Python en Ámbitos Científicos

Facundo Batista & Manuel Carlevaro

24 de septiembre de 2021

VERSIÓN PRELIMINAR

Título: Python en Ámbitos Científicos
Autores: Facundo Batista & Manuel Carlevaro

ISBN-13 (versión electrónica): ???-?-????-????-?

Primera Edición (versión preliminar)
Escrito con XeLaTeX.

Lugar: Olivos y La Plata, Buenos Aires, Argentina
Año: 2021
Web: https://github.com/facundobatista/libro-pyciencia

10 9 8 7 6 5 4 3 2 1

VERSIÓN PRELIMINAR

Prefacio

Cuando la Comunidad Científica notó que las computadoras podían facilitar su trabajo, sus integrantes se convirtieron posiblemente en los adoptantes tempranos más entusiastas de la tecnología. Aunque el uso y la potencia de las computadoras crece continuamente en el ámbito científico, poco se ha avanzado en la formación y desarrollo de habilidades en Desarrollo de Software.

El cálculo científico requiere realizar combinaciones de múltiples tareas de diversa clase. Por ejemplo, es necesario registrar automáticamente datos de un experimento y visualizarlos, realizar cálculos numéricos o simbólicos, ordenar, clasificar, simular, etc. Muchas veces podemos utilizar paquetes de software que realizan esas tareas por nosotros, pero muchas otras ocurre que nadie ha implementado un determinado cómputo en la forma que necesitamos, o simplemente queremos probar ideas nuevas. Cualquiera sea la naturaleza de nuestra actividad en la ciencia o en la tecnología, no resulta infrecuente la necesidad de interactuar con computadoras a través de programas propios.

El uso de Python en aplicaciones científicas ha aumentado sostenidamente en los últimos años, sin embargo es difícil encontrar libros o manuales en castellano de Python que no estén pensados para programadores.

Este libro nace con la idea primaria de acercar Python al mundo científico, en un libro pensado para científiques, a partir de nuestra experiencia en el dictado del curso "Herramientas Computacionales para Científicos"que ofrecemos en la Universidad Nacional de La Plata y la Universidad Tecnológica Nacional, desde 2007. De la misma manera, la elección del castellano como idioma de escritura es un factor crítico, porque aunque sabemos que el inglés es una herramienta fundamental tanto para programar como para hacer ciencia, estamos convencidos que no debería ser una barrera de entrada.

Más allá de las secciones básicas de un libro (índice, bibliografía, etc.), el libro tiene dos grandes partes. La primera habla de Python, algunas bibliotecas importantes y otros temas que son fundamentales. En la segunda, mostramos cómo abordar temas científicos básicos utilizando Python, de forma teórica y práctica.

En ambos casos este libro esquiva la pretensión de ser una referencia absoluta, sino que tiene el propósito de allanar el camino de les científiques para dar los primeros pasos en el lenguaje y solucionar los problemas básicos (pero no por eso menos importantes) de la ciencia y la tecnología.

Olivos y La Plata, Buenos Aires, Argentina,

Facundo Batista & Manuel Carlevaro

Índice general

Parte I
Herramientas fundamentales

Los capítulos siguientes desarrollan un conjunto de conocimientos y técnicas fundamentales que serán utilizados en la Parte II al abordar temas de aplicaciones específicas. Se sugiere, a los lectores que no tienen experiencia previa en el uso de Python, un recorrido secuencial a través de los capítulos que componen esta Parte.

Parte II
Temas específicos

En esta Parte desarrollaremos capítulos que abordan temas específicos de aplicación de herramientas de Python. Cada capítulo está autocontenido, de forma que el lector o lectora puede acceder directamente al tema de interés sin realizar un recorrido secuencial de los capítulos, aunque serán utilizados conceptos y herramientas tratados en la Parte I.

1 | Ecuaciones algebraicas

En este capítulo abordaremos la resolución de ecuaciones algebraicas. Es necesario distinguir entre ecuaciones univariadas o multivariadas (es decir, ecuaciones que contienen una variable desconocida o muchas), y también si las ecuaciones son lineales o no lineales. Esta clasificación es necesaria pues cada tipo de ecuación requiere la utilización de técnicas específicas de solución.

Comenzaremos con sistemas de ecuaciones lineales cuya utilidad es central en muchos campos científicos, y pueden resolverse mediante la aplicación del álgebra lineal. Por otra parte, las ecuaciones no lineales son más complicadas de resolver y usualmente demandan métodos que son computacionalmente intensivos.

Módulo	Versión
Matplotlib	3.3.4
NumPy	1.19.5
SciPy	1.6.0
Sympy	1.7.1

Código disponible

1.1. Sistema de ecuaciones lineales

Una de las aplicaciones más importantes del álgebra lineal es la resolución de sistemas de ecuaciones lineales. Tanto NumPy como SciPy poseen submódulos con rutinas de álgebra lineal. Sin embargo `scipy.linalg` contiene todas las funciones de `numpy.linalg` y algunas más avanzadas no disponibles en NumPy. Una diferencia importante entre ambos módulos es que `scipy.linalg` está compilado con soporte de las bibliotecas ATLAS LAPACK y BLAS, que son rutinas escritas en Fortran altamente optimizadas para realizar operaciones con vectores y matrices (BLAS) y para resolver sistemas de ecuaciones lineales (LAPACK), mientras que esto es opcional para NumPy. De este modo, la versión de linalg de scipy puede ser más rápida que la de NumPy, dependiendo de cómo se ha instalado Numpy. Por este motivo recomendamos el uso de `scipy.linalg`.

La forma general de un sistema de ecuaciones lineales es:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n &= b_m \end{aligned}$$

Este sistema representa un conjunto de m ecuaciones con n incógnitas x_j $(1 \leq j \leq n)$, en las que a_{ij} $(1 \leq i \leq m)$ y b_j son constantes o parámetros conocidos. Naturalmente, al trabajar

con sistemas de ecuaciones lineales resulta muy práctico hacerlo en notación matricial:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \tag{1.1}$$

o, en forma compacta, $\mathbb{A}\boldsymbol{x} = \boldsymbol{b}$. en la que $\mathbb{A}$ es una matriz $m \times n$, $\boldsymbol{b}$ es un vector de m componentes (o matriz $m \times 1$) y $\boldsymbol{x}$ es el vector incógnita o solución de n componentes (o matriz $n \times 1$).

El sistema representado por la ecuación (1.1) (o su forma compacta) puede tener solución o no, y en caso de que exista dicha solución la misma puede no ser única, y esto depende de las propiedades de la matriz $\mathbb{A}$.

Cuando tenemos menos ecuaciones que incógnitas, esto es, $n < m$, el sistema está subdeterminado y por lo tanto no puede determinarse una solución única. Por el contrario, cuando existen más ecuaciones que incógnitas ($n > m$) el sistema está sobredeterminado, y en este caso, por lo general, se producen restricciones que no permiten hallar una solución.

Los sistemas cuadrados, esto es, cuando $n = m$, son de importancia debido a que puede existir una solución única, y para ello la matriz $\mathbb{A}$ debe ser *no singular* lo que implica que existe la inversa de $\mathbb{A}$, y la solución puede escribirse como:

$$x = \mathbb{A}^{-1} \cdot \boldsymbol{b} \tag{1.2}$$

Si la matriz $\mathbb{A}$ es singular, esto es cuando el rango de la matriz es menor que n, $\text{rg}(\mathbb{A}) < n$, o equivalentemente el determinante es nulo ($\det(\mathbb{A}) = 0$), la ecuación (1.1) puede no tener solución o tener infinitas, dependiendo del vector $\boldsymbol{b}$. Cuando $\text{rg}(\mathbb{A}) < n$ hay filas o columnas que son combinaciones lineales de otras filas o columnas, y por lo tanto corresponden a ecuaciones que no representan una nueva relación entre las incógnitas, y por lo tanto el sistema está subdeterminado. De este modo, calcular el rango de la matriz que define el sistema de ecuaciones lineales es un método útil para saber si la matriz es singular o no, y en consecuencia saber de antemano si existe la solución del sistema.

