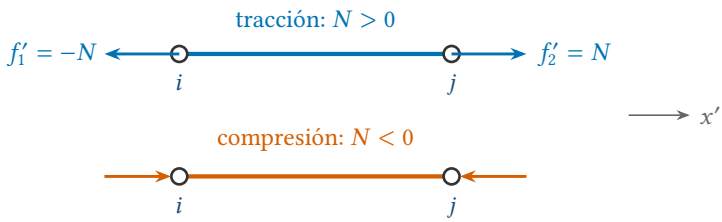


Figura 10.2: Convención de signos: la fuerza axial es positiva en tracción. En ambos casos se muestran las fuerzas que la barra recibe en sus extremos.

10.2. La fuerza axial en Python

Calculemos paso a paso la fuerza de la barra 2 del ejemplo A. Los desplazamientos de sus extremos se seleccionan del vector `U` con la lista de índices de la barra, como en el ensamblaje:

```
# Barra 2 del ejemplo A
g = geometria[2]
```

```
u = U[indices(gdl_de_barra(barras[2]))]
u_local = matriz_transformacion(g["c"], g["s"]) @ u
k_local = rigidez_local(2.0e8, 0.0020, g["longitud"])
f_local = k_local @ u_local
u1, u2 = u_local * 1000  # en mm
print(f"u'1 = {u1:.3f} mm, u'2 = {u2:.3f} mm")
print(f"Alargamiento: {u2 - u1:.3f} mm")
f1, f2 = f_local
print(f"f'1 = {f1:.2f} kN, f'2 = {f2:.2f} kN")
```

```
u'1 = -0.259 mm, u'2 = -0.858 mm
Alargamiento: -0.599 mm
f'1 = 47.92 kN, f'2 = -47.92 kN
```

La barra se acorta 0.599 mm, y con $EA/L = 80\,\text{kN/mm}$ resulta $N = -47.92$ kN: compresión. Las fuerzas en los extremos son iguales y opuestas, como corresponde a una barra en equilibrio.

La función que lo hace para todas las barras repite estos pasos en un bucle y guarda cada fuerza en un diccionario, con el número de barra como clave:

```
def fuerzas_axiales(barras, geometria, U):
    """Fuerza axial de cada barra (+ en tracción)."""
    N = {}
    for numero, barra in barras.items():
        g = geometria[numero]
        u = U[indices(gdl_de_barra(barra))]
        T = matriz_transformacion(g["c"], g["s"])
        u_local = T @ u
        k_local = rigidez_local(barra["elasticidad"],
                                barra["area"],
                                g["longitud"])
        N[numero] = (k_local @ u_local)[1]
    return N


N = fuerzas_axiales(barras, geometria, U)
```

Los corchetes al final de `(k_local @ u_local)[1]` seleccionan el segundo elemento, f'_2, del resultado del producto.

10.3. Tablas de resultados

Para informar los resultados, a la fuerza de cada barra le agregamos su estado y su *esfuerzo* (*stress*) $\sigma = N/A$, que comparamos con la resistencia del material. Clasificamos las fuerzas con una tolerancia, para no llamar «tracción» a un 10^{-15} producido por el redondeo:

```
def clasificar_fuerza(N, tolerancia=1e-6):
    """Tracción, compresión o sin fuerza."""
    if N > tolerancia:
        return "tracción"
    elif N < -tolerancia:
        return "compresión"
    else:
        return "sin fuerza"
```

Una tabla bien presentada evita errores de lectura. Escribimos una función general, que recibe los encabezados y una lista de filas, y alinea todas las columnas a la derecha; los textos se muestran tal cual y los números, con los decimales indicados:

```
def mostrar_tabla(encabezados, filas, decimales=2,
                  ancho=11):
    """Imprime una tabla con columnas alineadas. Los
    números se muestran con los decimales indicados."""
    linea = ""
    for encabezado in encabezados:
        linea += f"{encabezado:>{ancho}}"
    print(linea)
    for fila in filas:
        linea = ""
        for valor in fila:
            if isinstance(valor, str):
                linea += f"{valor:>{ancho}}"
            else:
                valor = round(valor, decimales) + 0.0
                linea += f"{valor:>{ancho}.{decimales}f}"
        print(linea)
```

> **Python**
>
> `isinstance(valor, str)` pregunta si `valor` es un texto (`str`, de *string*). Así la función decide cómo dar formato a cada celda. Las filas se construyen como listas con append, y el número de barra se convierte en texto con la función `str`, para que no se muestre con decimales.

```
filas = []
for numero, fuerza in N.items():
    sigma = fuerza / barras[numero]["area"] / 1000  # MPa
    filas.append([str(numero), fuerza, sigma,
                  clasificar_fuerza(fuerza)])
mostrar_tabla(["Barra", "N (kN)", "σ (MPa)", "Estado"],
              filas)
```

```
      Barra     N (kN)    σ (MPa)      Estado
```

```
1      28.75      14.38   tracción
2     -47.92     -23.96 compresión
3     -14.58      -7.29 compresión
```

La barra 1, el tirante inferior, trabaja en tracción, y las dos barras inclinadas, en compresión: el comportamiento que esperamos de una armadura triangular cargada desde arriba (figura 10.3). El mayor esfuerzo, 24 MPa, es menos del 10 % de la fluencia de un acero estructural ($f_y = 250$ MPa).

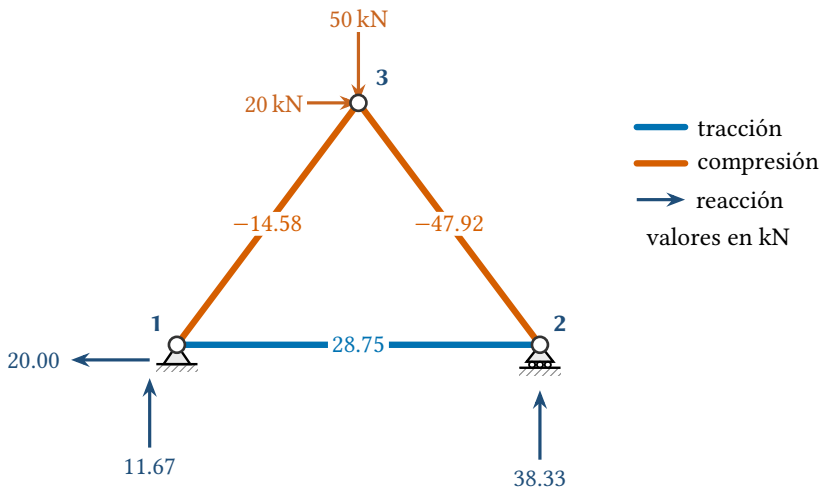


Figura 10.3: Resultados del ejemplo A: fuerzas axiales y reacciones. Las barras en tracción se dibujan en azul y las comprimidas, en naranja.

> **Cuidado**
>
> Este libro calcula fuerzas, no diseña barras. Una barra en compresión debe verificarse, además, por *pandeo*: una barra esbelta puede fallar por inestabilidad con un esfuerzo muy inferior a la fluencia. Esa verificación depende de la longitud de la barra, del momento de inercia de su sección y de la norma de diseño, y queda fuera del alcance del libro.

10.4. Verificación: el equilibrio de cada nodo

En el capítulo 9 verificamos el equilibrio de la armadura en conjunto. Con las fuerzas de las barras podemos hacer algo más exigente: verificar el equilibrio de *cada nodo*. En cada uno, las fuerzas de las barras que llegan a él, las cargas y las reacciones deben sumar cero. Una barra en tracción tira de su nodo inicial en la dirección (c, s) y de su nodo final en la dirección opuesta:

```
def equilibrio_nodal(nodos, barras, geometria, N, F, R,
```

```
                        restringidos):
    """Mayor desequilibrio en los nodos, con las fuerzas
    de las barras, las cargas y las reacciones."""
    residuo = F.copy()
    residuo[indices(restringidos)] += R
    for numero, barra in barras.items():
        g = geometria[numero]
        c, s = g["c"], g["s"]
        ix, iy, jx, jy = indices(gdl_de_barra(barra))
        # En tracción, la barra tira de cada nodo hacia
        # el otro extremo
        residuo[[ix, iy]] += N[numero] * np.array([c, s])
        residuo[[jx, jy]] -= N[numero] * np.array([c, s])
    return np.abs(residuo).max()
```

```
desequilibrio = equilibrio_nodal(nodos, barras, geometria,
                                 N, F, R, restringidos)
print(f"Desequilibrio nodal: {desequilibrio:.1e} kN")
```

```
Desequilibrio nodal: 1.2e-14 kN
```

Esta verificación no es circular: usa solo las fuerzas de las barras y la geometría, no la matriz global. Si hubiéramos cometido un error en el ensamblaje o en la solución, las fuerzas no equilibrarían los nodos.

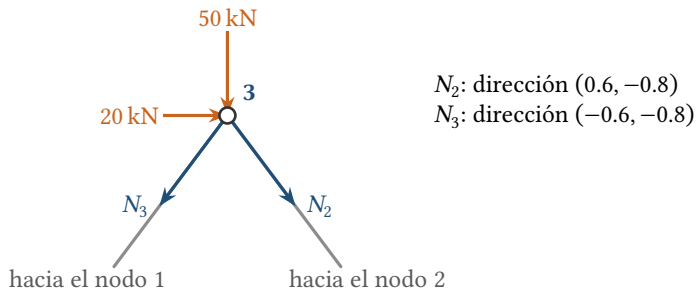


Figura 10.4: Diagrama de cuerpo libre del nodo 3 del ejemplo A. Las fuerzas de las barras se suponen en tracción: cada barra tira del nodo hacia su otro extremo.

Problema resuelto 10.1 · El método de los nudos

Calcular las fuerzas de las barras del ejemplo A con el método de los nudos y compararlas con las del método de rigidez.

Solución

En el nodo 3 concurren dos barras, y sus dos ecuaciones de equilibrio

dan las dos fuerzas (figura 10.4). Suponiendo tracción, cada barra tira del nodo 3 hacia su otro extremo: la barra 2, en la dirección $(0.6, -0.8)$, y la barra 3, en la dirección $(-0.6, -0.8)$. Con la carga $(20, -50)$:

$$0.6\,N_2 - 0.6\,N_3 + 20 = 0, \qquad -0.8\,N_2 - 0.8\,N_3 - 50 = 0.$$

En el nodo 2, el equilibrio horizontal (el rodillo no tiene reacción horizontal) da $-N_1 - 0.6\,N_2 = 0$. Resolvemos el sistema de 2×2 con la misma `np.linalg.solve`:

```
# Nodo 3: incógnitas N2 y N3, en ese orden.
# Barra 2: dirección del nodo 3 al nodo 2 (0.6, -0.8)
# Barra 3: dirección del nodo 3 al nodo 1 (-0.6, -0.8)
A_nodo_3 = np.array([[0.6, -0.6],
                     [-0.8, -0.8]])
b_nodo_3 = np.array([-20.0, 50.0])
N2, N3 = np.linalg.solve(A_nodo_3, b_nodo_3)

# Nodo 2, dirección x: -N1 - 0.6 N2 = 0
N1 = -0.6 * N2
print(f"N1 = {N1:.2f} kN, N2 = {N2:.2f} kN, "
      f"N3 = {N3:.2f} kN")
```

```
N1 = 28.75 kN, N2 = -47.92 kN, N3 = -14.58 kN
```

Verificación. Las tres fuerzas coinciden con las del método de rigidez. Era de esperar: en una armadura isostática, la estática basta, y ambos métodos deben dar el mismo resultado.

10.5. Las fuerzas del ejemplo B

El ejemplo B es hiperestático, y aquí el método de rigidez muestra todo su valor: el método de los nudos no puede repartir la carga entre sus dos diagonales, porque el reparto depende de la rigidez de cada barra. Aplicamos la cadena completa de funciones:

```
geometria_b = calcular_geometria(nodos_b, barras_b)
rigideces_b = calcular_rigideces(barras_b, geometria_b)
K_b = ensamblar_rigidez(nodos_b, barras_b, rigideces_b)
F_b = ensamblar_cargas(nodos_b, cargas_b)
libres_b, restringidos_b = clasificar_gdl(nodos_b,
                                          apoyos_b)
K_ll_b, K_lr_b, K_rl_b, K_rr_b = particionar(
    K_b, libres_b, restringidos_b)
U_r_b = np.zeros(len(restringidos_b))
```

```
U_l_b = resolver_desplazamientos(
    K_ll_b, K_lr_b, F_b[indices(libres_b)], U_r_b)
R_b = calcular_reacciones(
    K_rl_b, K_rr_b, U_l_b, U_r_b,
    F_b[indices(restringidos_b)])
U_b = np.zeros(2 * len(nodos_b))
U_b[indices(libres_b)] = U_l_b

N_b = fuerzas_axiales(barras_b, geometria_b, U_b)
filas = []
for numero, fuerza in N_b.items():
    filas.append([str(numero), fuerza,
                  clasificar_fuerza(fuerza)])
mostrar_tabla(["Barra", "N (kN)", "Estado"], filas)

desequilibrio = equilibrio_nodal(
    nodos_b, barras_b, geometria_b, N_b, F_b, R_b,
    restringidos_b)
print(f"Desequilibrio nodal: {desequilibrio:.1e} kN")
```

```
 Barra     N (kN)     Estado
     1      23.89   tracción
     2     -48.15 compresión
     3      -6.11 compresión
     4      -8.15 compresión
     5      10.19   tracción
     6     -39.81 compresión
Desequilibrio nodal: 1.8e-14 kN
```

Los nodos están en equilibrio. Las dos diagonales trabajan de forma muy distinta (figura 10.5): la diagonal 6, que va del apoyo con rodillo al nodo cargado 3, toma una compresión de 39.8 kN, y la diagonal 5, una tracción de 10.2 kN. La carga horizontal empuja el cuadro hacia la derecha, y la diagonal que se acorta con ese movimiento, la 6, es la que más trabaja.

Como el ejemplo B es hiperestático, el método de los nudos no puede confirmar estas fuerzas. En el anexo D las comparamos con las de OpenSees, un programa de análisis estructural de código abierto, y coinciden.

Repaso y resumen del capítulo

- La fuerza axial es $N = f'_2$, con $\mathbf{f'} = \mathbf{k'Tu}$: la rigidez axial por el alargamiento de la barra. Es positiva en tracción.
- El esfuerzo es $\sigma = N/A$; las barras en compresión deben verificarse además por pandeo.
- La función `mostrar_tabla` presenta resultados con columnas alineadas, e `isinstance` distingue textos de números.

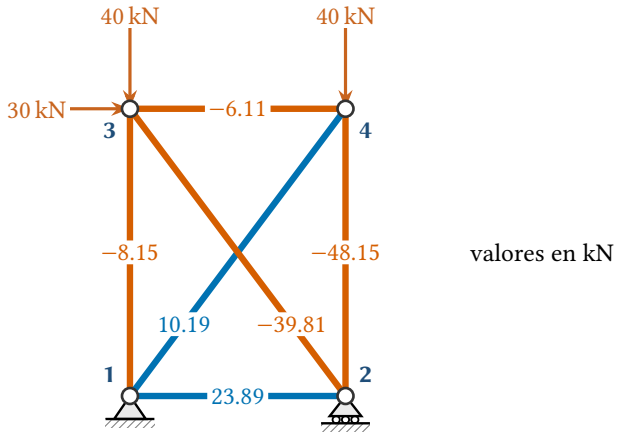


Figura 10.5: Fuerzas axiales del ejemplo B. Las dos diagonales se reparten la carga según su rigidez: algo que la estática sola no puede determinar.

- El equilibrio de cada nodo, con las fuerzas de las barras, verifica de forma independiente el ensamblaje y la solución.
- En una armadura isostática, el método de rigidez y el método de los nudos coinciden; en una hiperestática, solo el primero resuelve el problema.

Ejercicios

Ejercicio 10.1 Calcular el mayor esfuerzo, en valor absoluto, de las barras del ejemplo A y expresarlo como porcentaje de $f_y = 250\,\text{MPa}$. La función `max(a, b)` devuelve el mayor de dos números. Solución: p. 161

Ejercicio 10.2 Calcular las fuerzas del ejemplo A con dos apoyos articulados y compararlas con las originales. Explicar por qué cambia la fuerza de la barra 1. Solución: p. 161

Ejercicio 10.3 Duplicar el área de todas las barras del ejemplo A y comparar los desplazamientos y las fuerzas con los originales. Para no modificar el diccionario `barras`, crear una copia de cada barra con `dict(barra)`. Solución: p. 162

Punto de control

Si algo no funcionó o se omitió una parte, el notebook con todo el código hasta este capítulo permite continuar desde aquí:

Repositorio:

> `github.com/eder3232/libro-rigidez-python-armaduras2d`
> Notebook: `notebooks/control_cap10.ipynb`

Capítulo 11

Visualizar los resultados

Una tabla de desplazamientos y fuerzas contiene toda la información, pero no la muestra. Un dibujo de la armadura deformada, o de sus barras coloreadas según trabajen en tracción o en compresión, se entiende de un vistazo y revela errores que en una tabla pasarían inadvertidos: un apoyo que se mueve, una barra que debería estar comprimida y aparece traccionada. En este capítulo profundizamos en Matplotlib y escribimos las funciones que dibujarán los resultados de cualquier armadura.

Al terminar este capítulo podremos

- Controlar el color, el grosor, el estilo y las etiquetas de un gráfico de Matplotlib.
- Dibujar la deformada de una armadura con un factor de escala adecuado.
- Dibujar el diagrama de fuerzas axiales, con colores para la tracción y la compresión.
- Combinar varios gráficos en una misma figura.

11.1. Dibujar para entender

Los desplazamientos de una armadura son muy pequeños comparados con sus dimensiones: en el ejemplo A, menos de un milímetro en una armadura de seis metros. Dibujados a escala real, la armadura deformada sería indistinguible de la original. Por eso la deformada se dibuja con los desplazamientos multiplicados por un *factor de escala* (*scale factor*) λ: cada nodo de coordenadas (x, y) se dibuja en

$$x_d = x + \lambda\, u_x, \qquad y_d = y + \lambda\, u_y.$$

Un buen factor hace que el mayor desplazamiento dibujado sea una fracción visible del tamaño de la armadura, por ejemplo el 10 %. Si D es la mayor

dimensión de la armadura y $u_{\text{máx}}$ el mayor desplazamiento, $\lambda = 0{,}1\,D/u_{\text{máx}}$.