Al abordar el problema numéricamente pueden presentarse dos cuestiones adicionales que complican hallar la solución. Por un lado, aunque algunas ecuaciones no sean exactamente combinaciones lineales de otras, pueden estar tan cerca de ser linealmente dependientes que los errores de rendondeo en el cálculo con representación de punto flotante las hacen linealmente dependientes en alguna etapa del cálculo de la solución, y en este caso la solución encontrada no será tal.

Por otro lado, particularmente cuando el número n es muy grande, los errores de rendondeo acumulados pueden arruinar completamente la solución, aún cuando el proceso algorítmico es correcto. Este hecho se acentúa cuando $\mathbb{A}$ está cerca de ser singular.

1.1.1. Condicionamiento

Aún cuando $\text{rg}(\mathbb{A}) = n$ y por consiguiente la solución existe, puede no ser posible obtener dicha solución en forma precisa. En efecto, el número de condición de la matriz, $\text{cond}(\mathbb{A})$ determina si la matriz está bien o mal condicionada. Cuando $\text{cond}(\mathbb{A}) \approx 1$, la matriz está *bien condicionada*, pero cuando $\text{cond}(\mathbb{A}) \gg 1$ se dice que está *mal condicionada*. En este último caso, la solución del sistema de ecuaciones puede tener errores de gran magnitud. Una forma de interpretar cómo afecta el número de condición a los errores consiste en realizar un análisis del error. Supongamos que tenemos un sistema de ecuaciones lineales de la forma $\mathbb{A}\boldsymbol{x} = \boldsymbol{b}$ donde $\boldsymbol{x}$ es el vector solución. A una pequeña variación en $\boldsymbol{b}$, $\delta\boldsymbol{b}$, le corresponderá un cambio en la solución, $\delta\boldsymbol{x}$ dado por $\mathbb{A}(\boldsymbol{x} + \delta\boldsymbol{x}) = \boldsymbol{b} + \delta\boldsymbol{b}$, y dado que el sistema es lineal, resulta que $\mathbb{A}\delta\boldsymbol{x} = \delta\boldsymbol{b}$.

Entonces, podemos determinar cuán grande es el cambio relativo en $\boldsymbol{x}$ comparado con el cambio relativo en $\boldsymbol{b}$ haciendo uso de los cocientes de las normas de estos vectores. Específicamente, comparamos $\|\delta\boldsymbol{x}\|/\|\boldsymbol{x}\|$ y $\|\delta\boldsymbol{b}\|/\|\boldsymbol{b}\|$, donde $\|\boldsymbol{x}\|$ denota la norma del vector $\boldsymbol{x}$. Usando la relación $\|\mathbb{A}\boldsymbol{x}\| \leq \|\mathbb{A}\|\|\boldsymbol{x}\|$, podemos escribir

$$\frac{\|\delta\boldsymbol{x}\|}{\|\boldsymbol{x}\|} = \frac{\|\mathbb{A}^{-1}\delta\boldsymbol{b}\|}{\|\boldsymbol{x}\|} \leq \frac{\|\mathbb{A}^{-1}\|\|\delta\boldsymbol{b}\|}{\|\boldsymbol{x}\|} = \frac{\|\mathbb{A}^{-1}\|\|\boldsymbol{b}\|}{\|\boldsymbol{x}\|}\frac{\|\delta\boldsymbol{b}\|}{\|\boldsymbol{b}\|} \leq \|\mathbb{A}^{-1}\|\|\mathbb{A}\|\frac{\|\delta\boldsymbol{b}\|}{\|\boldsymbol{b}\|}$$

Dado que la definición del número de condición es $\text{cond}(\mathbb{A}) = \|\mathbb{A}\|\|\mathbb{A}^{-1}\|$, la expresión anterior da una cota superior al error relativo en la solución dado el error relativo en $\boldsymbol{b}$. De aquí se infiere que si un sistema lineal de ecuaciones está definido por una matriz $\mathbb{A}$ que está mal condicionada, pequeñas variaciones del vector $\boldsymbol{b}$ pueden originar grandes errores en el vector solución $\boldsymbol{x}$. Por esto es que al resolver estos sistemas es necesario calcular el número de condición para estimar la precisión de las soluciones obtenida, particularmente cuando se usan números con representación de punto flotante, que tal como vimos en el Capítulo **??** son solo aproximaciones a los números reales.

En el caso en que estemos tratando simbólicamente con una matriz, el rango, el número de condición y la norma se pueden obtener con los métodos `rank`, `condition_number` y `norm` del objeto `Matrix` de SimPy, mientras que si solo trabajamos en forma numérica, las mismas magnitudes se obtienen de las funciones del submódulo `linalg` de NumPy: `matrix_rank`, y `cond` y `norm` (SciPy solo tiene implementada la función `norm` en el submódulo `linalg`).

Como ejemplo veremos cómo calcular estas cantidades en una matriz de Hilbert de 2×2, que se conoce como ejemplo de matriz mal condicionada. Los elementos de esta matriz son $H_{ij} = 1/(i + j - 1)$. Primero importamos los módulos que vamos a usar:

CELL 01

```
import matplotlib.pyplot as plt
import numpy as np
import scipy.linalg as la
import sympy as sy
from sympy import Matrix, Rational, N
```

Luego definimos la matriz de Hilbert, utilizando el método `Rational` de SimPy para manter la expresión de los elementos de la matriz como fracciones (la llamamos `H_s` para destacar que es un objeto `Matrix` de SimPy):

CELL 02

```
# Matriz de Hilbert 2x2
H_s = Matrix([[1, Rational(1,2)],
              [Rational(1,2), Rational(1,3)]])
H_s
```

$$\begin{bmatrix} 1 & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{3} \end{bmatrix}$$

A continuación obtenemos el rango, el número de condición exacto y aproximado, y la norma de H

CELL 03

```
H_s.rank()
```

```
2
```

CELL 04

```
H_s.condition_number()
```

$$\frac{\sqrt{\frac{2\sqrt{13}}{9}+\frac{29}{36}}}{\sqrt{\frac{29}{36}-\frac{2\sqrt{13}}{9}}}$$

CELL 05

```
N(_)
```

$$19{,}281470067904$$

CELL 06

```
H_s.norm()
```

$$\frac{\sqrt{58}}{6}$$

Ahora pasamos a la representación en punto flotante de H, utilizando las rutinas de NumPy y SciPy para definirla (SciPy permite definir esta matriz indicando el rango de la misma) y para calcular su rango, número de condición y norma:

CELL 07

```
# Matriz de Hilbert 2x2 (punto flotante)
H = la.hilbert(2)
H
```

```
array([[1.        , 0.5       ],
       [0.5       , 0.33333333]])
```

CELL 08

```
np.linalg.matrix_rank(H)
```

```
2
```

CELL 09

```
np.linalg.cond(H)
```

```
19.28147006790397
```

CELL 10

```
la.norm(H)
```

```
1.2692955176439846
```

CELL 11

```
np.linalg.norm(H)
```

```
1.2692955176439846
```

1.1.2. Descomposición LU

La forma directa de resolver un sistema de ecuaciones lineales consiste en obtener la matriz inversa de $\mathbb{A}$, y multiplicarla por el vector $\boldsymbol{b}$. Sin embargo esta no es la forma más eficiente de hacerlo, particularmente cuando utilizamos el sistema caracterizado por $\mathbb{A}$ para resolver múltiples sistemas con distintos vectores $\boldsymbol{b}$: $\mathbb{A}\boldsymbol{x}_i = \boldsymbol{b}_i$.

La descomposición LU consiste en factorizar la matriz $\mathbb{A}$ de la siguiente forma:

$$\mathbb{L} \cdot \mathbb{U} = \mathbb{A} \tag{1.3}$$

donde $\mathbb{L}$ es una matriz diagonal inferior (es decir, tiene elementos no nulos en la diagonal y debajo de ella) y $\mathbb{U}$ es una matriz diagonal superior (con elementos no nulos en la diagonal y arriba de ella). Para el caso de una matriz $\mathbb{A}$ de 4×4, esto se vería:

$$\begin{bmatrix} L_{11} & 0 & 0 & 0 \\ L_{21} & L_{22} & 0 & 0 \\ L_{31} & L_{32} & L_{33} & 0 \\ L_{41} & L_{42} & L_{43} & L_{44} \end{bmatrix} \cdot \begin{bmatrix} U_{11} & U_{12} & U_{13} & U_{14} \\ 0 & U_{22} & U_{23} & U_{24} \\ 0 & 0 & U_{33} & U_{34} \\ 0 & 0 & 0 & U_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{m1} & a_{m2} & a_{43} & a_{44} \end{bmatrix} \tag{1.4}$$

Utilizando esta factorización podemos resolver el sistema lineal

$$\mathbb{A} \cdot \boldsymbol{x} = (\mathbb{L} \cdot \mathbb{U}) \cdot \boldsymbol{x} = \boldsymbol{b} \tag{1.5}$$

resolviendo primero para un vector $\boldsymbol{y}$ tal que

$$\mathbb{L} \cdot \boldsymbol{y} = \boldsymbol{b} \tag{1.6}$$

y luego resolvemos

$$\mathbb{U} \cdot \boldsymbol{x} = \boldsymbol{y} \tag{1.7}$$

Este procedimiento de dividir el problema dado por la ecuación (1.5) en dos ecuaciones sucesivas resulta eficiente debido a que obtener la solución de sistemas triangulares es trivial: la ecuación (1.6) se resuelve con sustitución hacia adelante mientras que la (1.7) se resuelve con sustitución hacia atrás. De este modo, en el caso de resolver nuestro sistema original para múltiples términos independientes $\boldsymbol{b}_i$, solo una vez hay que obtener la factorización LU y luego simplemente se resuelve para cada caso mediante las sustituciones de las ecuaciones (1.6) y (1.7).

Tanto Sympy como SciPy poseen rutinas que realizan la factorización LU, veamos ejemplos de aplicación de esos métodos. En las celdas siguientes utilizamos el método `LUdecomposition()` aplicado a la matriz de Hilbert:

Cell 12

```
L_s, U_s, P_s = H_s.LUdecomposition()
L_s
```

$$\begin{bmatrix} 1 & 0 \\ \frac{1}{2} & 1 \end{bmatrix}$$

Cell 13

```
U_s
```

$$\begin{bmatrix} 1 & \frac{1}{2} \\ 0 & \frac{1}{12} \end{bmatrix}$$

CELL 14

```
L_s * U_s
```

$$\begin{bmatrix} 1 & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{3} \end{bmatrix}$$

Podemos ver que la matriz `L` posee unos en su diagonal, lo que indica que Sympy realiza una descomposición de Doolittle. En general es posible realizar la factorización de muchas formas, en el caso en que los unos forman la diagonal de la matriz `U`, por ejemplo, estaríamos ante una descomposición de Crout. Tanto Sympy como SciPy tienen implmentada la descomposición de Doolittle.

Vemos además que el método devuelve también la matriz `P`, que está compuesta por filas con pares de índices de intercambio. Esto resulta de intercambiar el orden de las filas de modo eliminar el mal condicionamiento que puede ocurrir cuando un elemento a_{ij} de la matriz es mucho menor que uno. Las estrategias de pivoteo exceden el alcance de este capítulo, pero están implementadas en estos métodos para ofrecer soluciones robustas.

Por último, en la celda 14 podemos ver que el producto de `L` y `U` reconstruye la matriz de Hilbert, tal como establece su factorización.

En el caso de la factorización numérica con SciPy, la función `lu` del submódulo `linalg` devuelve las tres matrices anteriores pero en distinto orden, tal como se puede ver en las celdas siguientes:

CELL 15

```
P, L, U = la.lu(H)
L
```

```
array([[1. , 0. ],
       [0.5, 1. ]])
```

CELL 16

```
U
```

```
array([[1.        , 0.5       ],
       [0.        , 0.08333333]])
```

CELL 17

```
np.allclose(H - P @ L @ U, np.zeros((2,2)))
```

```
True
```

En la celda 17 comparamos elemento por elemento los valores de `H - P @ L @ U` con un array 2×2 con elementos nulos, lo que nos devuelve `True` comprobando de este modo que la descomposición LU recupera la matriz de Hilbert original al realizar el producto de las matrices que lo factorizan. Debemos notar en la celda 17 que denotamos el producto de matrices con el operador `@` que es un alias de la función `np.matmul` sobre arrays.

Utilizando la descomposición LU mostraremos ahora un ejemplo de resolución de un sistema lineal de ecuaciones en forma analítica con Sympy y numérica con SciPy, y evaluaremos el efecto del condicionamiento en la precisión de la solución numérica. Por lo general, la solución analítica solo es útil para matrices muy pequeñas, ya que al aumentar el rango de la matriz las expresiones pueden resultar muy difíciles de leer, y finalmente se evalúan numéricamente.

Nuestro sistema de ejemplo será la matriz:

$$\mathbb{A}(p) = \begin{bmatrix} 1 & \frac{1}{p+1} \\ \frac{1}{p+1} & \frac{1}{p+2} \end{bmatrix}$$

Podemos ver que cuando $p = 1$, $\mathbb{A}$ se convierte en la matriz de Hilbert 2×2. En la siguiente celda definimos esta matriz en forma simbólica con Sympy:

CELL 18

```
# Especificación del problema simbólico
p = sy.symbols("p", positive=True)
A_s = Matrix([[1, 1/(p + 1)], [1/(p + 1), 1/(p + 2)]])
A_cond = A_s.condition_number() # Número de condición
A_s
```

$$\begin{bmatrix} 1 & \frac{1}{p+1} \\ \frac{1}{p+1} & \frac{1}{p+2} \end{bmatrix}$$

y también calculamos el número de condición `A_cond`. Definimos a continuación el vector de términos independientes y obtenemos la solución del sistema, que obviamente es función del parámetro p:

CELL 19

```
# Término independiente y solución analítica
b_s = Matrix([5, 3])
x_s = A_s.solve(b_s)
x_s
```

$$\begin{bmatrix} \frac{2p^2+p-1}{p^2+p-1} \\ \frac{3p^3+7p^2-4}{p^2+p-1} \end{bmatrix}$$

A continuación definimos el problema numérico haciendo uso de funciones **`lambda`** de modo de poder resolver el problema especificando cada vez el valor del parámetro `p`.