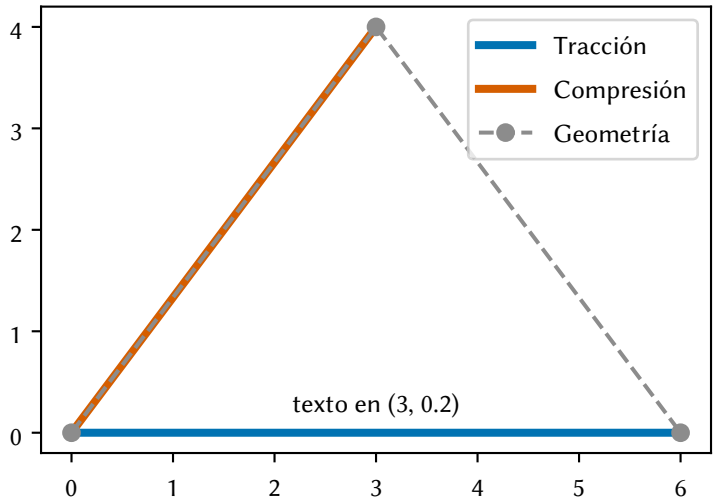
> **Cuidado**
>
> La deformada exagerada sirve para ver *cómo* se mueve la armadura, no cuánto. Con el factor de escala, las barras parecen cambiar mucho de longitud y los ángulos parecen deformarse; en la realidad, esos cambios son de milésimas. El título de cada deformada debe indicar siempre su factor de escala.

11.2. Matplotlib con más detalle

En el capítulo 3 usamos `ax.plot` con un texto corto, como `"k-o"`, para indicar el color, el estilo y el marcador. Para gráficos cuidados conviene indicarlos por separado, con *argumentos con nombre* (*keyword arguments*): `color` para el color, `linewidth` para el grosor, `linestyle` para el estilo (`"-"`, `"--"`, `":"`), `marker` para el símbolo de los puntos y `label` para el texto de la leyenda. `ax.text` escribe un texto en una posición, y sus argumentos `ha` y `va` lo alinean horizontal y verticalmente. Primero definimos los colores del libro como constantes, escritas en mayúsculas por convención:

```
# Colores del libro, distinguibles con daltonismo
TRACCION = "#0072B2"
COMPRESION = "#D55E00"
PRINCIPAL = "#1F4E79"
NEUTRO = "#8C8C8C"
```

```
fig, ax = plt.subplots()
ax.plot([0, 6], [0, 0], color=TRACCION, linewidth=3,
        label="Tracción")
ax.plot([0, 3], [0, 4], color=COMPRESION, linewidth=3,
        label="Compresión")
ax.plot([0, 3, 6], [0, 4, 0], color=NEUTRO,
        linestyle="--", marker="o", label="Geometría")
ax.text(3, 0.2, "texto en (3, 0.2)", ha="center")
ax.set_aspect("equal")
ax.legend()
plt.show()
```



> **Python**
>
> Los argumentos con nombre se escriben `nombre=valor` y pueden darse en cualquier orden. Nuestras propias funciones también los aceptan: es la forma de usar parámetros con valor por omisión, como `decimales=2` en `mostrar_vector`, sin tener que indicar todos los anteriores.

11.3. Una función para dibujar la armadura

Todos los gráficos de este capítulo dibujan las barras de una armadura, a veces en su posición original y a veces en la deformada. Escribimos una sola función, que recibe los ejes `ax` donde dibujar. Así puede usarse varias veces en un mismo gráfico o en distintos paneles de una figura:

```
def dibujar_armadura(ax, nodos, barras, color=NEUTRO,
                     estilo="-", etiqueta=None):
    """Dibuja las barras de una armadura en ax."""
    for barra in barras.values():
        i = nodos[barra["nodo_inicial"]]
        j = nodos[barra["nodo_final"]]
        ax.plot([i["x"], j["x"]], [i["y"], j["y"]],
                color=color, linestyle=estilo, marker="o",
                markersize=4, label=etiqueta)
        etiqueta = None  # la leyenda, una sola vez
    ax.set_aspect("equal")
```

El método `values` de un diccionario recorre solo sus valores, sin las claves, que aquí no necesitamos. La `etiqueta` de la leyenda se asigna a la primera

barra y después se anula, para que la leyenda no repita el mismo texto una vez por barra.

11.4. La deformada

Para dibujar la deformada no necesitamos una función nueva: basta con calcular las posiciones desplazadas de los nodos y dibujar la armadura con ellas. La función devuelve un diccionario con la misma forma que `nodos`, de modo que `dibujar_armadura` lo acepta sin cambios:

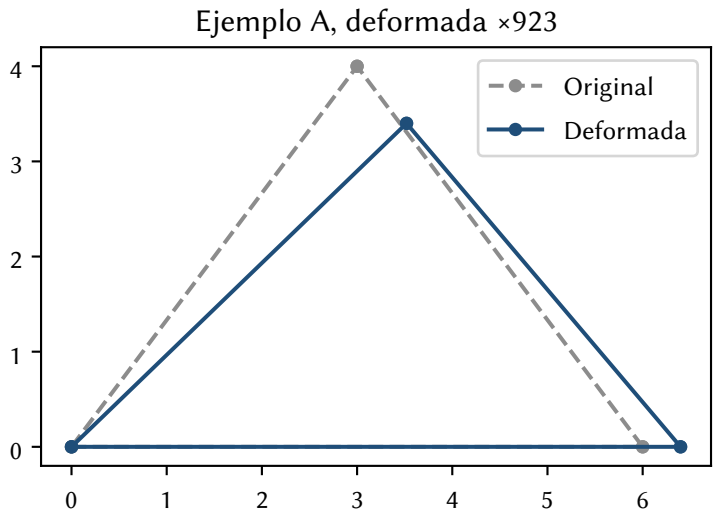
```
def nodos_desplazados(nodos, U, escala):
    """Nodos en su posición deformada, con los
    desplazamientos multiplicados por la escala."""
    nuevos = {}
    for nodo, punto in nodos.items():
        ix, iy = indices(gdl_del_nodo(nodo))
        nuevos[nodo] = {"x": punto["x"] + escala * U[ix],
                        "y": punto["y"] + escala * U[iy]}
    return nuevos
```

El factor de escala se calcula a partir de las coordenadas extremas de la armadura. Las funciones `max` y `min` de Python dan el mayor y el menor elemento de una lista:

```
def escala_automatica(nodos, U, fraccion=0.1):
    """Escala para que el mayor desplazamiento dibujado
    sea una fracción del tamaño de la armadura."""
    xs = []
    ys = []
    for punto in nodos.values():
        xs.append(punto["x"])
        ys.append(punto["y"])
    tamano = max(max(xs) - min(xs), max(ys) - min(ys))
    return fraccion * tamano / np.abs(U).max()
```

Dibujamos la armadura original con línea de trazos y, encima, la deformada:

```
escala = escala_automatica(nodos, U)
fig, ax = plt.subplots()
dibujar_armadura(ax, nodos, barras, estilo="--",
                 etiqueta="Original")
deformados = nodos_desplazados(nodos, U, escala)
dibujar_armadura(ax, deformados, barras, color=PRINCIPAL,
                 etiqueta="Deformada")
ax.set_title(f"Ejemplo A, deformada ×{escala:.0f}")
ax.legend()
plt.show()
```



Verificación. El dibujo debe respetar los apoyos y coincidir con los desplazamientos del capítulo 9. El nodo 1, articulado, no se mueve. El nodo 2 se desplaza solo en horizontal, hacia la derecha, porque el rodillo impide el movimiento vertical. El nodo 3 se mueve hacia la derecha y hacia abajo, como indican sus desplazamientos (u_x = 0.563 mm, u_y = −0.650 mm). La barra 1 se alarga, en tracción, y las barras inclinadas se acortan, en compresión.

11.5. El diagrama de fuerzas axiales

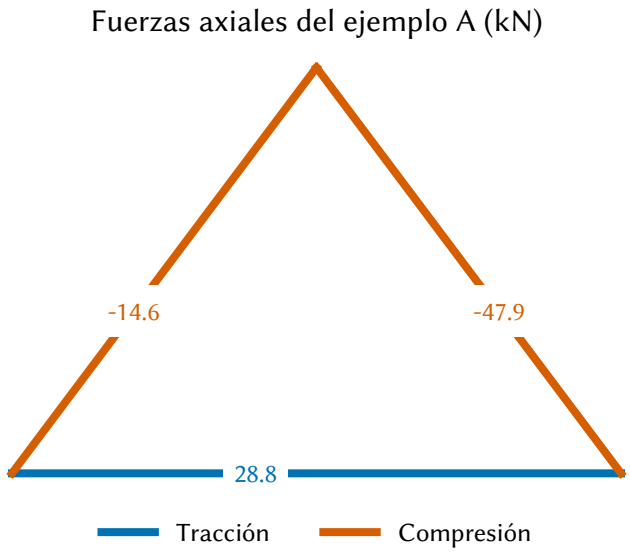
El diagrama de fuerzas axiales colorea cada barra según su estado y escribe su fuerza. Reutilizamos `clasificar_fuerza` del capítulo 10 para elegir el color. Los valores se escriben al 40 % de cada barra, desde su nodo inicial, y no en su punto medio: así, como vimos en el ejercicio 4.4, no se tapan los valores de dos diagonales que se cruzan.

```
def dibujar_fuerzas(ax, nodos, barras, N, decimales=1,
                    tamano=None):
    """Barras en colores según su fuerza axial, con su
    valor en kN al 40 % de cada barra. tamano: tamaño
    de letra de los valores."""
    for numero, barra in barras.items():
        estado = clasificar_fuerza(N[numero])
        if estado == "tracción":
            color = TRACCION
        elif estado == "compresión":
            color = COMPRESION
        else:
            color = NEUTRO
```

```
        i = nodos[barra["nodo_inicial"]]
        j = nodos[barra["nodo_final"]]
        ax.plot([i["x"], j["x"]], [i["y"], j["y"]],
                color=color, linewidth=3)
        # Al 40 % y no en el punto medio: así no se tapan
        # los valores de dos diagonales que se cruzan
        xm = i["x"] + 0.4 * (j["x"] - i["x"])
        ym = i["y"] + 0.4 * (j["y"] - i["y"])
        valor = round(N[numero], decimales) + 0.0
        fondo = {"facecolor": "white", "linewidth": 0}
        ax.text(xm, ym, f"{valor:.{decimales}f}",
                color=color, ha="center", va="center",
                bbox=fondo, fontsize=tamano)
    # Leyenda con líneas vacías: solo muestran los colores
    ax.plot([], [], color=TRACCION, linewidth=3,
            label="Tracción")
    ax.plot([], [], color=COMPRESION, linewidth=3,
            label="Compresión")
    # Leyenda debajo del dibujo, para no tapar las barras
    ax.legend(loc="upper center", bbox_to_anchor=(0.5, 0),
              ncol=2, frameon=False)
    ax.set_aspect("equal")
    ax.axis("off")
```

Para la leyenda dibujamos dos líneas vacías, `ax.plot([], [], ...)`, que no aparecen en el gráfico pero sí en la leyenda con su color. La leyenda va debajo del dibujo y se quitan los ejes, porque en un esquema de la armadura las coordenadas no aportan información. El argumento `bbox` recibe un diccionario con las opciones del recuadro blanco que hay detrás de cada valor.

```
fig, ax = plt.subplots()
dibujar_fuerzas(ax, nodos, barras, N)
ax.set_title("Fuerzas axiales del ejemplo A (kN)")
plt.show()
```



Problema resuelto 11.1 · Un informe gráfico

Preparar una figura con dos paneles para el ejemplo A: la deformada a la izquierda y el diagrama de fuerzas axiales a la derecha.

Solución

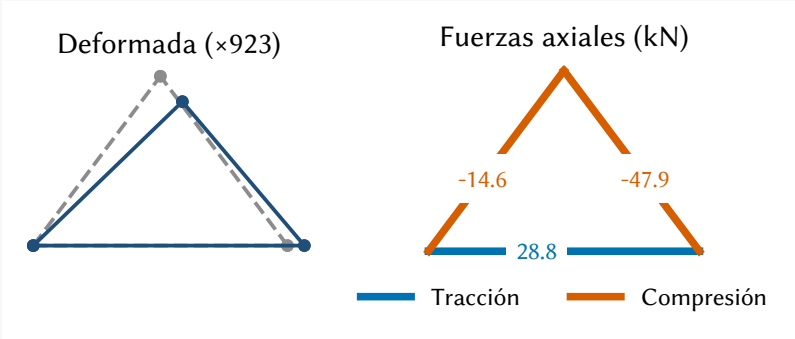
`plt.subplots(1, 2)` crea una figura con una fila y dos columnas de ejes, que recibimos en dos variables. El argumento `figsize` fija el tamaño de la figura en pulgadas, y `tight_layout` ajusta los márgenes para que nada se superponga. Cada función dibuja en los ejes que le indicamos:

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(4.5, 2.2))

escala = escala_automatica(nodos, U)
dibujar_armadura(ax1, nodos, barras, estilo="--")
dibujar_armadura(ax1, nodos_desplazados(nodos, U, escala),
                 barras, color=PRINCIPAL)
ax1.set_title(f"Deformada (×{escala:.0f})")

dibujar_fuerzas(ax2, nodos, barras, N)
ax2.set_title("Fuerzas axiales (kN)")

for ax in (ax1, ax2):
    ax.axis("off")  # sin ejes: es un esquema
fig.tight_layout()
plt.show()
```



Verificación. Ambos paneles cuentan la misma historia: la barra que se alarga en la deformada, la 1, es la única en tracción, y las que se acortan están comprimidas.

Repaso y resumen del capítulo

- La deformada se dibuja con los desplazamientos multiplicados por un factor de escala, $\lambda = 0.1\, D/u_{\text{máx}}$, que siempre debe indicarse.
- Los argumentos con nombre de `ax.plot`, como `color`, `linewidth` o `label`, controlan el aspecto de cada línea; `ax.text` escribe textos y `ax.legend` muestra la leyenda.
- Las funciones de dibujo reciben los ejes `ax` como parámetro, para poder combinarse en una misma figura.
- Cuatro funciones nuevas de la calculadora dibujan los resultados de cualquier armadura: la geometría, las posiciones deformadas, el factor de escala y el diagrama de fuerzas.
- `plt.subplots(1, 2)` crea una figura con dos paneles.
- Un dibujo es también una verificación: la deformada debe respetar los apoyos y los colores deben coincidir con el alargamiento de cada barra.

Ejercicios

Ejercicio 11.1 Dibujar la deformada del ejemplo A con factores de escala de 100 y de 2000, en dos paneles. ¿Cuál es más útil y por qué? Solución: p. 162

Ejercicio 11.2 Agregar al diagrama de fuerzas del ejemplo A un triángulo debajo de cada nodo apoyado. El argumento `marker="^"` de `ax.plot` dibuja un triángulo, y `markersize`, su tamaño. Solución: p. 163

Ejercicio 11.3 Preparar el informe gráfico del ejemplo B: su deformada y su diagrama de fuerzas en dos paneles. Solución: p. 164

Ejercicio 11.4 Dibujar el ejemplo A con el grosor de cada barra proporcional al valor absoluto de su fuerza axial, de modo que la barra más cargada se vea más gruesa. Solución: p. 165

Punto de control

Si algo no funcionó o se omitió una parte, el notebook con todo el código hasta este capítulo permite continuar desde aquí:

Repositorio:
`github.com/eder3232/libro-rigidez-python-armaduras2d`
Notebook: `notebooks/control_cap11.ipynb`

Capítulo 12

Mi calculadora de armaduras

A lo largo de los capítulos anteriores construimos, pieza por pieza, todas las etapas del método de rigidez. Cada una es una función que podemos leer, entender y verificar. Pero para analizar una armadura todavía tenemos que llamarlas una por una, en el orden correcto, como hicimos cada vez con el ejemplo B. En este capítulo damos el último paso: reunimos todas las funciones en una calculadora que, con los datos de una armadura, entrega sus desplazamientos, reacciones y fuerzas, verifica los resultados y los dibuja. Y la ponemos a prueba con una armadura de puente de 21 barras.

Al terminar este capítulo podremos

- Organizar el código en una función que llama a todas las demás.
- Devolver varios resultados con nombre en un diccionario.
- Presentar, verificar y dibujar los resultados de cualquier armadura.
- Analizar una armadura nueva cambiando solo sus datos.

12.1. El camino recorrido

La figura 12.1 resume la calculadora. Los datos describen la armadura (capítulo 3); de ellos salen la geometría y la rigidez de cada barra (capítulos 4 y 5); la numeración de los grados de libertad (capítulo 6) permite ensamblar el sistema y particionarlo (capítulos 7 y 8); su solución da los desplazamientos y las reacciones (capítulo 9), y de los desplazamientos salen las fuerzas axiales (capítulo 10). Por último, los resultados se presentan, se verifican y se dibujan.

12.2. Una función que lo hace todo

Una función puede llamar a otras funciones. La que escribimos ahora no calcula nada por sí misma: llama, en orden, a todas las que escribimos antes,

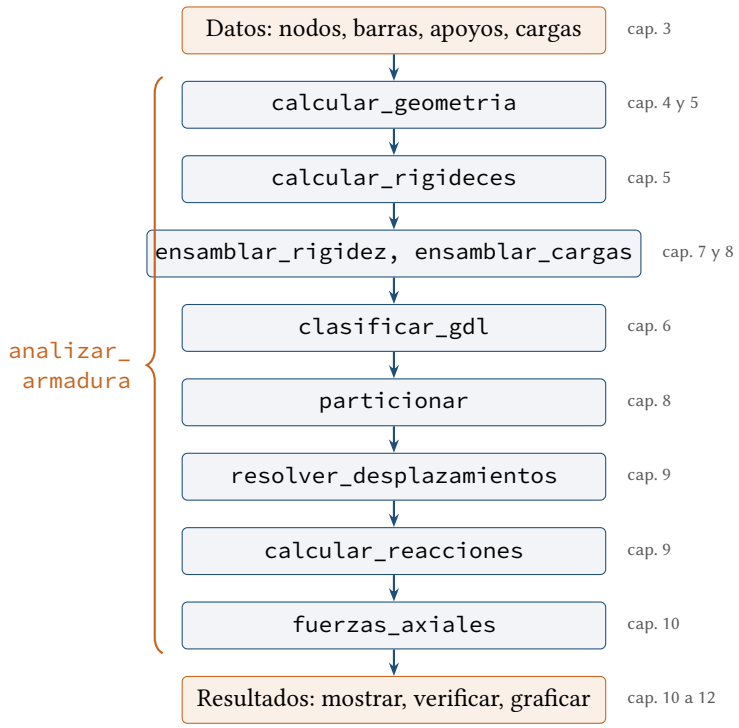


Figura 12.1: Las etapas de la calculadora, cada una con el capítulo donde la construimos. La función `analizar_armadura` encadena los pasos de cálculo.

pasando el resultado de cada etapa a la siguiente. Es la traducción directa de la figura 12.1:

```
def analizar_armadura(nodos, barras, apoyos, cargas):
    """Análisis completo de una armadura plana.

    Devuelve un diccionario con los resultados."""
    geometria = calcular_geometria(nodos, barras)
    rigideces = calcular_rigideces(barras, geometria)
    K = ensamblar_rigidez(nodos, barras, rigideces)
    F = ensamblar_cargas(nodos, cargas)
    libres, restringidos = clasificar_gdl(nodos, apoyos)
    K_ll, K_lr, K_rl, K_rr = particionar(K, libres,
                                         restringidos)
    U_r = np.zeros(len(restringidos))
    U_l = resolver_desplazamientos(
        K_ll, K_lr, F[indices(libres)], U_r)
    U = np.zeros(2 * len(nodos))
    U[indices(libres)] = U_l
    U[indices(restringidos)] = U_r
```

```
    R = calcular_reacciones(K_rl, K_rr, U_l, U_r,
                            F[indices(restringidos)])
    N = fuerzas_axiales(barras, geometria, U)
    return {"geometria": geometria, "K": K, "F": F,
            "U": U, "R": R, "N": N, "libres": libres,
            "restringidos": restringidos}
```

> **Python**
> Una función que produce muchos resultados puede devolverlos en un diccionario, cada uno con su nombre. Es más claro que devolver ocho valores separados por comas: quien la usa escribe `resultados["N"]` o `resultados["U"]` y no necesita recordar en qué posición estaba cada uno. El método `keys` da la lista de claves de un diccionario.

Con una sola llamada obtenemos todos los resultados del ejemplo A, y las fuerzas son, como debe ser, las mismas del capítulo 10:

```
resultados = analizar_armadura(nodos, barras, apoyos,
                               cargas)
print(list(resultados.keys()))
for numero, fuerza in resultados["N"].items():
    print(f"Barra {numero}: N = {fuerza:.2f} kN")
```

```
['geometria', 'K', 'F', 'U', 'R', 'N', 'libres',
 ↪ 'restringidos']
Barra 1: N = 28.75 kN
Barra 2: N = -47.92 kN
Barra 3: N = -14.58 kN
```

12.3. El informe de resultados

Con los resultados en un diccionario, escribimos una función que los presenta en tres tablas: los desplazamientos de cada nodo, las reacciones y las fuerzas axiales. Para las reacciones necesitamos el nodo y la dirección de cada grado de libertad restringido g, que obtenemos invirtiendo la numeración del capítulo 6: el nodo es $(g+1)/2$ redondeado hacia abajo, y la dirección es x si g es impar.

```
def mostrar_resultados(nodos, barras, resultados):
    """Tablas de desplazamientos, reacciones y fuerzas."""
    U = resultados["U"]
    R = resultados["R"]
    N = resultados["N"]
    print("Desplazamientos de los nodos (mm)")
    filas = []
    for nodo in nodos:
```

```
        ix, iy = indices(gdl_del_nodo(nodo))
        filas.append([str(nodo), U[ix] * 1000,
                      U[iy] * 1000])
    mostrar_tabla(["Nodo", "ux", "uy"], filas,
                  decimales=3)

    print("\nReacciones (kN)")
    filas = []
    restringidos = resultados["restringidos"]
    for posicion, g in enumerate(restringidos):
        nodo = (g + 1) // 2
        if g % 2 == 1:
            direccion = "x"
        else:
            direccion = "y"
        filas.append([str(nodo), direccion, R[posicion]])
    mostrar_tabla(["Nodo", "Dirección", "R"], filas)

    print("\nFuerzas axiales")
    filas = []
    for numero, fuerza in N.items():
        sigma = fuerza / barras[numero]["area"] / 1000
        filas.append([str(numero), fuerza, sigma,
                      clasificar_fuerza(fuerza)])
    encabezados = ["Barra", "N (kN)", "σ (MPa)", "Estado"]
    mostrar_tabla(encabezados, filas)
```

Python

El operador `//` es la *división entera* (*floor division*): divide y descarta los decimales, de modo que `7 // 2` es 3. El operador `%` da el *resto* (*remainder*) de la división: `7 % 2` es 1. Un número es impar si su resto al dividirlo por 2 es 1. La secuencia `"\n"` dentro de un texto es un salto de línea.

```
mostrar_resultados(nodos, barras, resultados)
```

```
Desplazamientos de los nodos (mm)
   Nodo         ux         uy
      1      0.000      0.000
      2      0.431      0.000
      3      0.563     -0.650

Reacciones (kN)
   Nodo  Dirección          R
      1          x     -20.00
      1          y      11.67
      2          y      38.33
```

```
Fuerzas axiales
      Barra     N (kN)    σ (MPa)     Estado
          1      28.75      14.38   tracción
          2     -47.92     -23.96 compresión
          3     -14.58      -7.29 compresión
```

La verificación del equilibrio, global y de cada nodo, también se convierte en una función, que reúne las de los capítulos 9 y 10 e informa si ambas se cumplen:

```
def verificar_resultados(nodos, barras, resultados,
                         tolerancia=1e-6):
    """Equilibrio global y de cada nodo. Devuelve True
    si ambos se cumplen dentro de la tolerancia."""
    F, R = resultados["F"], resultados["R"]
    restringidos = resultados["restringidos"]
    suma_x, suma_y, momento = comprobar_equilibrio(
        nodos, F, R, restringidos)
    global_ok = max(abs(suma_x), abs(suma_y),
                    abs(momento)) < tolerancia
    nodal_ok = equilibrio_nodal(
        nodos, barras, resultados["geometria"],
        resultados["N"], F, R, restringidos) < tolerancia
    for nombre, correcto in (("global", global_ok),
                             ("nodal", nodal_ok)):
        if correcto:
            print(f"Equilibrio {nombre}: correcto")
        else:
            print(f"Equilibrio {nombre}: ERROR")
    return global_ok and nodal_ok
```

Y lo mismo con los gráficos del capítulo 11. Para que las armaduras largas y bajas, como las de los puentes, se vean bien, la función coloca los dos paneles uno sobre otro cuando la armadura es más de dos veces más ancha que alta:

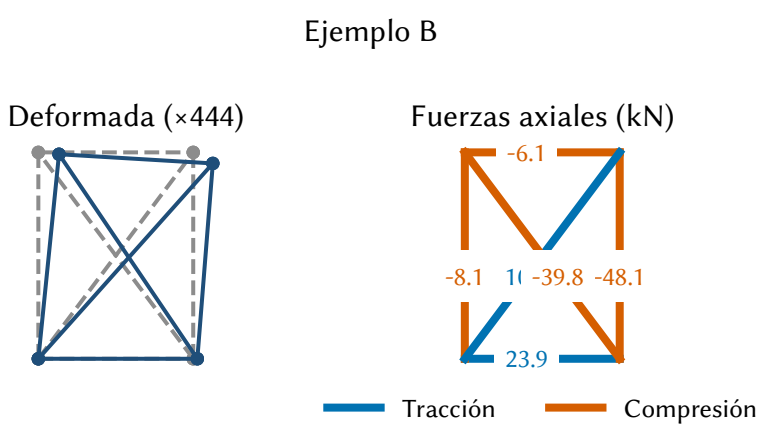
```
def graficar_resultados(nodos, barras, resultados,
                        titulo="", decimales=1,
                        tamano=None):
    """Deformada y diagrama de fuerzas, en dos paneles:
    uno sobre otro si la armadura es muy alargada."""
    U, N = resultados["U"], resultados["N"]
    xs = []
    ys = []
    for punto in nodos.values():
        xs.append(punto["x"])
        ys.append(punto["y"])
    ancho = max(xs) - min(xs)
    alto = max(ys) - min(ys)
    if ancho > 2 * alto:
```

```
        fig, (ax1, ax2) = plt.subplots(2, 1,
                                       figsize=(4.5, 3.6))
    else:
        fig, (ax1, ax2) = plt.subplots(1, 2,
                                       figsize=(4.5, 2.4))
    escala = escala_automatica(nodos, U)
    dibujar_armadura(ax1, nodos, barras, estilo="--")
    deformados = nodos_desplazados(nodos, U, escala)
    dibujar_armadura(ax1, deformados, barras,
                     color=PRINCIPAL)
    ax1.set_title(f"Deformada (×{escala:.0f})")
    ax1.axis("off")
    dibujar_fuerzas(ax2, nodos, barras, N, decimales,
                    tamano)
    ax2.set_title("Fuerzas axiales (kN)")
    fig.suptitle(titulo)
    fig.tight_layout()
    plt.show()
```

Con estas funciones, el ejemplo B, que en cada capítulo nos costó una docena de líneas, se resuelve, verifica y dibuja en tres:

```
resultados_b = analizar_armadura(nodos_b, barras_b,
                                 apoyos_b, cargas_b)
verificar_resultados(nodos_b, barras_b, resultados_b)
graficar_resultados(nodos_b, barras_b, resultados_b,
                    "Ejemplo B")
```

```
Equilibrio global: correcto
Equilibrio nodal: correcto
```



12.4. El ejemplo C: un puente peatonal

Pongamos la calculadora a prueba con una armadura de verdad: un puente peatonal de tipo Pratt, de 18 m de luz y 3 m de altura, con seis paneles de 3 m (figura 12.2). Los cordones y las diagonales extremas tienen 30 cm^2 de área; los montantes y las diagonales interiores, 15 cm^2. El tablero transmite una carga de 25 kN a cada nodo interior del cordón inferior.

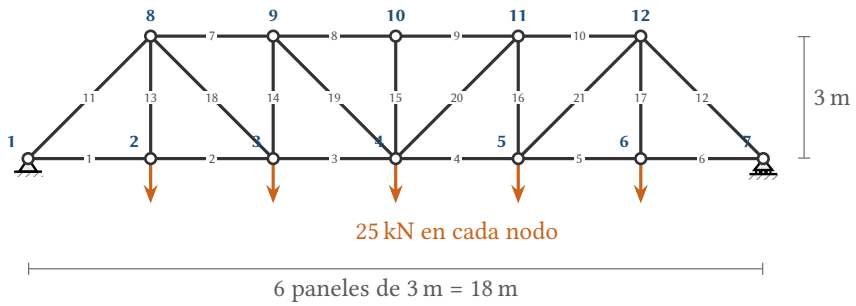


Figura 12.2: Ejemplo C: puente peatonal tipo Pratt, con la numeración de sus 12 nodos y 21 barras.

Con 12 nodos y 21 barras, escribir los datos a mano como en el capítulo 3 sería largo y propenso a errores. Las coordenadas siguen un patrón, así que las generamos con bucles. Para las barras escribimos una pequeña función que arma el diccionario de cada una, de modo que cada barra ocupa una sola expresión. Las áreas se definen una vez como constantes, en mayúsculas por convención:

```
# Ejemplo C: puente peatonal tipo Pratt, 6 paneles de
# 3 m (luz de 18 m) y 3 m de altura. Unidades: kN y m

# Nodos: cordón inferior (1 a 7) y superior (8 a 12)
nodos_c = {}
for k in range(7):
    nodos_c[k + 1] = {"x": 3.0 * k, "y": 0.0}
for k in range(5):
    nodos_c[k + 8] = {"x": 3.0 * (k + 1), "y": 3.0}


def barra(i, j, area, elasticidad=2.0e8):
    """Datos de una barra del nodo i al nodo j."""
    return {"nodo_inicial": i, "nodo_final": j,
            "elasticidad": elasticidad, "area": area}
```

```
A_CORDON = 0.0030  # m², cordones y diagonales extremas
A_ALMA = 0.0015    # m², montantes y diagonales

barras_c = {
    # Cordón inferior
    1: barra(1, 2, A_CORDON), 2: barra(2, 3, A_CORDON),
    3: barra(3, 4, A_CORDON), 4: barra(4, 5, A_CORDON),
    5: barra(5, 6, A_CORDON), 6: barra(6, 7, A_CORDON),
    # Cordón superior
    7: barra(8, 9, A_CORDON), 8: barra(9, 10, A_CORDON),
    9: barra(10, 11, A_CORDON),
    10: barra(11, 12, A_CORDON),
    # Diagonales extremas
    11: barra(1, 8, A_CORDON), 12: barra(12, 7, A_CORDON),
    # Montantes
    13: barra(2, 8, A_ALMA), 14: barra(3, 9, A_ALMA),
    15: barra(4, 10, A_ALMA), 16: barra(5, 11, A_ALMA),
    17: barra(6, 12, A_ALMA),
    # Diagonales, descendiendo hacia el centro
    18: barra(8, 3, A_ALMA), 19: barra(9, 4, A_ALMA),
    20: barra(11, 4, A_ALMA), 21: barra(12, 5, A_ALMA),
}

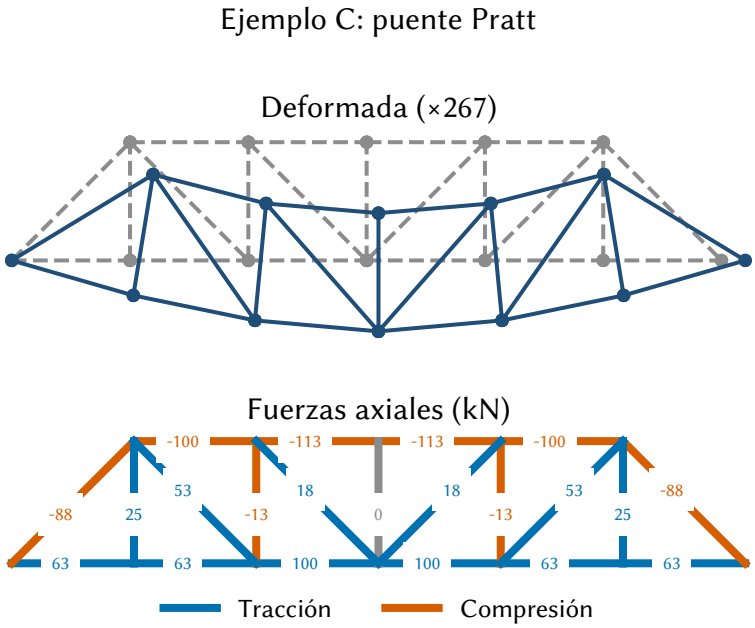
apoyos_c = {
    1: {"x": True, "y": True},   # articulado
    7: {"x": False, "y": True},  # rodillo
}

# 25 kN en cada nodo interior del cordón inferior
cargas_c = {}
for nodo in range(2, 7):
    cargas_c[nodo] = {"x": 0.0, "y": -25.0}
```

El análisis completo, con su verificación y sus gráficos, es igual que para cualquier otra armadura:

```
resultados_c = analizar_armadura(nodos_c, barras_c,
                                 apoyos_c, cargas_c)
verificar_resultados(nodos_c, barras_c, resultados_c)
graficar_resultados(nodos_c, barras_c, resultados_c,
                    "Ejemplo C: puente Pratt",
                    decimales=0, tamano=7)
```

```
Equilibrio global: correcto
Equilibrio nodal: correcto
```



Los resultados muestran el comportamiento típico de una armadura Pratt: el cordón superior trabaja en compresión y el inferior en tracción, como las fibras superior e inferior de una viga; las diagonales interiores, inclinadas hacia el centro, trabajan en tracción, mientras que las extremas, que llevan la carga a los apoyos, y los montantes interiores trabajan en compresión. El montante central no tiene fuerza, y los montantes extremos, de los que «cuelga» la carga de los nodos 2 y 6, trabajan en tracción. La flecha en el centro del vano es de 6.74 mm, del orden de $L/2700$, y cada apoyo recibe 62.5 kN, la mitad de la carga total. Las tablas completas son:

```
mostrar_resultados(nodos_c, barras_c, resultados_c)
```

```
Desplazamientos de los nodos (mm)
       Nodo          ux          uy
          1       0.000       0.000
          2       0.313      -3.321
          3       0.625      -5.695
          4       1.125      -6.736
          5       1.625      -5.695
          6       1.938      -3.321
          7       2.250       0.000
          8       2.188      -3.071
          9       1.688      -5.820
         10       1.125      -6.736
         11       0.563      -5.820
```

```
         12      0.063     -3.071

Reacciones (kN)
       Nodo  Dirección          R
          1          x       0.00
          1          y      62.50
          7          y      62.50

Fuerzas axiales
      Barra     N (kN)    σ (MPa)     Estado
          1      62.50      20.83   tracción
          2      62.50      20.83   tracción
          3     100.00      33.33   tracción
          4     100.00      33.33   tracción
          5      62.50      20.83   tracción
          6      62.50      20.83   tracción
          7    -100.00     -33.33 compresión
          8    -112.50     -37.50 compresión
          9    -112.50     -37.50 compresión
         10    -100.00     -33.33 compresión
         11     -88.39     -29.46 compresión
         12     -88.39     -29.46 compresión
         13      25.00      16.67   tracción
         14     -12.50      -8.33 compresión
         15       0.00       0.00 sin fuerza
         16     -12.50      -8.33 compresión
         17      25.00      16.67   tracción
         18      53.03      35.36   tracción
         19      17.68      11.79   tracción
         20      17.68      11.79   tracción
         21      53.03      35.36   tracción
```

Problema resuelto 12.1 · El método de las secciones

Verificar las fuerzas de las barras 3 (cordón inferior) y 8 (cordón superior) del ejemplo C con el método de las secciones.

Solución

Cortamos la armadura por el tercer panel, entre $x = 6$ y $x = 9$ m, y estudiamos el tramo izquierdo. Por simetría, cada reacción vale la mitad de la carga total, $R = 5 \cdot 25/2 = 62.5$ kN. Tomando momentos respecto del nodo 9, en $(6, 3)$, solo la barra 3 tiene momento entre las barras cortadas; respecto del nodo 4, en $(9, 0)$, solo la barra 8:

$$N_3 = \frac{62.5 \cdot 6 - 25 \cdot 3}{3}, \qquad N_8 = -\frac{62.5 \cdot 9 - 25 \cdot 6 - 25 \cdot 3}{3}.$$

```
# Corte por el tercer panel (x entre 6 y 9 m)
```

```
R1 = 5 * 25.0 / 2  # reacción en cada apoyo, por simetría
h = 3.0
# Momentos respecto del nodo 9 (x = 6): cordón inferior
N_inferior = (R1 * 6 - 25.0 * 3) / h
# Momentos respecto del nodo 4 (x = 9): cordón superior
N_superior = -(R1 * 9 - 25.0 * 6 - 25.0 * 3) / h

N_c = resultados_c["N"]
print(f"Barra 3: secciones {N_inferior:8.2f} kN, "
      f"calculadora {N_c[3]:8.2f} kN")
print(f"Barra 8: secciones {N_superior:8.2f} kN, "
      f"calculadora {N_c[8]:8.2f} kN")
```