CELL 20

```
# Especificación del problema numérico
A_n = lambda p: np.array([[1, 1/(p + 1)], [1/(p + 1), 1/(p + 2)]])
b_n = np.array([5, 3])
x_n = lambda p: np.linalg.solve(A_n(p), b_n)
```

Para el caso en que $p = 1$ podemos obtener las soluciones del sistema determinado por la matriz de Hilbert tanto en forma exacta como numérica:

CELL 21

```
# Solución exacta para p = 1 (matriz de Hilbert)
x_s.subs(p, 1)
```

$$\begin{bmatrix} 2 \\ 6 \end{bmatrix}$$

CELL 22

```
# Solución numérica para p = 1 (matriz de Hilbert)
x_n(1)
```

```
array([2., 6.])
```

También podemos encontrar la solución haciendo uso de la descomposición LU, lo que sería muy eficiente si quisiéramos resolver el sistema para múltiples términos independientes. En primer lugar realizamos la factorización de la matriz `A` para $p = 1$, luego obtenemos la solución numérica `x_lu` y finalmente verificamos que al reemplazar la solución obtenida en el sistema se verifica cada una de las ecuaciones (es decir, que $\mathbb{A}(1)\boldsymbol{x} - \boldsymbol{b} = 0$):

CELL 23

```
# Solución numérica usando la factorización LU
lu, piv = la.lu_factor(A_n(1))
x_lu = la.lu_solve((lu, piv), b_n)
np.allclose(A_n(1) @ x_lu - b_n, np.zeros(2))
```

```
True
```

Para finalizar exploraremos el efecto que tiene el condicionamiento de la matriz $\mathbb{A}(p)$ en la precisión de los resultados obtenidos. Para ello notemos que el número de condición diverge para $p = (\sqrt{5} - 1)/2$ (dejamos la verificación de esto para la lectora o el lector). En consecuencia, construiremos un array de valores de `p` para los que resolveremos el sistema tanto en forma exacta como numérica, y evaluaremos el error relativo de la solución numérica.

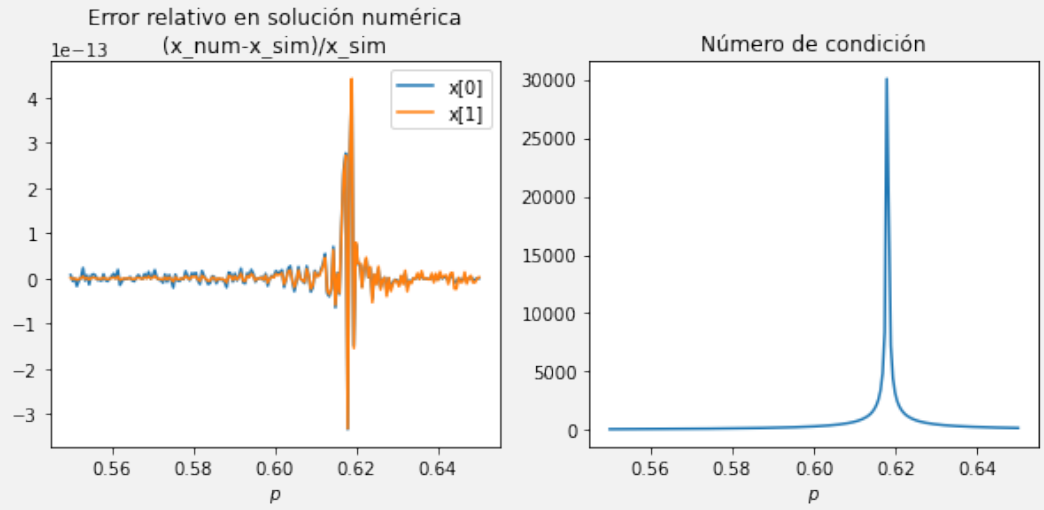
En la celda 24 graficamos a la izquierda este error relativo para cada una de las componentes del vector solución, mientras que el el panel de la derecha mostramos el número de condición. Podemos apreciar que donde diverge el número de condición ($p \approx 0{,}618$), el error relativo de la solución numérica aumenta significativamente. Pese a que el método de solución es muy robusto para este caso, dado que el error relativo es del orden de 4×10^{-13}, este valor es tres órdenes de magnitud mayor que el épsilon de la máquina donde ejecutamos el ejemplo (ver Sección **??**), como muestra la celda 25, con lo cual es un error apreciable que puede magnificarse al resolver sistemas grandes de ecuaciones (o al resolver sucesivamente sistemas lineales de ecuaciones en un algoritmo).

CELL 24

```
# Gráfico de la diferencia entre las soluciones analítica (exacta) y numérica
fig, ax = plt.subplots(1, 2, figsize=(10, 4))
p_valores = np.linspace(0.55, 0.65, 200)
for n in range(2):
    x_sim = np.array([x_s[n].subs(p, pp).evalf() for pp in p_valores])
    x_num = np.array([x_n(pp)[n] for pp in p_valores])
    ax[0].plot(p_valores, (x_num - x_sim)/x_sim, label=f'x[{n}]')

ax[0].set_title("Error relativo en solución numérica\n(x_num-x_sim)/x_sim")
ax[0].set_xlabel(r'$p$')
ax[0].legend()

ax[1].plot(p_valores, [A_cond.subs(p, pp).evalf() for pp in p_valores])
ax[1].set_title("Número de condición")
ax[1].set_xlabel(r'$p$')
plt.show()
```



CELL 25

```
import sys
print(sys.float_info.epsilon)
```

```
2.220446049250313e-16
```

1.2. Problema de autovalores

Así como los sistemas de ecuaciones lineales suelen originarse a partir de problemas en estado estacionario, los problemas de autovalores son de gran importancia en la representación de problemas dinámicos, por lo que aparecen frecuentemente en muchas áreas de la ciencia y de la ingeniería.

Si $\mathbb{A}$ es una matriz cuadrada $n \times n$, el problema de autovalores de esta matriz se define como

$$\mathbb{A} \cdot \boldsymbol{x} = \lambda \boldsymbol{x} \tag{1.8}$$

donde λ un escalar denominado *autovalor* y $\boldsymbol{x}$ es el *autovector* correspondiente, ambos desconocidos. Obviamente, cualquier múltiplo de un autovector es también un autovector, por lo que no se consideran autovectores distintos (el autovector cero no se considera un autovector). Resolver el problema de autovalores (1.8) consiste en determinar este par autovalor/autovector.

Para abordar el problema, podemos introducir la matriz identidad $\mathbb{I}$ (matriz cuyos elementos diagonales son unos y el resto son ceros), y reescribir la ecuación (1.8) como

$$(\mathbb{A} - \lambda\mathbb{I}) \cdot \boldsymbol{x} = \mathbf{0} \tag{1.9}$$

Esta expresión nos dice que la matriz $\mathbb{A} - \lambda\mathbb{I}$ multiplicada por el vector $\boldsymbol{x}$ nos da el vector nulo, es decir, los autovectores construyen el subespacio nulo de $\mathbb{A} - \lambda\mathbb{I}$. Para que esta expresión sea útil, el subespacio nulo debe contener otros vectores distintos del vector cero, lo que implica que $\mathbb{A} - \lambda\mathbb{I}$ debe ser *singular*, y esto ocurre si y solo si

$$\det(\mathbb{A} - \lambda\mathbb{I}) = 0 \tag{1.10}$$

que da origen a una ecuación polinómica en λ denominada *ecuación característica* de grado n. El Teorema Fundamental del Álgebra asegura que la matriz $\mathbb{A}$, de $n \times n$ elementos, siempre tiene n autovalores, aunque éstos pueden no ser reales ni tampoco todos distintos (autovalores iguales que provienen de raíces múltiples se denominan *degenerados*). Los autovalores complejos de una matriz real siempre ocurren en pares conjugados, esto es, si $\alpha + i\beta$ es un autovalor de una matriz real, también lo es $\alpha - i\beta$, siendo $i = \sqrt{-1}$ la unidad imaginaria. Una vez determinados los autovalores se pueden obtener los autovectores resolviendo la ecuación (1.9).