```
Barra 3: secciones   100.00 kN, calculadora   100.00 kN
Barra 8: secciones  -112.50 kN, calculadora  -112.50 kN
```

Verificación. Ambos métodos dan los mismos valores: la calculadora reproduce el análisis clásico de una armadura isostática.

12.5. Mi calculadora

Llegamos al producto final del libro: un notebook propio, organizado en tres secciones.

Motor. Todas las funciones que escribimos en el libro. Se ejecuta una vez y no hace falta modificarlo.

Datos. Los cuatro diccionarios que describen la armadura. Es lo único que cambia de un problema a otro.

Resultados. La llamada a `analizar_armadura`, la verificación, los gráficos y las tablas.

Separar el motor de los datos es la lección más valiosa de este capítulo: la misma herramienta, escrita y verificada una vez, sirve para cualquier armadura.

> **Mi calculadora de armaduras**
>
> El notebook con la calculadora completa, organizado en Motor, Datos y Resultados, listo para copiar a nuestro Google Drive:
>
> Repositorio:
> `github.com/eder3232/libro-rigidez-python-armaduras2d`
> Notebook: `notebooks/mi_calculadora.ipynb`

Estrategia de resolución

Para analizar una armadura propia con la calculadora:

1. Dibujar la armadura y numerar sus nodos (de forma consecutiva, desde 1) y sus barras.
2. Convertir todos los datos a kN y m: E en kN/m^2 y A en m^2.
3. Reemplazar los diccionarios de la sección Datos y ejecutar la sección Resultados.
4. Comprobar que la verificación del equilibrio sea correcta y que el dibujo coincida con la armadura y sus apoyos.
5. Revisar que los resultados sean razonables: desplazamientos de milímetros, esfuerzos muy inferiores a la fluencia y un patrón de tracciones y compresiones con sentido físico.

Los errores más frecuentes al usar la calculadora, y cómo reconocerlos, son los de la tabla 12.1.

Tabla 12.1: Errores frecuentes al usar la calculadora.

Síntoma	Causa probable	Solución
`KeyError`	Barra con un nodo inexistente	Revisar `barras`
`ZeroDivisionError`	Dos nodos coinciden	Revisar `nodos`
«Es inestable»	Faltan barras o apoyos	Revisar el dibujo
Desplazamientos enormes	Unidades mezcladas	Pasar a kN y m
El dibujo no coincide	Datos mal escritos	Corregir los datos

Repaso y resumen del capítulo

- La función `analizar_armadura` encadena todas las etapas del método y devuelve los resultados en un diccionario.
- Tres funciones más presentan, verifican y dibujan los resultados de cualquier armadura.
- `//` es la división entera y `%`, el resto; con ellos se obtienen el nodo y la dirección de un grado de libertad.
- Los datos de una armadura grande pueden generarse con bucles y funciones auxiliares, como `barra`.
- El notebook final separa el motor, que no cambia, de los datos, que describen cada problema, y de los resultados.
- Toda solución nueva se verifica: equilibrio, dibujo y sentido físico.

Ejercicios

Ejercicio 12.1 Calcular la flecha (el desplazamiento vertical hacia abajo) del nodo 4 del ejemplo C, en el centro del vano, y la relación entre la luz y la flecha. Solución: p. 166

Ejercicio 12.2 Analizar con la calculadora la armadura de cubierta del ejercicio 3.2. ¿Por qué la barra 5 no tiene fuerza? Solución: p. 167

Ejercicio 12.3 Agregar al ejemplo C una carga de viento de 10 kN horizontales en cada nodo del cordón superior, además de las cargas verticales, y comparar el diagrama de fuerzas con el original. Solución: p. 168

Ejercicio 12.4 Quitar del ejemplo C las diagonales 18 y 19 e intentar analizarlo. Explicar el mensaje de la calculadora con ayuda de la figura 12.2. La expresión `numero not in (18, 19)` es verdadera si el número no es 18 ni 19. Solución: p. 169

Punto de control

Si algo no funcionó o se omitió una parte, el notebook con todo el código hasta este capítulo permite continuar desde aquí:

Repositorio:
`github.com/eder3232/libro-rigidez-python-armaduras2d`
Notebook: `notebooks/control_cap12.ipynb`

Anexo A

Referencia rápida de Python y NumPy

Este anexo reúne, para consultarlas rápidamente, todas las herramientas de Python que usamos en el libro. No está pensado para leerse de corrido: cada tabla indica el capítulo donde la herramienta se explica con detalle, y los ejemplos usan los mismos nombres que la calculadora.

A.1. Notebooks en Google Colab

- Un notebook se compone de *celdas* de código y de texto. Una celda de código se ejecuta con Mayús + Enter.
- Las variables y funciones definidas en una celda existen para todas las celdas que se ejecuten después, en el orden en que se ejecutan, no en el que aparecen.
- Si el notebook se comporta de forma extraña, el menú Entorno de ejecución > Reiniciar sesión y ejecutar todo lo ejecuta de nuevo desde el principio y en orden.
- Un notebook propio se guarda en nuestro Google Drive; uno abierto desde un enlace debe copiarse con Archivo > Guardar una copia en Drive para conservar nuestros cambios.

A.2. Números, texto y resultados

Las operaciones aritméticas siguen las prioridades habituales: primero las potencias, después productos y divisiones, y por último sumas y restas. El operador `%`, que da el resto de una división, se usa en el capítulo 12.

Tabla A.1: Variables, operaciones y resultados.

Código	Qué hace	Cap.
`area = 0.0020`	Crea o cambia una variable	2
`E = 2.0e8`	Notación científica: 2.0×10^8	2
`dx**2`	Potencia	2
`7 // 2`	División entera: 3	12
`import math`	Carga el módulo de matemática	2
`math.sqrt(x)`	Raíz cuadrada	2
`math.atan2(dy, dx)`	Ángulo en radianes, en su cuadrante	2
`math.degrees(a)`	Convierte radianes a grados	2
`abs(x)`	Valor absoluto	3
`round(x, 2)`	Redondea a 2 decimales	7
`print("L =", L)`	Muestra valores	2
`f"{L:.2f} m"`	Texto con un número de 2 decimales	2
`f"{g:>4}"`	Número alineado a la derecha en 4 espacios	4
`str(3)`	Convierte un número en texto	10

A.3. Listas y diccionarios

Una *lista* guarda valores en orden y se consulta por posición, desde 0. Un *diccionario* guarda valores con nombre y se consulta por clave. En la calculadora, los nodos, las barras, los apoyos y las cargas son diccionarios cuyas claves son los números de nodo o de barra, desde 1, y cuyos valores son a su vez diccionarios.

Tabla A.2: Listas y diccionarios.

Código	Qué hace	Cap.
`libres = [3, 5, 6]`	Crea una lista	3
`libres[0]`	Primer elemento (posición 0)	3
`len(libres)`	Cantidad de elementos	3
`libres.append(7)`	Agrega un elemento al final	3
`list(range(1, 9))`	Lista de 1 a 8	7
`nodos = {1: {"x": 0.0}}`	Crea un diccionario anidado	3
`nodos[1]["x"]`	Valor de una clave	3
`cargas[3] = {"x": 20.0}`	Agrega o cambia una clave	3
`3 in cargas`	¿Existe la clave 3?	3
`nodos.items()`	Pares de clave y valor	4
`nodos.values()`	Solo los valores	11
`resultados.keys()`	Solo las claves	12
`dict(barra)`	Copia un diccionario	10

A.4. Bucles y decisiones

Las instrucciones que se repiten o que dependen de una condición forman un *bloque*: van sangradas cuatro espacios debajo de una línea que termina en dos puntos.

Tabla A.3: Bucles, decisiones y comparaciones.

Código	Qué hace	Cap.
`for n, b in barras.items():`	Recorre un diccionario	4
`for g in range(4):`	Repite para 0, 1, 2 y 3	7
`for i, g in enumerate(gdl):`	Recorre con la posición	7
`total += L`	Igual que `total = total + L`	4
`if apoyo["x"]:`	Ejecuta si se cumple	6
`else:`	Si no se cumple	6
`==`, `!=`, `<`, `>=`	Comparaciones	6
`True`, `False`	Valores lógicos	3
`and`, `not`, `not in`	Combinan condiciones	12
`None`	El valor «nada»	7

A.5. Funciones y errores

Una función recibe datos por sus parámetros y devuelve un resultado con `return`. Las variables creadas dentro de ella son locales: no existen fuera de la función.

Tabla A.4: Funciones y excepciones.

Código	Qué hace	Cap.
`def rigidez_local(E, A, L):`	Define una función	5
`"""Documentación."""`	Explica qué hace la función	5
`return k`	Devuelve el resultado	5
`return {"U": U, "N": N}`	Devuelve varios resultados	12
`decimales=1`	Valor por omisión	7
`mostrar_vector(F, decimales=3)`	Argumento con nombre	11
`raise ValueError("...")`	Produce un error	9
`except ValueError:`	Atrapa un error	9
`isinstance(x, str)`	¿Es x un texto?	10

A.6. NumPy

NumPy trabaja con arreglos de números: vectores y matrices. Sus índices, como los de las listas, empiezan en 0; la función `indices` de la calculadora (capítulo 6) traduce los grados de libertad, numerados desde 1, a esos índices.

Tabla A.5: Vectores y matrices con NumPy.

Código	Qué hace	Cap.
`import numpy as np`	Carga NumPy	5
`np.array([[1, -1], [-1, 1]])`	Crea una matriz	5
`T.T @ k @ T`	Transpuesta y producto	5
`K.shape`	Dimensiones	5
`np.zeros((8, 8))`	Matriz nula de 8×8	7
`K[0, 1]`	Fila 0, columna 1	7
`K[np.ix_(filas, cols)]`	Submatriz	7
`K[np.ix_(e, e)] += k`	Ensambla una barra	7
`F[0::2]`	Elementos 0, 2, 4, …	8
`np.allclose(k, k.T)`	¿Casi iguales?	5
`np.linalg.matrix_rank(K)`	Rango	5
`np.linalg.solve(K, F)`	Resuelve el sistema	9
`np.abs(U).max()`	Mayor desplazamiento	9
`np.set_printoptions(precision=1)`	Decimales	5

> **Cuidado**
>
> Para resolver un sistema de ecuaciones usamos `np.linalg.solve`, no la inversa `np.linalg.inv`: es más rápida y más precisa (capítulo 9).

A.7. Matplotlib

Cada gráfico se dibuja en unos *ejes*, `ax`, que pertenecen a una *figura*, `fig`. Primero se crean ambos, después se dibuja sobre los ejes y al final se muestra la figura.

Tabla A.6: Gráficos con Matplotlib.

Código	Qué hace	Cap.
`import matplotlib.pyplot as plt`	Carga Matplotlib	3
`fig, ax = plt.subplots()`	Figura y ejes	3
`fig, (ax1, ax2) = plt.subplots(1, 2)`	Dos paneles	11
`ax.plot([x1, x2], [y1, y2])`	Dibuja una línea	3
`color="black", linewidth=2`	Color y grosor	11
`linestyle="--"`	Línea discontinua	11
`ax.text(x, y, "3")`	Escribe un texto	4
`ax.set_aspect("equal")`	Misma escala en x e y	3
`ax.set_title("Deformada")`	Título de los ejes	3
`ax.legend()`	Muestra la leyenda	11
`ax.axis("off")`	Oculta los ejes	11
`plt.show()`	Muestra la figura	3

Anexo B

Leer un error de Python

Cometer errores al programar es inevitable, y no es una mala señal: es parte del trabajo. Lo que distingue a quien programa con soltura no es que no se equivoque, sino que sabe leer lo que Python le dice cuando algo falla. Este anexo enseña a leer esos mensajes con los errores más frecuentes al construir la calculadora. Todos los mensajes que se muestran son reales: los produjo el código del libro.

B.1. Cómo se lee un mensaje de error

Supongamos que, al describir una armadura, escribimos por descuido un nodo que no existe: la barra 2 del ejemplo A va al nodo 4, pero la armadura solo tiene tres nodos.

```
barras_mal = {
    1: barra(1, 2, 0.0020),
    2: barra(2, 4, 0.0020),
    3: barra(3, 1, 0.0020),
}
geometria_mal = calcular_geometria(nodos, barras_mal)
```

Python se detiene y muestra este mensaje:

```
Traceback (most recent call last):
  File "<celda 44>", line 6, in <module>
    geometria_mal = calcular_geometria(nodos, barras_mal)
                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "<celda 9>", line 10, in calcular_geometria
    final = nodos[barra["nodo_final"]]
            ~~~~~^^^^^^^^^^^^^^^^^^^^^
KeyError: 4
```

El mensaje se llama *traza* (*traceback*) y se lee *de abajo hacia arriba*:

1. **La última línea dice qué pasó.** Primero el tipo de error, `KeyError`, y después el detalle: la clave `4` no existe en el diccionario.

2. **Las líneas anteriores dicen dónde pasó.** Cada bloque que empieza con `File` es un paso. El primero es nuestra celda; el segundo, la línea 10 de la función que llamamos, `calcular_geometria`, que busca el nodo final de cada barra. Los signos `~` y `^` subrayan la parte exacta de la línea que falló.
3. **El primer bloque es nuestra celda.** Ahí está la línea que escribimos nosotros y que desencadenó el error.

Que el error aparezca dentro de una función no significa que la función esté mal: casi siempre el problema está en los datos que le dimos. Aquí, `calcular_geometria` buscó el nodo final de una barra y no lo encontró, porque nuestro diccionario `barras_mal` usa un nodo inexistente.

> **Para saber más**
>
> Colab muestra la misma información con otro formato: agrega colores, marca la línea que falló con una flecha `----->` y nombra las celdas de otra manera. Se lee igual, de abajo hacia arriba. Colab también puede ofrecer un botón para que una inteligencia artificial explique el error; es una ayuda útil, pero conviene aprender a leer el mensaje por nuestra cuenta.

B.2. Un nombre mal escrito

El error más frecuente de todos es escribir mal un nombre:

```
geometria = calcular_geometria(nodos, barras)
rigideces = calcular_rigideces(barras, geometira)
```

```
Traceback (most recent call last):
  File "<celda 45>", line 2, in <module>
    rigideces = calcular_rigideces(barras, geometira)
                                           ^^^^^^^^^
NameError: name 'geometira' is not defined. Did you mean:
 ↪  'geometria'?
```

Python no conoce `geometira` y, como el nombre se parece a uno que sí existe, nos sugiere el correcto (*Did you mean*, ¿quisiste decir?).

Un `NameError` aparece también cuando usamos una variable o una función cuya celda no hemos ejecutado: por ejemplo, al abrir de nuevo un notebook, porque Colab no recuerda nada de la sesión anterior. La solución es ejecutar las celdas anteriores, o todas con Entorno de ejecución > Ejecutar todas.

B.3. Errores de escritura

Algunos errores impiden que Python entienda el código, y aparecen antes de que se ejecute una sola línea. El más común es olvidar los dos puntos al

final de un `for`, un `if` o un `def`:

```
for numero in barras
    print(numero)
```

```
  File "<celda 46>", line 1
    for numero in barras
                       ^
SyntaxError: expected ':'
```

El mensaje no empieza con `Traceback`, porque el código no llegó a ejecutarse: indica la línea y, con el signo `^`, el lugar donde Python esperaba los dos puntos.

El otro error de escritura frecuente es olvidar la sangría del bloque que sigue a un `for`:

```
for numero in barras:
print(numero)
```

```
  File "<celda 47>", line 2
    print(numero)
    ^^^^^
IndentationError: expected an indented block after 'for'
 ↪ statement on line 1
```

B.4. Datos equivocados

Si llamamos a una función con menos argumentos de los que necesita, el mensaje dice cuál falta. Aquí olvidamos la longitud de la barra:

```
k = rigidez_local(2.0e8, 0.0020)
```

```
Traceback (most recent call last):
  File "<celda 48>", line 1, in <module>
    k = rigidez_local(2.0e8, 0.0020)
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
TypeError: rigidez_local() missing 1 required positional
 ↪ argument: 'L'
```

Un `IndexError` indica que pedimos una posición que no existe. En la calculadora casi siempre se debe a confundir la numeración desde 1 con los índices desde 0. Queremos el desplazamiento vertical del nodo 3, que es el grado de libertad 6, y lo pedimos con `U[6]`:

```
resultados = analizar_armadura(nodos, barras,
                               apoyos, cargas)
U = resultados["U"]
uy_3 = U[6]
```

```
Traceback (most recent call last):
  File "<celda 49>", line 4, in <module>
    uy_3 = U[6]
           ~^^^
IndexError: index 6 is out of bounds for axis 0 with size 6
```

`U` tiene seis elementos, con índices de 0 a 5. El grado de libertad 6 está en la posición 5, y la forma segura de obtenerla es con la función `indices` del capítulo 6.

Una división por cero suele indicar dos nodos con las mismas coordenadas: la barra que los une tiene longitud nula.