Tanto Sympy como SciPy tienen rutinas para calcular autovalores y autovectores. En el caso de Sympy podemos resolver un problema de autovalores con elementos simbólicos en la matriz $\mathbb{A}$. Por ejemplo, para la siguiente matriz simétrica:

Cell 01

```
import sympy as sy
sy.init_printing()
a, b = sy.symbols("alpha, beta")
A = sy.Matrix([[a, b],[b, -a]])
A
```

$$\begin{bmatrix} \alpha & \beta \\ \beta & -\alpha \end{bmatrix}$$

los autovalores se obtienen con `eigenvals()`:

Cell 02

```
A.eigenvals()
```

$$\left\{-\sqrt{\alpha^2+\beta^2} : 1,\ \sqrt{\alpha^2+\beta^2} : 1\right\}$$

La salida de `eigenvals()` es un diccionario, cuya clave es el autovalor y su valor es la multiplicidad del autovalor. Para obtener los autovectores de la matriz, utilizamos el método `eigenvects()`:

Cell 03

```
A.eigenvects()
```

$$\left[\left(-\sqrt{\alpha^2+\beta^2},\ 1,\ \left[\begin{bmatrix} -\frac{\beta}{\alpha+\sqrt{\alpha^2+\beta^2}} \\ 1 \end{bmatrix}\right]\right),\ \left(\sqrt{\alpha^2+\beta^2},\ 1,\ \left[\begin{bmatrix} -\frac{\beta}{\alpha-\sqrt{\alpha^2+\beta^2}} \\ 1 \end{bmatrix}\right]\right)\right]$$

En este caso, la salida es una lista cuyos elementos son tuplas. A su vez, cada tupla está compuesta por el autovalor, la multiplicidad del mismo y el correspondiente autovector. Podemos

desempacar esta información (notando que el autovector es una lista de un solo elemento) para verificar que los autovectores son ortogonales[1]:

CELL 04

```
(val1, _, vec1), (val2, _, vec2) = A.eigenvects()
sy.simplify(vec1[0].T * vec2[0])
```

$$\begin{bmatrix}0\end{bmatrix}$$

Por supuesto que también podemos usar Sympy para resolver el problema de autovalores con matrices con elementos fraccionales, por ejemplo nuestra conocida matriz de Hilbert 2×2:

CELL 05

```
H_s = sy.Matrix([[1, sy.Rational(1,2)], [sy.Rational(1,2), sy.Rational(1,3)]])
H_s.eigenvals()
```

$$\left\{\frac{2}{3} - \frac{\sqrt{13}}{6} : 1,\ \frac{\sqrt{13}}{6} + \frac{2}{3} : 1\right\}$$

CELL 06

```
H_s.eigenvects()
```

$$\left[\left(\frac{2}{3} - \frac{\sqrt{13}}{6},\ 1,\ \left[\begin{bmatrix}-\frac{1}{2\left(\frac{1}{3}+\frac{\sqrt{13}}{6}\right)}\\ 1\end{bmatrix}\right]\right),\ \left(\frac{\sqrt{13}}{6} + \frac{2}{3},\ 1,\ \left[\begin{bmatrix}-\frac{1}{2\left(\frac{1}{3}-\frac{\sqrt{13}}{6}\right)}\\ 1\end{bmatrix}\right]\right)\right]$$

El abordaje simbólico se hace impracticable para matrices mayores a 3×3 debido a la dificultad de encontrar las raíces de la ecuación característica. De echo, el teorema de Abel-Rufini[2] establece que no existe una solución general para la solución de ecuaciones polinómicas de grado cinco o mayor. Por esto tienen especial importancia los métodos numéricos para la solución del problema de autovalores.

Pese a que los problemas de autovalores son en cierta forma similares a los sistemas de ecuaciones lineales que vimos en la sección anterior, no podemos utilizar los mismos métodos para resolverlos: si a una fila de $\mathbb{A}$ le sumamos otra fila, o intercambiamos filas, los autovalores generalmente cambian. Es por esto que existen rutinas especializadas que resuelven eficientemente los problemas de autovalores (cuya descripción excede el alcance de este libro), y que por lo general obtienen una solución parcial. Por ejemplo, pueden obtener todos los autovalores y ningún autovector, o solo algunos autovectores, o solo los primeros autovalores mayores, o solo los negativos. Estas especializaciones permiten obtener resultados útiles sobre matrices grandes ahorrando tiempo de cálculo y espacio de almacenamiento.

SciPy tiene varias funciones para calcular autovalores y autovectores en el submódulo `linalg`, especializadas en abordar matrices con determinadas estrucrturas. Para matrices generales disponemos de `eigvals` (para autovalores) y `eig` para obtener autovalores y autovectores. Por ejemplo, para la matriz de Hilbert 4×4 podemos calcular sus autovalores y autovectores con `eig`:

[1] Los autovectores de una matriz normal, como la simétrica del ejemplo, con autovalores no degenerados (es decir, todos distintos) forman la base de un espacio vectorial n-dimensional.

[2] Ver entrada en Wikipedia.

CELL 07

```
import scipy.linalg as la
H = la.hilbert(4)
H
```

```
array([[1.        , 0.5       , 0.33333333, 0.25      ],
       [0.5       , 0.33333333, 0.25      , 0.2       ],
       [0.33333333, 0.25      , 0.2       , 0.16666667],
       [0.25      , 0.2       , 0.16666667, 0.14285714]])
```

CELL 08

```
vals, vects = la.eig(H)
vals
```

```
array([1.50021428e+00+0.j, 1.69141220e-01+0.j, 6.73827361e-03+0.j,
       9.67023040e-05+0.j])
```

CELL 09

```
vects
```

```
array([[ 0.79260829,  0.5820757 ,  0.17918629, -0.02919332],
       [ 0.45192312, -0.37050219, -0.74191779,  0.32871206],
       [ 0.3224164 , -0.50957863,  0.10022814, -0.79141115],
       [ 0.25216117, -0.51404827,  0.63828253,  0.51455275]])
```

Cada uno de los autovectores aparece como una fila del array mostrado en la celda 9. A diferencia de los valores que se obtienen con Sympy, estos autovalores se encuentran normalizados. En la celda 10 podemos verificar esta normalización (con precisión cercana al épsilon de la máquina), y también que los cuatro autovectores son ortogonales:

CELL 10

```
for i in range(len(vects)):
    for j in range(i, len(vects)):
        print(f'v[{i}] . v[{j}] = {vects[i] @ vects[j]:+.15f}')
```

```
v[0] . v[0] = +1.000000000000000
v[0] . v[1] = -0.000000000000000
v[0] . v[2] = +0.000000000000000
v[0] . v[3] = -0.000000000000000
v[1] . v[1] = +1.000000000000001
v[1] . v[2] = -0.000000000000001
v[1] . v[3] = +0.000000000000000
v[2] . v[2] = +1.000000000000001
v[2] . v[3] = +0.000000000000001
v[3] . v[3] = +0.999999999999999
```

En la celda 8 vemos que los autovalores se muestran en formato de número complejo (SciPy utiliza la letra `j` para denotar la parte compleja), aunque por ser la matriz de Hilbert una matriz real simétrica, todos sus autovalores son reales. `scipy.linalg` tiene una rutina específica para trabajar con matrices simétricas o hermíticas, `eigh`, que devuelve un array con los autovalores en formato de punto flotante real:

CELL 11

```
vals_real, vects_real = la.eigh(H)
vals_real
```

```
array([9.67023040e-05, 6.73827361e-03, 1.69141220e-01, 1.50021428e+00])
```

Podemos verificar dos propiedades de los autovalores, la que establece que la suma de los mismos es igual a la traza de la matriz:

CELL 12

```
import numpy as np
vals_real.sum()- np.trace(H)
```

$$-4{,}44089209850063 \cdot 10^{-16}$$

y que el producto de los autovalores dan el determinante de la matriz:

CELL 13

```
np.prod(vals_real) - la.det(H)
```

$$1{,}96670462440608 \cdot 10^{-20}$$

Además de la función para matrices simétricas/hermíticas, `linalg` tiene también rutinas para calcular autovalores y autovectores en matrices banda (`eig_banded`, `eigvals_banded` para solo autovalores), y tridiagonales (`eigh_tridiagonal`, `eigvalsh_tridiagonal` para obtener unicamente los autovalores). En el caso de trabajar con matrices ralas, el submódulo de SciPy `sparse.linalg` tiene las funciones `eigs` para obtener autovalores y autovectores de una matriz arbitraria, y `eigsh` para matrices simétricas o hermíticas.

Cabe destacar que tanto `linalg.eig` como `sparse.linalg.eig`, y sus versiones para matrices simétricas/hermíticas, permiten resolver también el problema de autovalores general

$$\mathbb{A} \cdot \boldsymbol{v} = \lambda \mathbb{B} \cdot \boldsymbol{v} \tag{1.11}$$

para las matrices cuadradas $\mathbb{A}$ y $\mathbb{B}$, que es una generalización del problema de autovalores usual con $\mathbb{A} = \mathbb{I}$.

1.3. Ecuaciones no lineales

El problema de resolver ecuaciones no lineales aparece frecuentemente en todas las áreas de la ciencia y la tecnología. A diferencia de los sistemas lineales, las soluciones de ecuaciones no lineales no pueden obtenerse fácilmente por sustitucion u otro método directo, por lo cual es necesario recurrir a métodos iterativos en los cuales comenzamos con una estimación inicial de la solución y luego nos acercamos a ella mediante aproximaciones sucesivas hasta satisfacer algún criterio de convergencia.

La forma general del problema puede enunciarse simplemente como

$$f(x) = 0 \tag{1.12}$$

donde f es una función no lineal de x. Cuando hay solo una variable dependiente, el problema es unidimensional o univariado, y resolver el problema consiste en hallar la *raíz* o *raíces* de la función. Un conjunto de n ecuaciones no lineales puede representarse en notación vectorial, y se trata de encontrar uno o más vectores solución $\boldsymbol{x}$ (de dimensión n) tal que

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{0} \tag{1.13}$$

donde $\boldsymbol{f}$ es una función vectorial n-dimensional cuyas componentes son las ecuaciones que deben satisfacerse simultáneamente. Pese a la aparente similitud que existe entre las ecuaciones (1.12) y (1.13), resolver sistemas de ecuaciones no lineales es *mucho* más difícil que el caso unidimensional, fundamentalmente porque en este último caso es posible encerrar la solución en un intervalo (*bracketing*), y mediante un algoritmo reducir este intervalo hasta dar con la raíz, mientras que en el caso multidimensional esto no es tan simple.

Una complejidad adicional proviene de la existencia de raíces múltiples, especialmente si la multiplicidad de la raíz es un número par. Esto puede cuantificarse mediante el número de condición que, del mismo modo que vimos en la subsección 1.1.1, establece una medida de cuánto afecta un cambio en el valor de la función al cambio del valor de su argumento. Para establecer este número de condición, llamemos x^* a la raíz de la ecuación (1.12), y $\tilde{x}^*$ a nuestra aproximación a la raíz. Expandiendo en serie de Taylor $f(\tilde{x}^*)$ alrededor de x^*, obtenemos

$$f(\tilde{x}^*) = f(x^* + \delta x^*) = f(x^*) + f'(x^*)\delta x^* + O\left(\delta x^{*2}\right) \tag{1.14}$$

en donde usamos $\delta x^* = \tilde{x}^* - x^*$. Al asumir que δx^* es un valor pequeño, podemos despreciar los términos de orden superior en (1.14), y entonces resulta:

$$\delta x^* = \tilde{x}^* - x^* \approx \frac{1}{f'(x^*)}\left[f(\tilde{x}^*) - f(x^*)\right] \tag{1.15}$$

lo que nos permite definir el *número de condición* de nuestro problema:

$$\kappa_f = \frac{1}{f'(x^*)} \tag{1.16}$$

Esta magnitud cuantifica el efecto de un pequeño cambio en el valor de la función sobre la evaluación de la raíz. En efecto, una función que cruza "rápidamente" el eje x tiene un valor grande de $f'(x^*)$, y por lo tanto un número de condición κ_f pequeño, lo que define un problema *bien condicionado*. Por otra parte, si la función es bastante "plana" alrededor de x^*, cruzando el eje x suavemente, esto corresponde a un valor pequeño de $f'(x^*)$ y en consecuencia a un valor grande de κ_f, lo que caracteriza a un problema *mal condicionado*. Este condicionamiento afecta a la dificultad de encontrar la raíz de f, ya que puede ser difícil distinguirla de valores vecinos.

1.3.1. Ecuaciones no lineales unidimensionales

Para abordar estos problemas siempre es conveniente "encerrar" la raíz en un intervalo y para ello es útil visualizar el comportamiento de la función. Los métodos cerrados, o de *bracketing*, consisten en determinar un intervalo inicial donde la función cambia de signo, y si dicha función es continua en el intervalo entonces los algoritmos de *bracketing* garantizan la convergencia sobre la raíz (aunque pueden hacerlo con diferente velocidad).

Por otra parte, cuando la raíz se encuentra en un intervalo en que la función no cambia de signo (como en el caso de raíces con multiplicidad par), es necesario utilizar métodos abiertos que no garantizan la convergencia a la solución. El submódulo `optimize` de SciPy ofrece una variedad de métodos tanto cerrados como abiertos. Veremos aquí algunos ejemplos de uso.

Comenzaremos analizando un polinomio de cuarto grado, cuyas raíces pueden obtenerse fácilmente por medio de su factorización, de modo que sabemos de antemano cuáles son para evaluar el desempeño de los algoritmos. El polinomio

$$p(x) = x^4 - 5x^3 + 8x^2 - 4x$$

tiene raíces simples en $x_1 = 0$ y $x_2 = 1$, y una raíz doble en ($x_{34} = 2$) Si graficamos esta función, tal como vemos en la celda 1 del *jupyter-notebook* siguiente

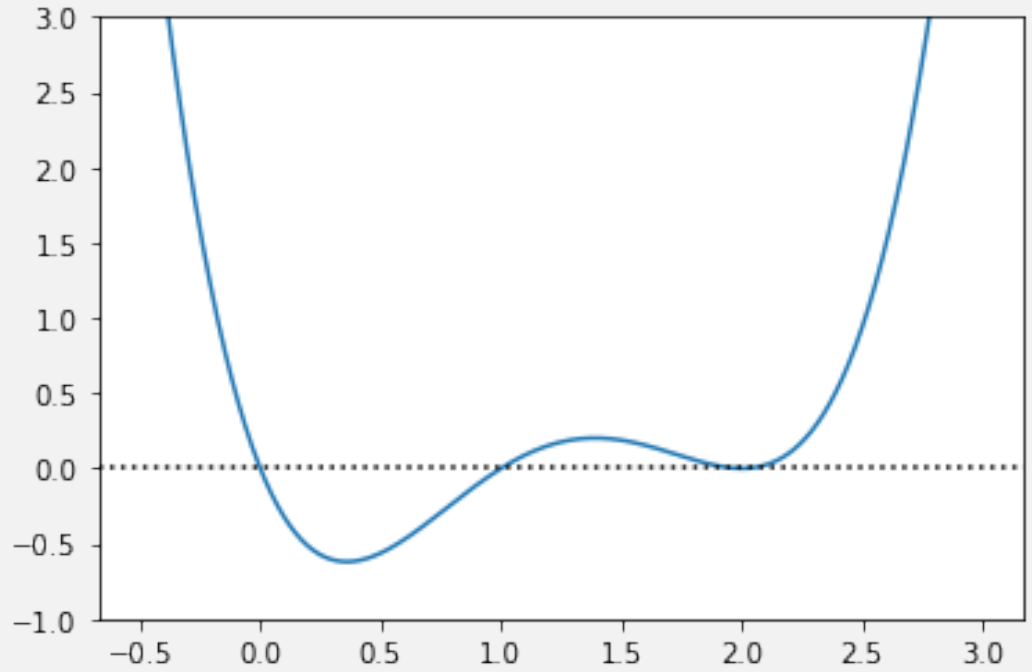
CELL 01

```
import matplotlib.pyplot as plt
import numpy as np
from scipy import optimize

def p(x):
    return x**4 - 5 * x**3 + 8 * x**2 - 4 * x

def p1(x): # Derivada primera de p
    return 4 * x**3 - 15 * x**2 + 16 * x - 4

x = np.linspace(-0.5, 3, 1000)
fig, ax = plt.subplots()
ax.plot(x, p(x), lw=1.5)
ax.set_ylim([-1, 3])
ax.axhline(0, ls=':', color='k');
```



vemos que $p(x)$ está bien condicionada en x_1 ($\kappa_p = -1/4$). En el submódulo encontramos la función `root_scalar` que permite hallar la raíz de una función mediante diversos métodos. Utilizaremos para el ejemplo el método de Brent, que combina un método de *bracketing* como el de bisección con el método de la secante y una interpolación cuadrática inversa, lo que brinda un algoritmo robusto y de rápida convergencia.