```
nodos_mal = {
    1: {"x": 0.0, "y": 0.0},
    2: {"x": 6.0, "y": 0.0},
    3: {"x": 6.0, "y": 0.0},
}
geometria_mal = calcular_geometria(nodos_mal, barras)
```

```
Traceback (most recent call last):
  File "<celda 50>", line 6, in <module>
    geometria_mal = calcular_geometria(nodos_mal, barras)
                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "<celda 9>", line 15, in calcular_geometria
    "c": dx / L, "s": dy / L}
         ~~~^~~
ZeroDivisionError: float division by zero
```

B.5. Los errores de la calculadora

Algunos errores los produce la propia calculadora, a propósito, cuando detecta datos que no tienen solución. Si a la armadura le falta un apoyo, $\mathbf{K}_{ll}$ es singular y `resolver_desplazamientos` se detiene con un mensaje propio (capítulo 9):

```
apoyos_mal = {1: {"x": True, "y": True}}
resultados = analizar_armadura(nodos, barras,
                               apoyos_mal, cargas)
```

```
Traceback (most recent call last):
  File "<celda 51>", line 2, in <module>
    resultados = analizar_armadura(nodos, barras,
                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "<celda 37>", line 13, in analizar_armadura
    U_l = resolver_desplazamientos(
          ^^^^^^^^^^^^^^^^^^^^^^^^
  File "<celda 24>", line 4, in resolver_desplazamientos
    raise ValueError("La armadura es inestable: "
ValueError: La armadura es inestable: K_ll es singular.
```

La traza recorre tres pasos: nuestra celda llamó a `analizar_armadura`, esta llamó a `resolver_desplazamientos`, y ahí se detectó el problema. El mensaje está en español porque lo escribimos nosotros.

B.6. Los errores que no avisan

Los errores más peligrosos son los que no producen ningún mensaje: una carga en la unidad equivocada, una coordenada mal escrita o un apoyo en el nodo equivocado dan resultados, pero resultados falsos. Contra ellos solo sirve la verificación: comprobar el equilibrio, comparar el dibujo con la armadura y preguntarse si los números tienen sentido físico, como hicimos a lo largo del libro. La tabla 12.1 del capítulo 12 resume los síntomas más comunes.

Estrategia de resolución

Cuando aparece un error:

1. Leer la última línea: el tipo de error y su detalle.
2. Subir hasta el primer bloque y buscar la línea de nuestra celda que lo desencadenó.
3. Revisar los datos de esa línea: nombres, números de nodo y de barra, cantidad de argumentos.
4. Si el mensaje no basta, imprimir con `print` los valores que usa la línea, para ver cuál no es el que esperábamos.
5. Corregir, y ejecutar de nuevo la celda y las que dependen de ella.

Anexo C

Trabajar en local

Todo el libro puede seguirse con Google Colab, sin instalar nada. Este anexo es opcional: está pensado para quien quiera usar la calculadora en su propio computador, sin conexión a internet, o incorporarla en programas más grandes.

C.1. Qué necesitamos

Para trabajar en local necesitamos tres cosas:

- **Python**, la misma versión que usa Colab (3.12 al escribir este libro) o una más reciente.
- **Los paquetes** que usa la calculadora: NumPy y Matplotlib, y Jupyter si queremos trabajar con notebooks.
- **Un editor.** Recomendamos Visual Studio Code, gratuito, con las extensiones *Python* y *Jupyter*: abre los notebooks del libro igual que Colab.

C.2. El problema de los entornos

Instalar Python es fácil; lo que suele complicarse es instalar los paquetes. Si instalamos todos los paquetes de todos nuestros proyectos en una misma instalación de Python, tarde o temprano dos proyectos necesitarán versiones distintas del mismo paquete, y uno de los dos dejará de funcionar. Además, es frecuente terminar con varias instalaciones de Python en el mismo computador sin saber cuál se está usando.

La solución es un *entorno virtual* (*virtual environment*): una carpeta con su propia copia de Python y sus propios paquetes, separada del resto del sistema. Cada proyecto tiene su entorno, y lo que instalamos en uno no afecta a los demás. Hay tres caminos habituales para crearlos:

Anaconda. Una distribución que trae Python, cientos de paquetes y su propio gestor de entornos, `conda`. Es cómoda, pero ocupa varios gigabytes, y mezclar `conda` con `pip` en un mismo entorno es una fuente frecuente de conflictos. Sus condiciones de uso para organizaciones han cambiado en los últimos años; conviene revisarlas antes de usarla en una empresa.

Python, `venv` y `pip`. El camino clásico: Python descargado de `https://www.python.org`, entornos con el módulo `venv` y paquetes con `pip`, que vienen incluidos.

uv. Una herramienta más reciente que instala Python, crea el entorno e instala los paquetes con un solo programa, y lo hace muy rápido. Es la que usamos para escribir este libro.

Para un proyecto pequeño como la calculadora, que solo necesita NumPy y Matplotlib, recomendamos uv o el camino clásico.

C.3. Con uv

Instalamos uv siguiendo las instrucciones de su sitio, `https://docs.astral.sh/uv`. Después, en una terminal, creamos un proyecto para la calculadora e instalamos sus paquetes:

```
uv init mi-calculadora
cd mi-calculadora
uv add numpy matplotlib jupyter
```

uv descarga la versión de Python que haga falta y crea el entorno dentro de la carpeta del proyecto. Para abrir Jupyter en el navegador, o para ejecutar un archivo de Python con el entorno del proyecto:

```
uv run jupyter lab
uv run python mi_armadura.py
```

C.4. Con venv y pip

Instalamos Python desde `https://www.python.org`. En Windows, durante la instalación, marcamos la opción *Add python.exe to PATH*. Después, en una terminal abierta en la carpeta del proyecto, creamos el entorno y lo activamos. En Windows:

```
python -m venv .venv
.venv\Scripts\activate
```

En macOS o Linux:

```
python3 -m venv .venv
source .venv/bin/activate
```

Con el entorno activo, el nombre `(.venv)` aparece al comienzo de la línea de la terminal, e instalamos los paquetes:

```
pip install numpy matplotlib jupyter
```

El entorno debe activarse de nuevo cada vez que abrimos una terminal. Visual Studio Code lo detecta solo si la carpeta `.venv` está en el proyecto.

C.5. La calculadora como módulo

En Colab, el motor de la calculadora es la primera celda de nuestro notebook. En local podemos guardarlo en un archivo aparte y usarlo desde cualquier notebook o programa: un archivo `.py` con funciones que se importa desde otros archivos se llama *módulo* (*module*), como `math` o NumPy.

El repositorio del libro incluye ese archivo, `calculadora.py`, en la carpeta `notebooks`. Contiene exactamente el mismo código que la sección Motor de `mi_calculadora.ipynb` y se genera a partir de él, de modo que siempre coinciden. Lo copiamos en la carpeta de nuestro proyecto y, en un archivo o notebook de la misma carpeta, importamos las funciones que necesitamos:

```
from calculadora import (analizar_armadura,
                         mostrar_resultados,
                         verificar_resultados,
                         graficar_resultados)

nodos = {1: {"x": 0.0, "y": 0.0},
         2: {"x": 6.0, "y": 0.0},
         3: {"x": 3.0, "y": 4.0}}
# ... barras, apoyos y cargas, como en el capítulo 3

resultados = analizar_armadura(nodos, barras,
                               apoyos, cargas)
verificar_resultados(nodos, barras, resultados)
mostrar_resultados(nodos, barras, resultados)
graficar_resultados(nodos, barras, resultados)
```

Si lo ejecutamos como programa, con `uv run python mi_armadura.py` o `python mi_armadura.py`, los resultados se imprimen en la terminal y los gráficos se abren en una ventana.

> **Para saber más**
>
> Separar el motor en un módulo es el primer paso para convertir la calculadora en un programa más grande: leer los datos de una planilla, analizar muchas armaduras en un bucle o comparar alternativas de diseño. El motor no cambia; solo cambia el programa que lo usa.

Anexo D

Validación con OpenSees

A lo largo del libro verificamos la calculadora con el cálculo a mano (ejemplo A), con propiedades que deben cumplirse (simetría, equilibrio) y con el método de las secciones (ejemplo C). Este anexo agrega una verificación independiente: analizar los ejemplos B y C con otro programa, escrito por otras personas, y comparar los resultados.

D.1. OpenSees

OpenSees es un programa de análisis estructural de código abierto, desarrollado en la Universidad de California en Berkeley y muy usado en la investigación en ingeniería sísmica; su diseño se describe en McKenna (2011). Es gratuito y se usa desde Python con la biblioteca `openseespy`, de modo que cualquiera puede repetir esta comparación en Colab.

Para validar la calculadora, OpenSees tiene una ventaja sobre un programa comercial de pórticos: su elemento `Truss` es exactamente la barra de armadura del libro, articulada en sus extremos y con solo fuerza axial. No hay que liberar momentos ni anular el peso propio; basta con describir la misma armadura.

D.2. El modelo en OpenSees

La función `analizar_con_opensees` recibe los mismos diccionarios que `analizar_armadura` (capítulo 12) y devuelve los desplazamientos, las reacciones y las fuerzas en el mismo formato, para compararlos directamente:

```
import openseespy.opensees as ops


def analizar_con_opensees(nodos, barras, apoyos, cargas):
    """Analiza la armadura con OpenSees. Recibe los mismos
    datos que analizar_armadura y devuelve U, R y N en
    el mismo formato, para compararlos."""
```

```
    ops.wipe()  # borra el modelo anterior
    ops.model("basic", "-ndm", 2, "-ndf", 2)
    for nodo, punto in nodos.items():
        ops.node(nodo, punto["x"], punto["y"])
    for nodo, apoyo in apoyos.items():
        ops.fix(nodo, int(apoyo["x"]), int(apoyo["y"]))
    for numero, barra in barras.items():
        # Un material elástico por barra, con su E
        ops.uniaxialMaterial("Elastic", numero,
                             barra["elasticidad"])
        ops.element("Truss", numero,
                    barra["nodo_inicial"],
                    barra["nodo_final"], barra["area"],
                    numero)
    ops.timeSeries("Linear", 1)
    ops.pattern("Plain", 1, 1)
    for nodo, carga in cargas.items():
        ops.load(nodo, carga["x"], carga["y"])

    # Análisis estático y lineal, en un solo paso
    ops.system("BandSPD")
    ops.numberer("RCM")
    ops.constraints("Plain")
    ops.integrator("LoadControl", 1.0)
    ops.algorithm("Linear")
    ops.analysis("Static")
    ops.analyze(1)
    ops.reactions()

    # Los resultados, ordenados como los de la calculadora
    U = np.zeros(2 * len(nodos))
    for nodo in nodos:
        ix, iy = indices(gdl_del_nodo(nodo))
        U[ix], U[iy] = ops.nodeDisp(nodo)
    _, restringidos = clasificar_gdl(nodos, apoyos)
    R = []
    for g in restringidos:
        nodo = (g + 1) // 2
        direccion = 1 if g % 2 == 1 else 2  # x o y
        R.append(ops.nodeReaction(nodo, direccion))
    N = {}
    for numero in barras:
        fuerza = ops.eleResponse(numero, "axialForce")
        N[numero] = fuerza[0]
    return {"U": U, "R": np.array(R), "N": N,
            "restringidos": restringidos}
```

Cada paso tiene su equivalente en la calculadora. `ops.node` y `ops.fix` de-

finen los nodos y los apoyos (con `1` para un grado de libertad restringido); `ops.element("Truss", ...)` crea cada barra con su área y un material elástico con su módulo E; `ops.load` aplica las cargas. Las seis líneas siguientes eligen cómo resolver el sistema (un análisis estático y lineal en un solo paso, que equivale a resolver $\mathbf{K}_{ll}\mathbf{U}_l = \mathbf{F}_l$ una vez), `ops.analyze(1)` lo resuelve y `ops.reactions()` calcula las reacciones. Al final, los resultados se ordenan como los de la calculadora: `clasificar_gdl` solo se usa para listar las reacciones en el mismo orden.

La función `comparar_resultados` muestra los dos análisis lado a lado:

```
def comparar_resultados(nodos, calculadora, opensees,
                        tolerancia=1e-9):
    """Muestra los dos análisis lado a lado e indica si
    coinciden."""
    filas = []
    for nodo in nodos:
        for direccion, g in zip("xy", gdl_del_nodo(nodo)):
            i = indices([g])[0]
            filas.append([f"u{direccion} {nodo} (mm)",
                          calculadora["U"][i] * 1000,
                          opensees["U"][i] * 1000])
    restringidos = calculadora["restringidos"]
    for posicion, g in enumerate(restringidos):
        filas.append([f"R{g} (kN)",
                      calculadora["R"][posicion],
                      opensees["R"][posicion]])
    for numero in calculadora["N"]:
        filas.append([f"N{numero} (kN)",
                      calculadora["N"][numero],
                      opensees["N"][numero]])
    mostrar_tabla(["Resultado", "Calculadora",
                   "OpenSees"], filas, decimales=3,
                  ancho=13)
    diferencia = 0.0
    for _, valor_calculadora, valor_opensees in filas:
        diferencia = max(diferencia, abs(
            valor_calculadora - valor_opensees))
    if diferencia < tolerancia:
        print("\nLos dos análisis coinciden.")
    else:
        print(f"\nDifieren hasta en {diferencia:.3g}.")
```

```
resultados_b = analizar_armadura(nodos_b, barras_b,
                                 apoyos_b, cargas_b)
opensees_b = analizar_con_opensees(nodos_b, barras_b,
                                   apoyos_b, cargas_b)
comparar_resultados(nodos_b, resultados_b, opensees_b)
```

```
   Resultado  Calculadora     OpenSees
   ux 1 (mm)        0.000        0.000
   uy 1 (mm)        0.000        0.000
   ux 2 (mm)        0.179        0.179
   uy 2 (mm)        0.000        0.000
   ux 3 (mm)        0.900        0.900
   uy 3 (mm)       -0.081       -0.081
   ux 4 (mm)        0.854        0.854
   uy 4 (mm)       -0.481       -0.481
     R1 (kN)      -30.000      -30.000
     R2 (kN)        0.000        0.000
     R4 (kN)       80.000       80.000
     N1 (kN)       23.889       23.889
     N2 (kN)      -48.148      -48.148
     N3 (kN)       -6.111       -6.111
     N4 (kN)       -8.148       -8.148
     N5 (kN)       10.185       10.185
     N6 (kN)      -39.815      -39.815