CELL 02

```
sol_1 = optimize.root_scalar(p, method='brentq', bracket=[-0.5, 0.5])
sol_1
```

```
      converged: True
           flag: 'converged'
  function_calls: 11
     iterations: 10
           root: 1.2352144989557884e-14
```

CELL 03

```
sol_2 = optimize.root_scalar(p, method='brentq', bracket=[0.5, 1.5])
sol_2
```

```
       converged: True
            flag: 'converged'
   function_calls: 10
      iterations: 9
            root: 1.0000000000000004
```

En la celda 2 vemos que llamamos a la función `root_scalar` pasando como primer argumento a la función `p`, luego seleccionamos el método de solución `'brentq'`, y por último establecemos el intervalo que encierra la raíz buscada, que cumple con la condición que $p(x)$ tiene signos opuestos en los extremos del intervalo. Como resultado de la función `root_scalar`, obtenemos un objeto `RootResults` que almacenamos en la variable `sol_1`, y que contiene diversos atributos que dan cuenta del resultado de la aplicación del algoritmo como el motivo de la finalización (`flag`), si hubo convergencia (`converged`), la cantidad de iteraciones y llamadas a la función, y por supuesto, la raíz encontrada. En la celda 3 volvemos a aplicar el método de Brent para hallar x_2, encerrando esta raíz en el intervalo $[1/2, 3/2]$.

La situación es diferente para la raíz doble en x_{34}, donde la función presenta un mal condicionamiento ya que también la derivada de $p(x)$ se anula en ese punto. Si queremos determinar la raíz, no podemos usar un método de *bracketing* dado que la función no cambia de signo a ambos lados de la raíz. Podemos acudir entonces a métodos abiertos, como el de Newton, que no nos garantiza la convergencia a la solución pero si partimos de una estimación inicial suficientemente cercana, tenemos buenas posibilidades de éxito. La celda 4 muestra el uso de la función `root_scalar` seleccionando el método de Newton. A diferencia del caso anterior, no especificamos un intervalo que encierra a la raíz, pero si un valor inicial `x0`. Además, el método de Newton requiere especificar también la derivada de la función en el argumento `fprime`, al que le pasamos la función `p1(x)` que definimos en la celda 1 (si no disponemos de una expresión para la derivada, es posible utilizar el método de la secante, en el que la derivada se aproxima por diferencia finita):

CELL 04

```
sol_3 = optimize.root_scalar(p, method='newton', x0=2.5, fprime=p1)
sol_3
```

```
       converged: True
            flag: 'converged'
   function_calls: 61
      iterations: 30
            root: 1.9999999814380152
```

Puede verse que en comparación con el método de Brent, el método de Newton es más costoso computacionalmente debido la complejidad misma del método (que requiere evaluar dos funciones). Además en este caso la convergencia es más lenta debido al mal condicionamiento de la función en x_{34}.

Para mostrar la importancia de la elección de la estimación inicial, notemos que $p(x)$ tiene un máximo relativo en $x_m = (\sqrt{17} + 7)/8 \approx 1{,}3904$. Si el valor de `x0` se encuentra a la derecha de x_m, el método de Newton convergerá a la solución buscada, tal como muestra la celda 5, mientras que si nuestra estimación inicial está a la izquierda de x_m, el método converge a x_1 como puede verse en la celda 6. Una variación de apenas 0.072 % respecto del valor de x_m produce un cambio muy grande en la raíz encontrada, incluso "saltando" la raíz en x_2. Si nuestro valor inicial se encuentra más próximo al valor de x_m, el método no converge a ninguna raíz en el número máximo de iteraciones por defecto (50). Aumentando el número máximo de iteraciones vemos

que converge a x_{34} en 70 iteraciones (pero no siempre tenemos esa suerte).

CELL 05

```
sol_md = optimize.root_scalar(p, method='newton', x0=1.391, fprime=p1)
sol_md
```

```
      converged: True
           flag: 'converged'
 function_calls: 87
     iterations: 43
           root: 2.00000002189259
```

CELL 06

```
sol_mi = optimize.root_scalar(p, method='newton', x0=1.390, fprime=p1)
sol_mi
```

```
      converged: True
           flag: 'converged'
 function_calls: 50
     iterations: 25
           root: -5.9164567891575885e-31
```

CELL 07

```
sol_m = optimize.root_scalar(p, method='newton', x0=1.3904, fprime=p1)
sol_m
```

```
      converged: False
           flag: 'convergence error'
 function_calls: 100
     iterations: 50
           root: 2.000002582731664
```

CELL 08

```
sol_m71 = optimize.root_scalar(p, method='newton', x0=1.3904, fprime=p1, maxiter=71)
sol_m71
```

```
      converged: True
           flag: 'converged'
 function_calls: 141
     iterations: 70
           root: 2.000000047778081
```

Se puede suponer que podemos encontrar x_{34} utilizando un método de *bracketing* utilizando un intervalo más grande que contenga a esta raíz. Sin embargo, esto no siempre funciona, tal como muestra el resultado de la celda 9 en el que el método de Brent utilizando el intervalo $[1/2, 3/2]$ encuentra la raíz en x_1, pero no la raíz doble.

CELL 09

```
sol_x = optimize.root_scalar(p, method='brentq', bracket=[0.5, 2.5])
sol_x
```

```
      converged: True
           flag: 'converged'
 function_calls: 10
     iterations: 9
           root: 1.0000000000002822
```

1.3.2. Sistema de ecuaciones no lineales

Tal como anticipamos al comienzo de la sección 1.3, resolver un sistema de ecuaciones no lineales es mucho más complejo que el caso unidimensional, en parte debido a la gran variedad de escenarios posibles. Para mencionar solo un aspecto de esta complejidad, suele ser habitual no saber si existe solución del problema, o en caso de existir puede que no sea posible determinar cuántas hay. En consecuencia no existe un método que garantice la convergencia a una solución, tal como el método de bisección para el caso unidimensional. Por otra parte, los métodos disponibles son mas costosos computacionalmente que para el caso multidimensional, y este costo crece aceleradamente al aumentar el número de incógnitas.

En este contexto, SciPy ofrece un conjunto de métodos para abordar la solución de estos sistemas. Estos métodos pueden agruparse según el número de ecuaciones a resolver. El método híbrido de Powell (`hibr`) y la implementación del método de Levenberg-Marquardt (`lm`) de MINPACK son eficientes para para un sistema de pocas ecuaciones. Cuando el número de ecuaciones n aumenta, la aplicación de estos métodos se vuelve ineficiente debido a que requieren la inversión de una matriz jacobiana densa $n \times n$ en cada iteración. Para estos casos, SciPy ofrece funciones que implementan el método inexacto de Newton, en los que la matriz jacobiana se evalúa en forma aproximada. Estos métodos son `anderson`, `broyden2` y `krylov`.