Los dos análisis coinciden.
```

Validación con OpenSees

El notebook con la calculadora, los ejemplos B y C y su análisis con OpenSees, para repetir la comparación en Colab:

Repositorio:
`github.com/eder3232/libro-rigidez-python-armaduras2d`
Notebook: `notebooks/validacion_opensees.ipynb`

D.3. Ejemplo B

El ejemplo B (figura 3.2) es hiperestático, de modo que sus fuerzas dependen de la rigidez de las barras: es la prueba más exigente para la calculadora. La tabla D.1 compara los desplazamientos, y la tabla D.2, las reacciones y las fuerzas axiales.

Tabla D.1: Ejemplo B: desplazamientos de los nodos, en milímetros.

Nodo	u_x calculadora	u_x OpenSees	u_y calculadora	u_y OpenSees
1	0.000	0.000	0.000	0.000
2	0.179	0.179	0.000	0.000
3	0.900	0.900	−0.081	−0.081
4	0.854	0.854	−0.481	−0.481

Tabla D.2: Ejemplo B: reacciones y fuerzas axiales, en kilonewtons.

Resultado	Calculadora	OpenSees
Reacción x del nodo 1	−30.00	−30.00
Reacción y del nodo 1	0.00	0.00
Reacción y del nodo 2	80.00	80.00
Barra 1	23.89	23.89
Barra 2	−48.15	−48.15
Barra 3	−6.11	−6.11
Barra 4	−8.15	−8.15
Barra 5	10.19	10.19
Barra 6	−39.81	−39.81

Los resultados coinciden en todas las cifras que muestran las tablas. La figura D.1 lo muestra también a la vista: la deformada de OpenSees se dibuja con un trazo ancho y translúcido, y la de la calculadora, con uno fino encima. Si fueran distintas, el trazo fino se saldría del ancho; en cambio, corre por su centro en todas las barras. La figura D.2 es el diagrama de fuerzas axiales de OpenSees, con el grosor de cada barra proporcional a su fuerza: es el mismo que dibuja nuestra calculadora (figura 10.5).

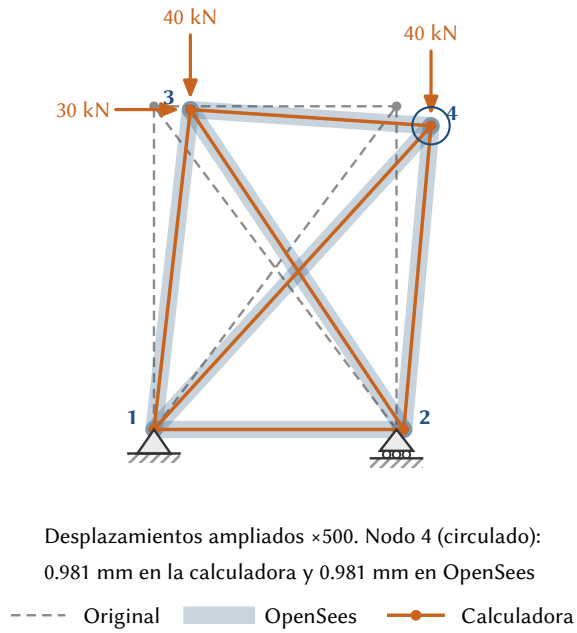


Figura D.1: Ejemplo B: la deformada de OpenSees (trazo ancho) y la de la calculadora (trazo fino) coinciden.

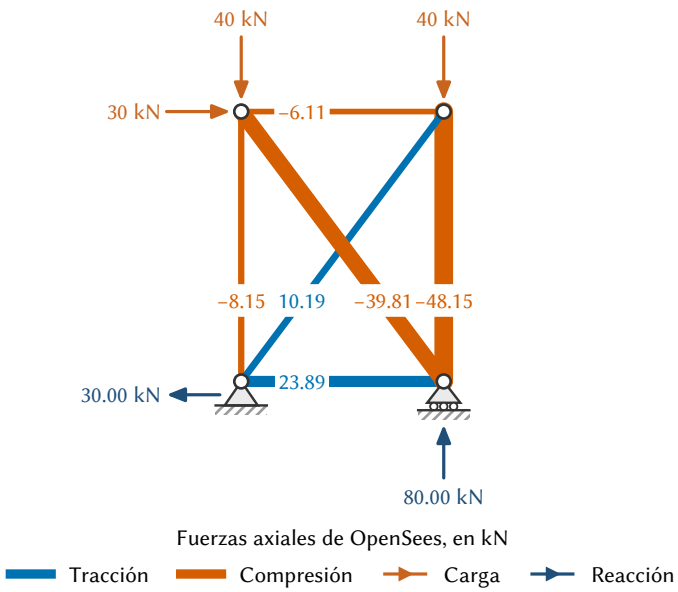


Figura D.2: Ejemplo B: fuerzas axiales y reacciones calculadas con OpenSees.

D.4. Ejemplo C

El ejemplo C (figura 12.2) es isostático: sus fuerzas dependen solo del equilibrio, y ya las verificamos con el método de las secciones en el capítulo 12. Compararlas con OpenSees confirma que ambos modelos tienen la misma geometría y las mismas cargas; la flecha, en cambio, depende de las áreas y del módulo de elasticidad, y verifica las rigideces.

Tabla D.3: Ejemplo C: flecha y reacciones.

Resultado	Calculadora	OpenSees
Desplazamiento u_y del nodo 4 (mm)	−6.736	−6.736
Reacción y del nodo 1 (kN)	62.50	62.50
Reacción y del nodo 7 (kN)	62.50	62.50

Tabla D.4: Ejemplo C: fuerzas axiales, en kilonewtons.

Barra	Calculadora	OpenSees	Barra	Calculadora	OpenSees
1	62.50	62.50	12	−88.39	−88.39
2	62.50	62.50	13	25.00	25.00
3	100.00	100.00	14	−12.50	−12.50
4	100.00	100.00	15	0.00	0.00
5	62.50	62.50	16	−12.50	−12.50
6	62.50	62.50	17	25.00	25.00
7	−100.00	−100.00	18	53.03	53.03
8	−112.50	−112.50	19	17.68	17.68
9	−112.50	−112.50	20	17.68	17.68
10	−100.00	−100.00	21	53.03	53.03
11	−88.39	−88.39			

También aquí los resultados coinciden. En la deformada (figura D.3), el trazo fino de la calculadora recorre el centro del de OpenSees en las 21 barras, y el diagrama de fuerzas axiales (figura D.4) muestra el patrón típico de una armadura Pratt: el cordón superior comprimido, el inferior traccionado, y el montante central sin fuerza.

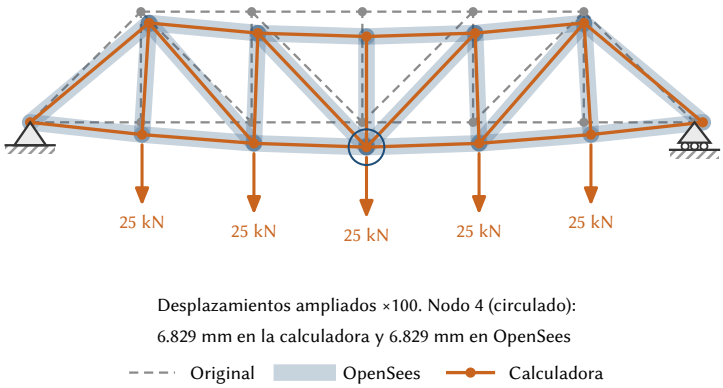


Figura D.3: Ejemplo C: las deformadas de OpenSees y de la calculadora coinciden.

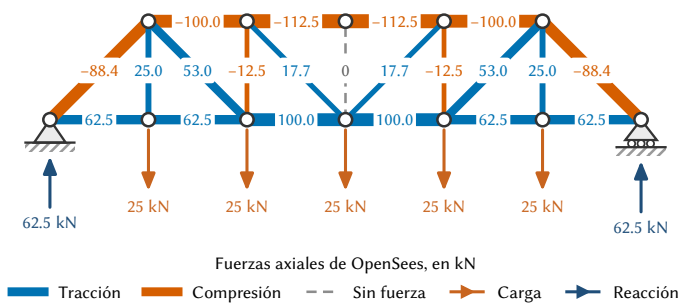


Figura D.4: Ejemplo C: fuerzas axiales y reacciones calculadas con OpenSees.

D.5. Todas las comparaciones de un vistazo

Las tablas muestran dos o tres decimales, pero la comparación puede ir mucho más lejos. La figura D.5 reúne todos los resultados de los dos ejemplos: los desplazamientos de todos los nodos, las reacciones y las fuerzas de todas las barras. Para cada uno dibuja la *diferencia relativa* entre los dos programas: la diferencia dividida por el mayor valor de su grupo, en escala logarítmica.

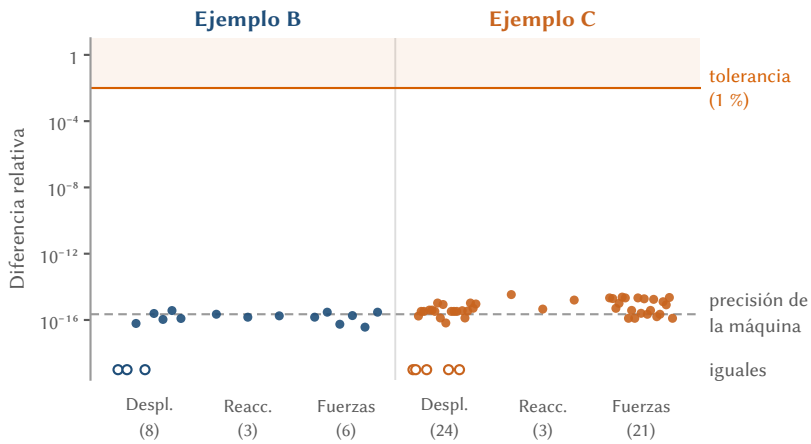


Figura D.5: Diferencia relativa entre la calculadora y OpenSees en cada resultado de los ejemplos B y C (entre paréntesis, cuántos hay de cada tipo). Los círculos vacíos son resultados exactamente iguales.

Una validación razonable aceptaría diferencias de hasta el 1 % (la franja sombreada); las de nuestra calculadora están unos catorce órdenes de magnitud por debajo, alrededor de 10^{-16}. Ese número no es casual: es la *precisión de la máquina*, el error de redondeo de los números decimales de 64 bits con los que calculan Python y OpenSees. Dos programas que resuelven el mismo sistema por caminos distintos solo pueden diferir en ese redondeo, y eso es lo que encontramos.

La figura D.6 muestra lo mismo de otra forma: cada resultado es un punto, con el valor de la calculadora en el eje horizontal y el de OpenSees en el vertical. Si coinciden, el punto cae sobre la recta $y = x$; los 65 resultados están sobre ella.

> **Para saber más**
>
> **Validar con un programa comercial.** La misma comparación puede hacerse con SAP2000, ETABS u otro programa de pórticos, cuidando que el modelo sea el de una armadura. Hay que liberar los momentos en los extremos de cada

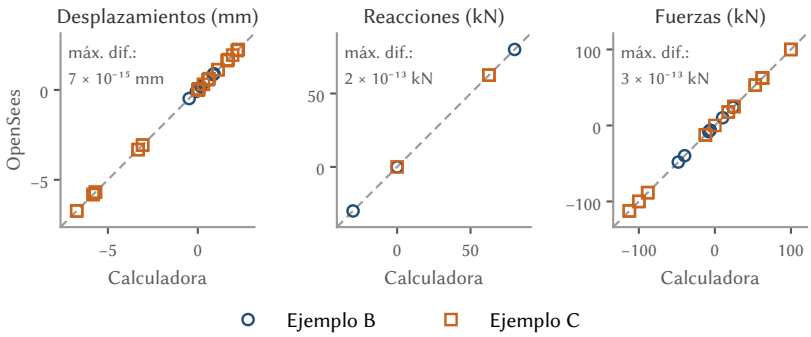


Figura D.6: Cada resultado de los ejemplos B y C, calculado con la calculadora y con OpenSees: todos caen sobre la recta $y = x$.

> barra (o analizar el modelo como armadura plana), anular el peso propio, usar el mismo E y las mismas áreas, y trabajar en kilonewtons y metros. Algunos programas, como SAP2000, dibujan en el plano xz: su eje z es nuestro eje y. Si el modelo está bien, los resultados coinciden en todas sus cifras significativas; una diferencia mayor casi siempre se debe a un apoyo, a las unidades o al peso propio.

D.6. Conclusión

Para una armadura hiperestática y para una isostática de 21 barras, la calculadora que construimos en este libro reproduce los desplazamientos, las reacciones y las fuerzas axiales de OpenSees hasta el redondeo de la máquina. No es casualidad: ambos resuelven el mismo sistema $\mathbf{K}\,\mathbf{U} = \mathbf{F}$ con el mismo método. La diferencia es que ahora sabemos qué hace cada paso.

Anexo E

Notación y convenciones de signos

Este anexo reúne la notación, las convenciones de signos y las unidades del libro, y la correspondencia entre cada símbolo y su nombre en la calculadora. Las matrices y los vectores se escriben en negrita, y los escalares en cursiva.

E.1. Símbolos

Tabla E.1: Datos y geometría.

Símbolo	Significado	Unidad	Cap.
n, b, r	Cantidad de nodos, barras y reacciones	—	6
$g = b + r - 2n$	Grado de indeterminación	—	6
(x_i, y_i)	Coordenadas del nodo i	m	2
$\Delta x, \Delta y$	Proyecciones de una barra	m	2
L	Longitud de una barra	m	2
θ	Ángulo de una barra con el eje x	—	2
c, s	Cosenos directores, $\cos\theta$ y $\sin\theta$	—	2
E	Módulo de elasticidad	kN/m^2	5
A	Área de la sección	m^2	5
EA/L	Rigidez axial de una barra	kN/m	5

E.2. Convenciones de signos

- **Ejes globales:** x hacia la derecha e y hacia arriba.
- **Ángulo de una barra:** θ se mide desde el eje x, en sentido antihorario, hasta la dirección que va del nodo inicial al nodo final (figura 2.1).

Tabla E.2: Matrices, vectores y resultados.

Símbolo	Significado	Cap.
$\mathbf{k}'$	Rigidez de una barra en coordenadas locales (2×2)	5
$\mathbf{T}$	Matriz de transformación (2×4)	5
$\mathbf{k} = \mathbf{T}^\mathsf{T}\,\mathbf{k}'\,\mathbf{T}$	Rigidez de una barra en coordenadas globales (4×4)	5
$\mathbf{u}'$, $\mathbf{u}$	Desplazamientos de los extremos de una barra	5
$2i-1$, $2i$	Grados de libertad x e y del nodo i	6
$\mathbf{K}$	Matriz de rigidez de la armadura ($2n \times 2n$)	7
$\mathbf{U}$, $\mathbf{F}$	Desplazamientos de los nodos y cargas aplicadas	7 y 8
l, r	Subíndices: grados de libertad libres y restringidos	8
$\mathbf{K}_{ll}$, $\mathbf{K}_{lr}$, $\mathbf{K}_{rl}$, $\mathbf{K}_{rr}$	Bloques de la partición de $\mathbf{K}$	8
$\mathbf{R}$	Reacciones de los apoyos	9
N	Fuerza axial de una barra	10
$\sigma = N/A$	Esfuerzo axial de una barra	10

Cambiar el orden de los nodos cambia el signo de c y s, pero no la matriz $\mathbf{k}$ ni la fuerza axial.

- **Desplazamientos, cargas y reacciones:** positivos en el sentido positivo de los ejes globales. Una carga hacia abajo es negativa.
- **Grados de libertad de una barra:** u_1 y u_2 son los desplazamientos x e y del nodo inicial; u_3 y u_4, los del nodo final (figura 5.1).
- **Fuerza axial:** positiva en tracción y negativa en compresión (figura 10.2).
- **Numeración:** nodos, barras y grados de libertad se numeran desde 1. La función `indices` es el único lugar de la calculadora que traduce esa numeración a los índices de NumPy, que empiezan en 0.

E.3. Unidades

Los cálculos se hacen siempre en kilonewtons y metros. Los resultados se presentan en las unidades más legibles para cada magnitud.

Las equivalencias más útiles son:

- $1\,\text{GPa} = 10^6\,\text{kN/m}^2$, de modo que $E = 200\,\text{GPa} = 2.0 \times 10^8\,\text{kN/m}^2$;
- $1\,\text{cm}^2 = 10^{-4}\,\text{m}^2$, de modo que $A = 20\,\text{cm}^2 = 0.0020\,\text{m}^2$;
- $1\,\text{MPa} = 1000\,\text{kN/m}^2$.

Tabla E.3: Unidades de cálculo y de presentación.

Magnitud	En los cálculos	Al presentar
Longitudes y coordenadas	m	m
Áreas	m^2	cm^2
Módulo de elasticidad	kN/m^2	GPa
Fuerzas, cargas y reacciones	kN	kN
Desplazamientos	m	mm
Esfuerzos	kN/m^2	MPa

E.4. Del símbolo al código

La tabla E.4 indica cómo se llama cada símbolo en la calculadora. En todas las funciones, los desplazamientos están en metros; las funciones que los muestran, como `mostrar_resultados`, los convierten a milímetros solo al imprimirlos.

Tabla E.4: Cada símbolo y su nombre en la calculadora.

Símbolo	En la calculadora
(x_i, y_i)	`nodos[i]["x"]`, `nodos[i]["y"]`
E, A	`barras[n]["elasticidad"]`, `barras[n]["area"]`
L	`geometria[n]["longitud"]`
c, s	`geometria[n]["c"]`, `geometria[n]["s"]`
$\mathbf{k}'$	`rigidez_local(E, A, L)`
$\mathbf{T}$	`matriz_transformacion(c, s)`
$\mathbf{k}$	`rigideces[n]`
$\mathbf{K}$, $\mathbf{F}$, $\mathbf{U}$	`K`, `F`, `U`
Grados de libertad l y r	`libres`, `restringidos`
$\mathbf{K}_{ll}$, $\mathbf{K}_{lr}$, $\mathbf{K}_{rl}$, $\mathbf{K}_{rr}$	`K_ll`, `K_lr`, `K_rl`, `K_rr`
$\mathbf{U}_l$, $\mathbf{U}_r$, $\mathbf{F}_l$, $\mathbf{F}_r$	`U_l`, `U_r`, `F_l`, `F_r`
$\mathbf{R}$	`R`
N de la barra n	`N[n]`

Anexo F

Soluciones de los ejercicios

Cada solución se presenta con el código que la resuelve y su resultado. El mismo código está en el notebook de soluciones de cada capítulo.

Capítulo 1

Ejercicio 1.1

Con un apoyo articulado (2 reacciones) y uno de rodillo (1 reacción), los tres ejemplos tienen $r = 3$:

$$g_A = 3 + 3 - 2 \cdot 3 = 0, \qquad g_B = 6 + 3 - 2 \cdot 4 = 1, \qquad g_C = 21 + 3 - 2 \cdot 12 = 0.$$

Los ejemplos A y C son isostáticos y pueden resolverse con el método de los nudos; el B es hiperestático de grado 1 y no.

Ejercicio 1.2

La matriz $\mathbf{K}$ tiene una fila y una columna por cada grado de libertad, es decir, $2n$: es de 6×6 en el ejemplo A, de 8×8 en el B y de 24×24 en el C. La del ejemplo C tiene $24^2 = 576$ números, aunque, como veremos, la mayoría son ceros.

Ejercicio 1.3

Un puente de p paneles tiene $p + 1$ nodos en el cordón inferior y $p - 1$ en el superior, es decir, $n = 2p$. Sus barras son p del cordón inferior, $p - 2$ del superior, $p - 1$ montantes y p diagonales (dos extremas y $p - 2$ interiores):

$$b = p + (p - 2) + (p - 1) + p = 4p - 3.$$

Para $p = 6$ resultan 12 nodos y 21 barras, como en el ejemplo C. Con $r = 3$, el grado de indeterminación es $g = (4p - 3) + 3 - 2 \cdot 2p = 0$: el puente es isostático para cualquier cantidad de paneles. Con 20 paneles tiene 40 nodos y 80 grados de libertad, y su matriz $\mathbf{K}$ es de 80×80, con 6400 números.

Capítulo 2

Ejercicio 2.1

```
# Barra 1 del ejemplo A: de (0, 0) a (6, 0)
dx = 6.0 - 0.0
dy = 0.0 - 0.0
L = math.sqrt(dx**2 + dy**2)
c = dx / L
s = dy / L
print(f"L = {L:.3f} m")
print(f"c = {c:.4f}, s = {s:.4f}")
```

```
L = 6.000 m
c = 1.0000, s = 0.0000
```

La barra es horizontal y apunta en el sentido positivo de x: su longitud es la diferencia de abscisas, $c = 1$ y $s = 0$.

Ejercicio 2.2

```
# Barra de cubierta: de (0, 0) a (4.5, 1.5)
dx = 4.5 - 0.0
dy = 1.5 - 0.0
L = math.sqrt(dx**2 + dy**2)
c = dx / L
s = dy / L
theta = math.degrees(math.atan2(dy, dx))
print(f"L = {L:.3f} m")
print(f"c = {c:.4f}, s = {s:.4f}")
print(f"θ = {theta:.2f}°")
```

```
L = 4.743 m
c = 0.9487, s = 0.3162
θ = 18.43°
```

Ejercicio 2.3

Con $E = 200\,\text{GPa} = 2.0 \times 10^8\,\text{kN/m}^2$ y $A = 20\,\text{cm}^2 = 0.0020\,\text{m}^2$:

```
E = 2.0e8  # kN/m²
A = 0.0020  # m²
L = 5.0  # m
rigidez_axial = E * A / L
print(f"EA/L = {rigidez_axial:.0f} kN/m")
```

```
EA/L = 80000 kN/m
```

Ejercicio 2.4

```
# Barra de (6, 0) a (0, 3): hacia la izquierda y arriba
dx = 0.0 - 6.0
dy = 3.0 - 0.0
con_atan = math.degrees(math.atan(dy / dx))
con_atan2 = math.degrees(math.atan2(dy, dx))
print(f"Con atan:  θ = {con_atan:.2f}°")
print(f"Con atan2: θ = {con_atan2:.2f}°")
```

```
Con atan:  θ = -26.57°
Con atan2: θ = 153.43°
```

La barra apunta hacia la izquierda y hacia arriba, de modo que su ángulo está en el segundo cuadrante: el correcto es el de `math.atan2`. El cociente $\Delta y/\Delta x = 3/(-6)$ es igual al de una barra que apunta hacia la derecha y hacia abajo, y `math.atan` no puede distinguir ambos casos.

Ejercicio 2.5

```
# Barra vertical: de (2, 3) a (2, -1)
dx = 2.0 - 2.0
dy = -1.0 - 3.0
L = math.sqrt(dx**2 + dy**2)
c = dx / L
s = dy / L
theta = math.degrees(math.atan2(dy, dx))
print(f"L = {L:.3f} m")
print(f"c = {c:.4f}, s = {s:.4f}")
print(f"θ = {theta:.2f}°")
```

```
L = 4.000 m
c = 0.0000, s = -1.0000
θ = -90.00°
```

Si se usa `math.atan(dy / dx)`, la división por $\Delta x = 0$ produce el error `ZeroDivisionError` y el programa se detiene.

Capítulo 3

Ejercicio 3.1

```
inicial = nodos[barras[3]["nodo_inicial"]]
final = nodos[barras[3]["nodo_final"]]
dx = final["x"] - inicial["x"]
dy = final["y"] - inicial["y"]
L = math.sqrt(dx**2 + dy**2)
print(f"L = {L:.3f} m")
```

```
L = 5.000 m
```

La consulta encadenada `nodos[barras[3]["nodo_inicial"]]` obtiene el nodo desde la barra, sin escribir su número a mano.

Ejercicio 3.2

```python
# Armadura de cubierta. Unidades: kN y m
nodos_cubierta = {
    1: {"x": 0.0, "y": 0.0},
    2: {"x": 4.0, "y": 0.0},
    3: {"x": 8.0, "y": 0.0},
    4: {"x": 4.0, "y": 1.5},
}

barras_cubierta = {
    1: {"nodo_inicial": 1, "nodo_final": 2,
        "elasticidad": 2.0e8, "area": 0.0015},
    2: {"nodo_inicial": 2, "nodo_final": 3,
        "elasticidad": 2.0e8, "area": 0.0015},
    3: {"nodo_inicial": 1, "nodo_final": 4,
        "elasticidad": 2.0e8, "area": 0.0015},
    4: {"nodo_inicial": 4, "nodo_final": 3,
        "elasticidad": 2.0e8, "area": 0.0015},
    5: {"nodo_inicial": 2, "nodo_final": 4,
        "elasticidad": 2.0e8, "area": 0.0010},
}

apoyos_cubierta = {
    1: {"x": True, "y": True},   # articulado
    3: {"x": False, "y": True},  # rodillo
}

cargas_cubierta = {
    4: {"x": 0.0, "y": -30.0},
}

print(f"Nodos: {len(nodos_cubierta)}")
print(f"Barras: {len(barras_cubierta)}")
```

```
Nodos: 4
Barras: 5
```

Ejercicio 3.3

```python
print(apoyos.get(3))
print(3 in cargas, 1 in cargas)
```

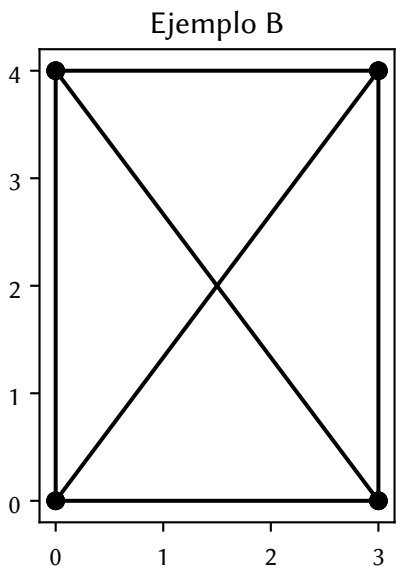
```
None
True False
```

El nodo 3 no tiene apoyo, de modo que `get` devuelve **`None`**. El nodo 3 tiene carga y el nodo 1 no.

Ejercicio 3.4

```
n1, n2 = nodos_b[1], nodos_b[2]
n3, n4 = nodos_b[3], nodos_b[4]

fig, ax = plt.subplots()
ax.plot([n1["x"], n2["x"]], [n1["y"], n2["y"]], "k-o")
ax.plot([n2["x"], n4["x"]], [n2["y"], n4["y"]], "k-o")
ax.plot([n3["x"], n4["x"]], [n3["y"], n4["y"]], "k-o")
ax.plot([n1["x"], n3["x"]], [n1["y"], n3["y"]], "k-o")
ax.plot([n1["x"], n4["x"]], [n1["y"], n4["y"]], "k-o")
ax.plot([n2["x"], n3["x"]], [n2["y"], n3["y"]], "k-o")
ax.set_aspect("equal")
ax.set_title("Ejemplo B")
plt.show()
```



Capítulo 4

Ejercicio 4.1

```
geometria_con_angulo = {}
for numero, barra in barras.items():
    inicial = nodos[barra["nodo_inicial"]]
    final = nodos[barra["nodo_final"]]
    dx = final["x"] - inicial["x"]
    dy = final["y"] - inicial["y"]
    theta = math.degrees(math.atan2(dy, dx))
    geometria_con_angulo[numero] = {"theta": theta}

for numero, g in geometria_con_angulo.items():
    print(f"Barra {numero}: θ = {g['theta']:.2f}°")
```

```
Barra 1: θ = 0.00°
Barra 2: θ = 126.87°
Barra 3: θ = 53.13°
```

Ejercicio 4.2

Usamos las longitudes ya calculadas en el diccionario `geometria`:

```
gamma = 78.5  # peso específico del acero, en kN/m³
peso_total = 0.0
for numero, barra in barras.items():
    L = geometria[numero]["longitud"]
    peso_total += gamma * barra["area"] * L
print(f"Peso propio: {peso_total:.3f} kN")
```

```
Peso propio: 2.512 kN
```

Ejercicio 4.3

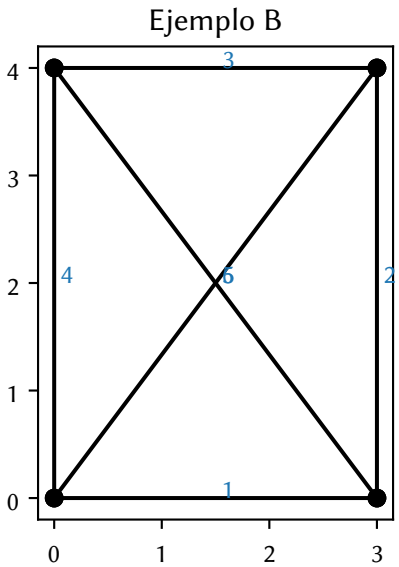
```
longitud_total_b = 0.0
for numero, barra in barras_b.items():
    inicial = nodos_b[barra["nodo_inicial"]]
    final = nodos_b[barra["nodo_final"]]
    dx = final["x"] - inicial["x"]
    dy = final["y"] - inicial["y"]
    longitud_total_b += math.sqrt(dx**2 + dy**2)
print(f"Longitud total: {longitud_total_b:.3f} m")
```

```
Longitud total: 24.000 m
```

El resultado coincide con la suma de las longitudes de la tabla del problema resuelto 4.1: $3 + 4 + 3 + 4 + 5 + 5 = 24\,\text{m}$.

Ejercicio 4.4

```
fig, ax = plt.subplots()
for numero, barra in barras_b.items():
    inicial = nodos_b[barra["nodo_inicial"]]
    final = nodos_b[barra["nodo_final"]]
    ax.plot([inicial["x"], final["x"]],
            [inicial["y"], final["y"]], "k-o")
    xm = (inicial["x"] + final["x"]) / 2
    ym = (inicial["y"] + final["y"]) / 2
    ax.text(xm, ym, f" {numero}", color="tab:blue")
ax.set_aspect("equal")
ax.set_title("Ejemplo B")
plt.show()
```



Los números de las diagonales 5 y 6 se superponen, porque ambas barras tienen el mismo punto medio. Una mejora sencilla es ubicar cada número en otro punto de su barra, por ejemplo al 40 % de su longitud desde el nodo inicial: `xm = inicial["x"] + 0.4 * (final["x"] - inicial["x"])`, y lo mismo para `ym`.

Capítulo 5

Ejercicio 5.1

```
def peso_barra(A, L, gamma=78.5):
    """Peso propio de una barra, en kN."""
```

```
    return gamma * A * L


peso_total = 0.0
for numero, barra in barras.items():
    L = geometria[numero]["longitud"]
    peso_total += peso_barra(barra["area"], L)
print(f"Peso propio: {peso_total:.3f} kN")
```

```
Peso propio: 2.512 kN
```

El resultado coincide con el del ejercicio 4.2, ahora con una función reutilizable.

Ejercicio 5.2

```
g = geometria[3]
c, s = g["c"], g["s"]
# El nodo final se mueve 1 mm perpendicular a la barra
perpendicular = 0.001 * np.array([0.0, 0.0, -s, c])
rigideces[3] @ perpendicular
```

```
array([ 0.,  0., -0., -0.])
```

Un desplazamiento perpendicular a la barra no cambia su longitud, de modo que no la deforma axialmente y no produce fuerzas. Los valores `-0.` son ceros: el signo es un detalle de la aritmética de punto flotante. Este movimiento corresponde a una pequeña rotación de la barra alrededor de su nodo inicial, otro movimiento de cuerpo rígido.

Ejercicio 5.3

```
# Barra vertical de 3 m: de (0, 0) a (0, 3)
k_vertical = rigidez_barra(2.0e8, 0.0010, 3.0, 0.0, 1.0)
k_vertical
```

```
array([[     0. ,      0. ,      0. ,      0. ],
       [     0. ,  66666.7,      0. , -66666.7],
       [     0. ,      0. ,      0. ,      0. ],
       [     0. , -66666.7,      0. ,  66666.7]])
```

Con $c = 0$ y $s = 1$, solo sobreviven los términos en s^2, asociados a los desplazamientos verticales: una barra vertical solo resiste desplazamientos en su propia dirección. $EA/L = 2.0 \times 10^8 \cdot 0.0010/3 = 66\,666.7\,\text{kN/m}$.

Ejercicio 5.4

```
geometria_b = calcular_geometria(nodos_b, barras_b)
rigideces_b = calcular_rigideces(barras_b, geometria_b)
for numero, k in rigideces_b.items():
    simetrica = np.allclose(k, k.T)
    rango = np.linalg.matrix_rank(k)
    print(f"Barra {numero}: simétrica = {simetrica}, "
          f"rango = {rango}")
```

```
Barra 1: simétrica = True, rango = 1
Barra 2: simétrica = True, rango = 1
Barra 3: simétrica = True, rango = 1
Barra 4: simétrica = True, rango = 1
Barra 5: simétrica = True, rango = 1
Barra 6: simétrica = True, rango = 1
```

Capítulo 6

Ejercicio 6.1

```
# La barra 6 va del nodo 2 al nodo 3
gdl_de_barra(barras_b[6])
```

```
[3, 4, 5, 6]
```

Los grados de libertad del nodo 2 son 3 y 4, y los del nodo 3, 5 y 6.

Ejercicio 6.2

```
apoyos_articulados = {
    1: {"x": True, "y": True},
    2: {"x": True, "y": True},
}
libres_2, restringidos_2 = clasificar_gdl(
    nodos, apoyos_articulados)
print(f"Libres:       {libres_2}")
print(f"Restringidos: {restringidos_2}")
grado = indeterminacion(nodos, barras, restringidos_2)
print(f"Grado de indeterminación: {grado}")
```

```
Libres:       [5, 6]
Restringidos: [1, 2, 3, 4]
Grado de indeterminación: 1
```

Con dos apoyos articulados hay 4 reacciones, y $g = 3 + 4 - 6 = 1$: la armadura pasa a ser hiperestática. Solo el nodo 3 puede desplazarse.

Ejercicio 6.3

```
barras_sin_1 = {2: barras[2], 3: barras[3]}
grado = indeterminacion(nodos, barras_sin_1, restringidos)
print(f"Grado de indeterminación: {grado}")
```

```
Grado de indeterminación: -1
```

Con $g = -1$ la armadura es inestable: sin la barra 1, nada impide que el rodillo del nodo 2 se deslice horizontalmente y que las barras 2 y 3 giren como un mecanismo.

Ejercicio 6.4

```
nueva_numeracion = libres + restringidos
for nuevo, original in enumerate(nueva_numeracion, 1):
    print(f"GDL {original} → {nuevo}")
```

```
GDL 3 → 1
GDL 5 → 2
GDL 6 → 3
GDL 1 → 4
GDL 2 → 5
GDL 4 → 6
```

Los grados de libertad libres 3, 5 y 6 pasan a ser los primeros (1, 2 y 3), y los restringidos 1, 2 y 4, los últimos (4, 5 y 6).

Capítulo 7

Ejercicio 7.1

Al nodo 3 llegan las barras 2 y 3. En ambas, el grado de libertad 6 es el cuarto (u_4), y su término es $EA/L\,s^2 = 80 \cdot 0.64 = 51.2\,\text{kN/mm}$:

```
# Al gdl 6 (nodo 3, dirección y) llegan las barras 2 y 3
# Su gdl local es u4 (posición 3 de la matriz de 4 x 4)
aporte_2 = rigideces[2][3, 3] / 1000
aporte_3 = rigideces[3][3, 3] / 1000
fila, columna = indices([6, 6])
termino = K_global[fila, columna] / 1000
print(f"Barra 2: {aporte_2:.1f} kN/mm")
print(f"Barra 3: {aporte_3:.1f} kN/mm")
print(f"Suma:    {aporte_2 + aporte_3:.1f} kN/mm")
print(f"K:       {termino:.1f} kN/mm")
```

```
Barra 2: 51.2 kN/mm
Barra 3: 51.2 kN/mm
Suma:    102.4 kN/mm
K:       102.4 kN/mm
```

Ejercicio 7.2

```
solo_barra_2 = {2: barras[2]}
K_2 = ensamblar_rigidez(nodos, solo_barra_2, rigideces)
mostrar_matriz(K_2 / 1000, list(range(1, 7)))
```

gdl	1	2	3	4	5	6
1	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	28.8	-38.4	-28.8	38.4
4	0.0	0.0	-38.4	51.2	38.4	-51.2
5	0.0	0.0	-28.8	38.4	28.8	-38.4
6	0.0	0.0	38.4	-51.2	-38.4	51.2

Solo el bloque de filas y columnas 3 a 6 es distinto de cero, como en la figura 7.1.

Ejercicio 7.3

```
traslacion_y = np.array([0.0, 1.0, 0.0, 1.0, 0.0, 1.0])
K_global @ traslacion_y
```

```
array([0., 0., 0., 0., 0., 0.])
```

Ejercicio 7.4

```
al_reves = {3: barras[3], 2: barras[2], 1: barras[1]}
K_al_reves = ensamblar_rigidez(nodos, al_reves, rigideces)
np.allclose(K_al_reves, K_global)
```

```
True
```

La suma es conmutativa: el orden en que las barras aportan sus términos no cambia el resultado.

Capítulo 8

Ejercicio 8.1

```
F_b = ensamblar_cargas(nodos_b, cargas_b)
print(f"Ejemplo B, en x: {F_b[0::2].sum():7.2f} kN")
print(f"Ejemplo B, en y: {F_b[1::2].sum():7.2f} kN")
```

```
Ejemplo B, en x:   30.00 kN
Ejemplo B, en y:  -80.00 kN
```

La resultante es de 30 kN hacia la derecha y 80 kN hacia abajo, la suma de las tres cargas de la figura.

Ejercicio 8.2

Creamos un diccionario nuevo con las dos cargas, para no modificar los datos del ejemplo:

```
cargas_2 = {
    2: {"x": 15.0, "y": 0.0},
    3: {"x": 20.0, "y": -50.0},
}
F_2 = ensamblar_cargas(nodos, cargas_2)
print(" gdl  F (kN)")
mostrar_vector(F_2, list(range(1, 7)))
print(" gdl  F_l (kN)")
mostrar_vector(F_2[indices(libres)], libres)
```

```
 gdl  F (kN)
   1      0.00
   2      0.00
   3     15.00
   4      0.00
   5     20.00
   6    -50.00
 gdl  F_l (kN)
   3     15.00
   5     20.00
   6    -50.00
```

El grado de libertad 3 (nodo 2, dirección x) es libre, de modo que la nueva carga aparece también en $\mathbf{F}_l$.

Ejercicio 8.3

```
barras_sin_1 = {2: barras[2], 3: barras[3]}
K_sin_1 = ensamblar_rigidez(nodos, barras_sin_1,
                            rigideces)
K_ll_sin_1 = particionar(K_sin_1, libres,
                         restringidos)[0]
print(f"Rango de K_ll: "
      f"{np.linalg.matrix_rank(K_ll_sin_1)} de 3")
```

```
Rango de K_ll: 2 de 3
```

Con rango 2 de 3, $\mathbf{K}_{ll}$ es singular: existe un movimiento de los nodos libres que no deforma ninguna barra, el mecanismo que encontramos en el ejercicio 6.3.

Ejercicio 8.4

```
orden = libres + restringidos
K_ordenada = K_global[np.ix_(indices(orden),
                             indices(orden))]
mostrar_matriz(K_ordenada / 1000, orden)
```

```
  gdl      3      5      6      1      2      4
    3   95.5  -28.8   38.4  -66.7    0.0  -38.4
    5  -28.8   57.6    0.0  -28.8  -38.4   38.4
    6   38.4    0.0  102.4  -38.4  -51.2  -51.2
    1  -66.7  -28.8  -38.4   95.5   38.4    0.0
    2    0.0  -38.4  -51.2   38.4   51.2    0.0
    4  -38.4   38.4  -51.2    0.0    0.0   51.2
```

El bloque superior izquierdo de 3×3 es $\mathbf{K}_{ll}$, y el superior derecho, $\mathbf{K}_{lr}$.

Capítulo 9

Ejercicio 9.1

La suma de fuerzas horizontales da $R_{1x} = -20\,\text{kN}$. Con los momentos respecto del nodo 1 obtenemos R_{2y}, y con la suma de fuerzas verticales, R_{1y}:

```
# Momentos respecto del nodo 1, positivos antihorarios:
# R2y · 6 − 20 · 4 − 50 · 3 = 0
R2y = (20 * 4 + 50 * 3) / 6
R1y = 50 - R2y
R1x = -20
print(f"R1x = {R1x:.2f} kN, R1y = {R1y:.2f} kN, "
      f"R2y = {R2y:.2f} kN")
```

```
R1x = -20.00 kN, R1y = 11.67 kN, R2y = 38.33 kN
```

Los valores coinciden con los del programa.

Ejercicio 9.2

```
apoyos_2 = {
    1: {"x": True, "y": True},
    2: {"x": True, "y": True},
}
libres_2, restringidos_2 = clasificar_gdl(nodos, apoyos_2)
K_ll_2, K_lr_2, K_rl_2, K_rr_2 = particionar(
    K_global, libres_2, restringidos_2)
U_r_2 = np.zeros(len(restringidos_2))
U_l_2 = resolver_desplazamientos(
    K_ll_2, K_lr_2, F[indices(libres_2)], U_r_2)
```

```
R_2 = calcular_reacciones(K_rl_2, K_rr_2, U_l_2, U_r_2,
                          F[indices(restringidos_2)])
print(" gdl    U (mm)")
mostrar_vector(U_l_2 * 1000, libres_2, decimales=3)
print(" gdl    R (kN)")
mostrar_vector(R_2, restringidos_2)
```

```
 gdl    U (mm)
   5     0.347
   6    -0.488
 gdl    R (kN)
   1      8.75
   2     11.67
   3    -28.75
   4     38.33
```

Con dos apoyos articulados hay cuatro reacciones y solo tres ecuaciones de equilibrio: la armadura es hiperestática externamente, y las reacciones horizontales no pueden obtenerse con la estática. Su suma sí: $8.75 - 28.75 = -20$ kN, que equilibra la carga horizontal. Las reacciones verticales no cambian, porque los momentos respecto del nodo 1 no dependen de las reacciones horizontales.

Ejercicio 9.3

```
U_l_doble = resolver_desplazamientos(K_ll, K_lr,
                                     2 * F_l, U_r)
np.allclose(U_l_doble, 2 * U_l)
```

```
True
```

La hipótesis de comportamiento elástico lineal con desplazamientos pequeños: $\mathbf{K}$ no depende de las cargas, de modo que los desplazamientos son proporcionales a ellas.

Ejercicio 9.4

```
geometria_b = calcular_geometria(nodos_b, barras_b)
rigideces_b = calcular_rigideces(barras_b, geometria_b)
K_b = ensamblar_rigidez(nodos_b, barras_b, rigideces_b)
F_b = ensamblar_cargas(nodos_b, cargas_b)
libres_b, restringidos_b = clasificar_gdl(nodos_b,
                                          apoyos_b)
K_ll_b, K_lr_b, _, _ = particionar(K_b, libres_b,
                                   restringidos_b)
U_l_b = resolver_desplazamientos(
    K_ll_b, K_lr_b, F_b[indices(libres_b)],
```

```
    np.zeros(len(restringidos_b)))
maximo = np.abs(U_l_b * 1000).max()
print(f"Mayor desplazamiento: {maximo:.3f} mm")
```

```
Mayor desplazamiento: 0.900 mm
```

Capítulo 10

Ejercicio 10.1

```
sigma_maxima = 0.0
for numero, fuerza in N.items():
    sigma = abs(fuerza) / barras[numero]["area"] / 1000
    sigma_maxima = max(sigma_maxima, sigma)
print(f"Mayor esfuerzo: {sigma_maxima:.2f} MPa")
fraccion = sigma_maxima / 250  # fy = 250 MPa
print(f"Fracción de la fluencia: {fraccion:.1%}")
```

```
Mayor esfuerzo: 23.96 MPa
Fracción de la fluencia: 9.6%
```

Un formato terminado en el signo % muestra el número como porcentaje; con `.1` antes del signo, con un decimal.

Ejercicio 10.2

```
apoyos_2 = {
    1: {"x": True, "y": True},
    2: {"x": True, "y": True},
}
libres_2, restringidos_2 = clasificar_gdl(nodos, apoyos_2)
K_ll_2, K_lr_2, _, _ = particionar(K_global, libres_2,
                                   restringidos_2)
U_2 = np.zeros(6)
U_2[indices(libres_2)] = resolver_desplazamientos(
    K_ll_2, K_lr_2, F[indices(libres_2)],
    np.zeros(len(restringidos_2)))
N_2 = fuerzas_axiales(barras, geometria, U_2)
filas = []
for numero in barras:
    filas.append([str(numero), N[numero], N_2[numero]])
mostrar_tabla(["Barra", "Rodillo", "Articulado"], filas)
```

```
   Barra    Rodillo Articulado
       1      28.75       0.00
       2     -47.92     -47.92
       3     -14.58     -14.58
```

Con dos apoyos articulados, la carga horizontal se reparte entre los apoyos (ejercicio 9.2), y el apoyo del nodo 2 impide que la barra 1 se alargue: la barra queda sin fuerza. Las barras 2 y 3 no cambian, porque el equilibrio del nodo 3 sigue determinando sus fuerzas.

Ejercicio 10.3

```
barras_dobles = {}
for numero, barra in barras.items():
    copia = dict(barra)
    copia["area"] = 2 * barra["area"]
    barras_dobles[numero] = copia

geometria_d = calcular_geometria(nodos, barras_dobles)
rigideces_d = calcular_rigideces(barras_dobles,
                                 geometria_d)
K_d = ensamblar_rigidez(nodos, barras_dobles,
                        rigideces_d)
K_ll_d, K_lr_d, _, _ = particionar(K_d, libres,
                                   restringidos)
U_d = np.zeros(6)
U_d[indices(libres)] = resolver_desplazamientos(
    K_ll_d, K_lr_d, F_l, U_r)
N_d = fuerzas_axiales(barras_dobles, geometria_d, U_d)
print(f"Desplazamientos a la mitad: "
      f"{np.allclose(U_d, U / 2)}")
filas = []
for numero in barras:
    filas.append([str(numero), N[numero], N_d[numero]])
mostrar_tabla(["Barra", "N (kN)", "N, 2A (kN)"], filas)
```

```
Desplazamientos a la mitad: True
      Barra     N (kN) N, 2A (kN)
          1      28.75      28.75
          2     -47.92     -47.92
          3     -14.58     -14.58
```

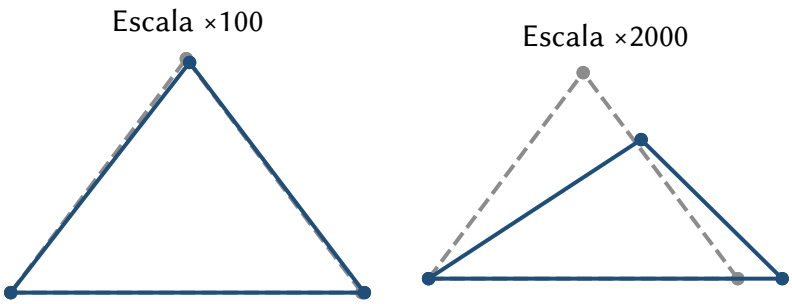
Los desplazamientos se reducen a la mitad, pero las fuerzas no cambian. En una armadura isostática, las fuerzas dependen solo del equilibrio y no de la rigidez de las barras; en una hiperestática, en cambio, cambiar la rigidez de una barra cambia el reparto de las fuerzas.

Capítulo 11

Ejercicio 11.1

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(4.5, 2.2))
```

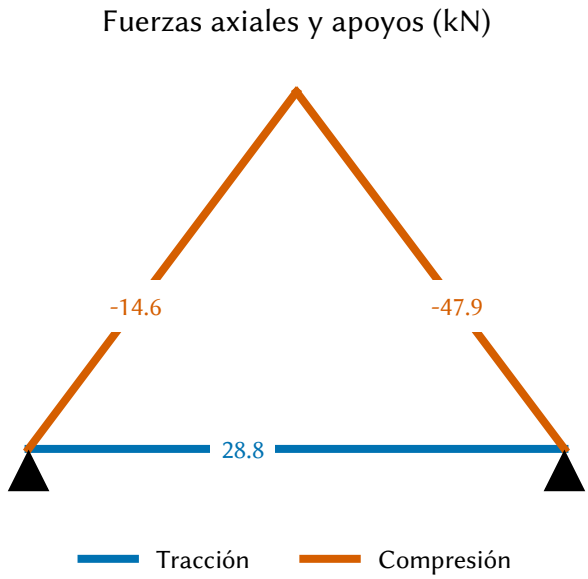
```
for ax, escala in ((ax1, 100), (ax2, 2000)):
    dibujar_armadura(ax, nodos, barras, estilo="--")
    deformados = nodos_desplazados(nodos, U, escala)
    dibujar_armadura(ax, deformados, barras,
                     color=PRINCIPAL)
    ax.set_title(f"Escala ×{escala}")
    ax.axis("off")
fig.tight_layout()
plt.show()
```



Con un factor de 100, la deformada casi se confunde con la armadura original y no permite ver el movimiento. Con 2000, el movimiento es evidente, aunque la armadura parece muy distorsionada. Un factor intermedio, como el automático, muestra el movimiento sin exagerarlo.

Ejercicio 11.2

```
fig, ax = plt.subplots()
dibujar_fuerzas(ax, nodos, barras, N)
for nodo in apoyos:
    punto = nodos[nodo]
    ax.plot(punto["x"], punto["y"] - 0.25, marker="^",
            markersize=14, color="black")
ax.set_ylim(-0.8, 4.5)
ax.set_title("Fuerzas axiales y apoyos (kN)")
plt.show()
```



Los triángulos se dibujan un poco por debajo de cada nodo, y `ax.set_ylim` amplía el rango vertical para que entren en el gráfico.

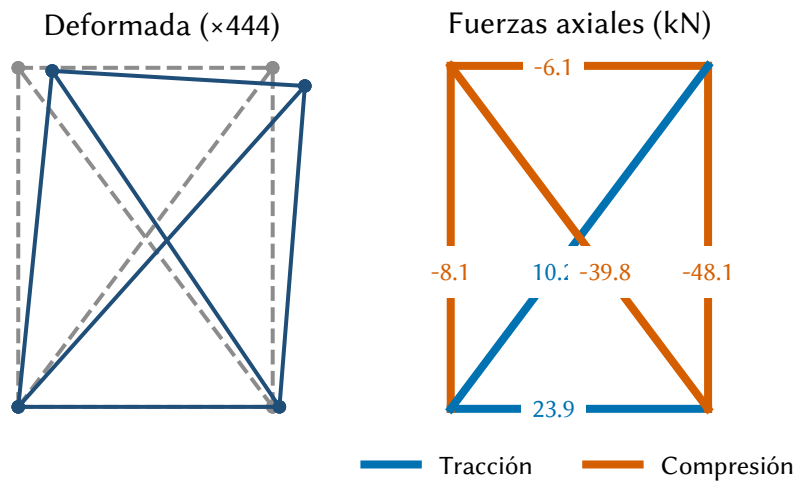
Ejercicio 11.3

Calculamos primero los resultados del ejemplo B con las funciones de los capítulos anteriores:

```
geometria_b = calcular_geometria(nodos_b, barras_b)
rigideces_b = calcular_rigideces(barras_b, geometria_b)
K_b = ensamblar_rigidez(nodos_b, barras_b, rigideces_b)
F_b = ensamblar_cargas(nodos_b, cargas_b)
libres_b, restringidos_b = clasificar_gdl(nodos_b,
                                          apoyos_b)
K_ll_b, K_lr_b, _, _ = particionar(K_b, libres_b,
                                   restringidos_b)
U_b = np.zeros(2 * len(nodos_b))
U_b[indices(libres_b)] = resolver_desplazamientos(
    K_ll_b, K_lr_b, F_b[indices(libres_b)],
    np.zeros(len(restringidos_b)))
N_b = fuerzas_axiales(barras_b, geometria_b, U_b)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(4.5, 2.8))
escala = escala_automatica(nodos_b, U_b)
dibujar_armadura(ax1, nodos_b, barras_b, estilo="--")
deformados = nodos_desplazados(nodos_b, U_b, escala)
dibujar_armadura(ax1, deformados, barras_b,
```

```
                  color=PRINCIPAL)
ax1.set_title(f"Deformada (×{escala:.0f})")
dibujar_fuerzas(ax2, nodos_b, barras_b, N_b)
ax2.set_title("Fuerzas axiales (kN)")
for ax in (ax1, ax2):
    ax.axis("off")
fig.tight_layout()
plt.show()
```

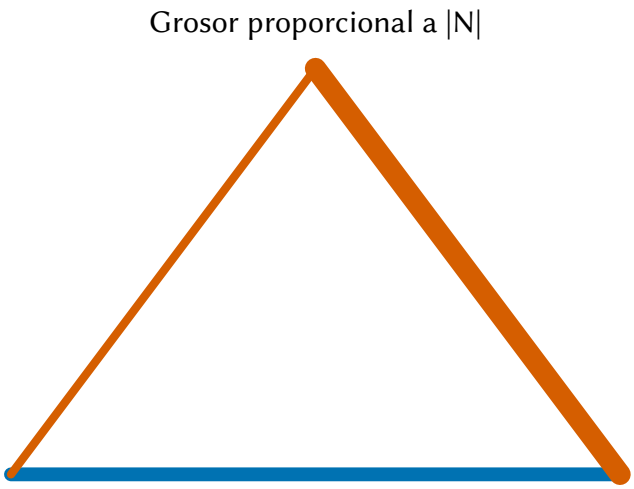


La deformada muestra el cuadro desplazándose hacia la derecha, empujado por la carga horizontal: la diagonal 6 se acorta y la 5 se alarga, como indican sus colores.

Ejercicio 11.4

```
fig, ax = plt.subplots()
mayor = 0.0
for fuerza in N.values():
    mayor = max(mayor, abs(fuerza))
for numero, barra in barras.items():
    i = nodos[barra["nodo_inicial"]]
    j = nodos[barra["nodo_final"]]
    if N[numero] > 0:
        color = TRACCION
    else:
        color = COMPRESION
    grosor = 1 + 7 * abs(N[numero]) / mayor
    ax.plot([i["x"], j["x"]], [i["y"], j["y"]],
            color=color, linewidth=grosor,
            solid_capstyle="round")
```

```
ax.set_aspect("equal")
ax.axis("off")
ax.set_title("Grosor proporcional a |N|")
plt.show()
```



El grosor va de 1 a 8 puntos, proporcional a $|N|$ respecto de la mayor fuerza. El argumento `solid_capstyle="round"` redondea los extremos de las líneas gruesas.

Capítulo 12

Ejercicio 12.1

El nodo 4 tiene los grados de libertad 7 y 8; la flecha es el desplazamiento vertical con el signo cambiado:

```
resultados_c = analizar_armadura(nodos_c, barras_c,
                                 apoyos_c, cargas_c)
iy = indices([8])[0]  # gdl 8: nodo 4, dirección y
flecha = -resultados_c["U"][iy] * 1000
print(f"Flecha en el centro: {flecha:.2f} mm")
print(f"Luz / flecha: {18000 / flecha:.0f}")
```

```
Flecha en el centro: 6.74 mm
Luz / flecha: 2672
```

Una relación luz/flecha del orden de 2700 es muy holgada: los límites de servicio habituales están entre 300 y 1000.

Ejercicio 12.2

Usamos la función `barra` del ejemplo C para escribir las barras:

```
nodos_cubierta = {
    1: {"x": 0.0, "y": 0.0},
    2: {"x": 4.0, "y": 0.0},
    3: {"x": 8.0, "y": 0.0},
    4: {"x": 4.0, "y": 1.5},
}
barras_cubierta = {
    1: barra(1, 2, 0.0015), 2: barra(2, 3, 0.0015),
    3: barra(1, 4, 0.0015), 4: barra(4, 3, 0.0015),
    5: barra(2, 4, 0.0010),
}
apoyos_cubierta = {
    1: {"x": True, "y": True},
    3: {"x": False, "y": True},
}
cargas_cubierta = {4: {"x": 0.0, "y": -30.0}}

resultados = analizar_armadura(
    nodos_cubierta, barras_cubierta, apoyos_cubierta,
    cargas_cubierta)
verificar_resultados(nodos_cubierta, barras_cubierta,
                     resultados)
mostrar_resultados(nodos_cubierta, barras_cubierta,
                   resultados)
```

```
Equilibrio global: correcto
Equilibrio nodal: correcto
Desplazamientos de los nodos (mm)
       Nodo         ux         uy
          1      0.000      0.000
          2      0.533     -3.155
          3      1.067      0.000
          4      0.533     -3.155

Reacciones (kN)
       Nodo  Dirección          R
          1          x       0.00
          1          y      15.00
          3          y      15.00

Fuerzas axiales
      Barra     N (kN)    σ (MPa)     Estado
          1      40.00      26.67   tracción
          2      40.00      26.67   tracción
          3     -42.72     -28.48 compresión
```

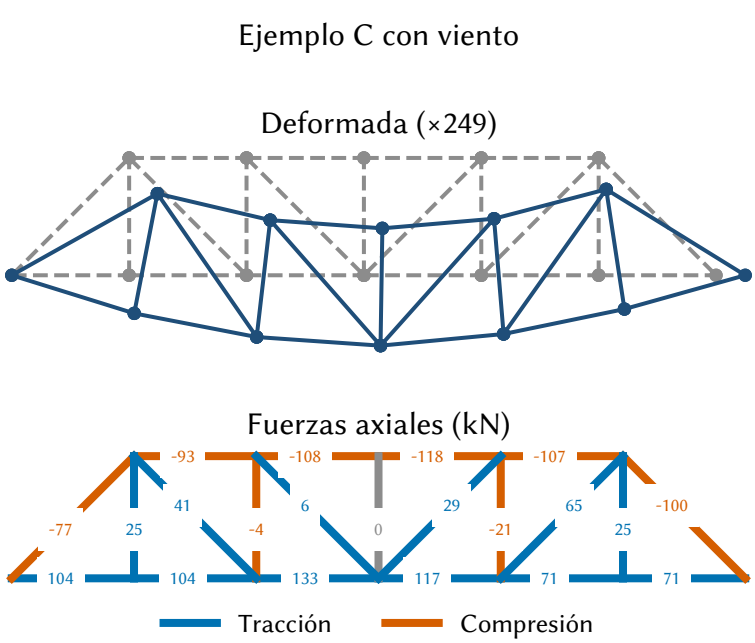
```
           4     -42.72     -28.48 compresión
           5       0.00       0.00 sin fuerza
```

En el nodo 2 concurren las barras 1 y 2, horizontales, y la barra 5, vertical. Como el nodo 2 no tiene carga, el equilibrio vertical exige que la barra 5 no tenga fuerza.

Ejercicio 12.3

```
# Viento: 10 kN horizontales en cada nodo superior
cargas_viento = {}
for nodo in range(2, 7):
    cargas_viento[nodo] = {"x": 0.0, "y": -25.0}
for nodo in range(8, 13):
    cargas_viento[nodo] = {"x": 10.0, "y": 0.0}

resultados_v = analizar_armadura(nodos_c, barras_c,
                                 apoyos_c, cargas_viento)
verificar_resultados(nodos_c, barras_c, resultados_v)
graficar_resultados(nodos_c, barras_c, resultados_v,
                    "Ejemplo C con viento", decimales=0,
                    tamano=7)
```



El viento empuja el puente hacia la derecha: las fuerzas dejan de ser simétricas y la reacción horizontal del apoyo articulado equilibra los 50 kN de viento.

Ejercicio 12.4

```
# Sin las diagonales 18 y 19, el tramo izquierdo
# queda formado por rectángulos sin arriostrar
barras_sin_diagonales = {}
for numero, datos in barras_c.items():
    if numero not in (18, 19):
        barras_sin_diagonales[numero] = datos

try:
    analizar_armadura(nodos_c, barras_sin_diagonales,
                      apoyos_c, cargas_c)
except ValueError as error:
    print(f"No se puede analizar. {error}")
```

```
No se puede analizar. La armadura es inestable: K_ll es
↪  singular.
```

Sin las diagonales 18 y 19, los paneles segundo y tercero son rectángulos formados por cordones y montantes, sin ninguna barra que impida que se deformen como un paralelogramo: la armadura es un mecanismo. Además, $g = 19 + 3 - 24 = -2$.

Bibliografía

Kassimali, Aslam (2012). *Matrix Analysis of Structures*. 2.ª ed. Cengage Learning.

McGuire, William, Richard H. Gallagher y Ronald D. Ziemian (2000). *Matrix Structural Analysis*. 2.ª ed. John Wiley & Sons.

McKenna, Frank (2011). «OpenSees: A Framework for Earthquake Engineering Simulation». En: *Computing in Science & Engineering* 13.4, págs. 58-66.

Índice alfabético

A

alargamiento, 81
`analizar_armadura`, 101
argumentos, 41
argumentos con nombre, 92
armadura Pratt, 107
arreglo, 42
asignación, 12

B

booleanos, 24
bucle, 32

C

calculadora
    notebook final, 111
comparaciones, 48
compresión, 81
cosenos directores, 9

D

`def`, 40
deformada, 94
desplazamientos de los nodos, 76
diagrama de fuerzas axiales, 95
diccionario, 23
diccionarios anidados, 24
diferencia relativa, 140
división entera, 104
documentación, 41

E

ejemplo A, 6
ejemplo B, 6
ejemplo C, 6
ensamblaje, 55
entorno virtual, 127
equilibrio
    de los nodos, 85
    global, 77
errores de Python, 121
esfuerzo, 83
estabilidad, 52
excepción, 75

F

f-strings, 14
factor de escala, 91
`for`, 32
fuerza axial, 81
función, 40

G

geometría de las barras, 33
Google Colab, 10
grado de indeterminación, 52
grados de libertad, 5, 39
    libres y restringidos, 50
    numeración de los, 49

H

hiperestática, 4

I

`if`, 48
índice, 23
`indices`, 51
índices de NumPy, 51
inestabilidad, 75
isostática, 4
`items`, 33

L

lista, 22

M

`math`, 13
Matplotlib, 26
matriz de rigidez, 4
    de una barra, 39
matriz de rigidez global, 55
matriz de rigidez local, 39
matriz de transformación, 40
método de las secciones, 107
método de los nudos, 4
método de rigidez, 4
    pasos del, 5
modelo de datos, 21
módulo, 13, 129
movimiento de cuerpo rígido, 44

N

notación científica, 13
notebook, 10
`np.ix_`, 59
`np.linalg.solve`, 74
`np.zeros`, 56
numeración desde 1, 49
NumPy, 42

P

parámetros, 40
partición del sistema, 68
precisión de la máquina, 140
`print`, 14

R

`raise`, 75
rango, 44
reacciones, 76
rebanadas, 67
residuo, 74
resto, 104
`return`, 40
rigidez axial, 39

S

simetría, 44

T

tolerancia, 74
tracción, 81
traza, 121
`try`, 75

V

variable, 12
vector de cargas, 66
vector de correspondencia, 48