Como ejemplo (sencillo) de resolución de un sistema no lineal de ecuaciones, determinaremos las raíces del sistema

$$\begin{cases} f_1(x, y) = y - x^2 + x\cos(\pi x) = 0 \\ f_2(x, y) = y - x^3 + 4x - \ln(y + 1)^4 = 0 \end{cases} \tag{1.17}$$

Por ser un sistema con dos incógnitas, cada función define una superficie tridimensional que cruza el plano xy. El siguiente *script* genera la figura 1.1 en donde visualizamos para cada función la curva de nivel cero, es decir, las intersecciones de las superficies generadas por $f_1(x, y)$ y $f_3(x, y)$ con el plano xy (ver las líneas 16 y 18 del *script*). Esto significa que sobre cada una de esas curvas las funciones se anulan individualmente. Y en los puntos donde las curvas se cruzan, ambas funciones se anulan *simultáneamente*. Esto puede orientarnos a proponer valores iniciales para los algoritmos de solución que estén cercanos a los puntos que son solución del sistema.

```
1  #!/usr/bin/env python3
2
3  import numpy as np
4  import matplotlib.pyplot as plt
5  from math import pi
6
7  plt.rcParams.update({"text.usetex": True})
8
9  n_puntos = 300
10 xs = np.linspace(-3, 3, n_puntos)
11 ys = np.linspace(-5, 15, n_puntos)
12 X, Y = np.meshgrid(xs, ys)
13 Z1 = Y - X**2 + X * np.cos(pi * X)
14 Z2 = Y -(X -2)*(X+2) * X - np.log((Y+1)**4)
15 fig, ax = plt.subplots()
16 CS1 = ax.contour(X, Y, Z1, levels=[0], colors=['royalblue'])
17 ax.clabel(CS1, levels=[0], fmt={0:r'$f_1$'}, fontsize=12)
18 CS2 = ax.contour(X, Y, Z2, levels=[0], colors=['maroon'])
19 ax.clabel(CS2, levels=[0], fmt={0:r'$f_2$'})
20 plt.savefig('f1_f2_ceros.pdf')
```

En el siguiente programa determinaremos las soluciones del sistema no lineal de ecuaciones (1.17), y exploraremos las regiones de convergencia a cada solución. En las líneas 3–6 importamos

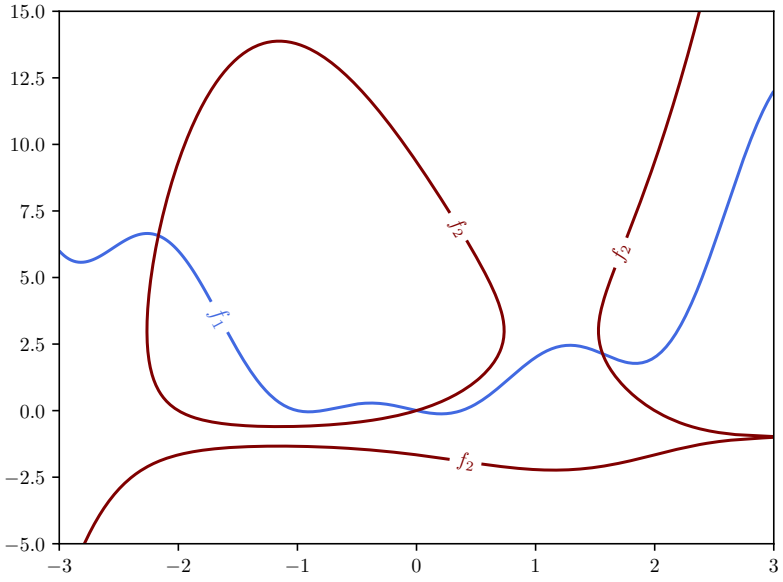


FIGURA 1.1: Ceros de las funciones del sistema no lineal de ecuaciones.

los módulos NumPy, Matplotlib, las funciones coseno, logaritmo (natural) y la constante π del módulo `math`, y el submódulo `optimize` de SciPy. En la línea 8 especificamos que el renderizado de la figura se haga utilizando LaTeXpor imprescindibles cuestiones estéticas.

```
1  #!/usr/bin/env python3
2
3  import numpy as np
4  import matplotlib.pyplot as plt
5  from math import cos, log, pi
6  from scipy import optimize
7
8  plt.rcParams.update({"text.usetex": True})
9
10 n_puntos = 300
11 xs = np.linspace(-3, 3, n_puntos)
12 ys = np.linspace(-5, 15, n_puntos)
13 X, Y = np.meshgrid(xs, ys)
14 Z1 = Y - X**2 + X * np.cos(pi * X)
15 Z2 = Y -(X -2)*(X+2) * X - np.log((Y+1)**4)
16 fig, ax = plt.subplots()
17 CS1 = ax.contour(X, Y, Z1, levels=[0], colors=['royalblue'])
18 ax.clabel(CS1, levels=[0], fmt={0:r'$f_1$'}, fontsize=12)
19 CS2 = ax.contour(X, Y, Z2, levels=[0], colors=['maroon'])
20 ax.clabel(CS2, levels=[0], fmt={0:r'$f_2$'})
21
22
23 def f(x):
24     return[x[1] - x[0]**2 + x[0] * cos(pi * x[0]),
25            x[1] - x[0]**3 + 4 * x[0] - log((x[1] + 1)**4)]
26
27 x_guess = [[-2.0, 5.0], [0, 0], [1.5, 2.0]]
28 sols = []
29 for x_g in x_guess:
30     sol = optimize.root(f, x_g, method='hybr')
31     print(sol.message)
```

```
32      print(f'x: [{sol.x[0]:.3f}, {sol.x[1]:.3f}] -- root: {f(sol.x)}')
33      sols.append(sol.x)
34
35  colors = ['r', 'b', 'g']
36  for i, s in enumerate(sols):
37      ax.plot(s[0], s[1], colors[i]+'*', markersize=15)
38
39  n_trial = 100
40  xs = np.linspace(-3, 3, n_trial)
41  ys = np.linspace(-5, 15, n_trial)
42  n_div = 0
43  for xc in xs:
44      for yc in ys:
45          ss = optimize.root(f, [xc, yc], method='hybr')
46          if ss.success:
47              idx = (abs(sols - ss.x)**2).sum(axis=1).argmin()
48              ax.plot(xc, yc, marker='.', color=colors[idx], alpha=0.15)
49          else:
50              n_div += 1
51  print(f'Proporción de no convergencia: {n_div / n_trial**2:.4f}')
52  plt.ylim([-5, 15])
53  plt.savefig('sol_hybr.pdf')
```

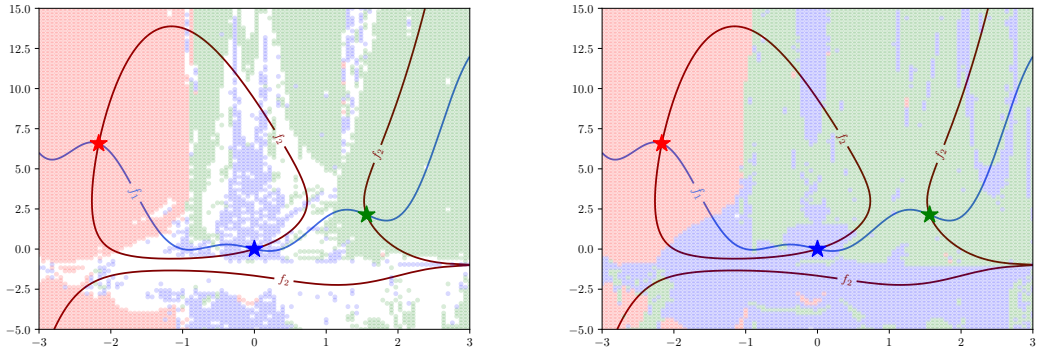
Entre las líneas 10 y 20 generamos nuevamente el mismo gráfico que mostramos en la figura 1.1, que usaremos como base para mostrar encima las raíces y las cuencas de convergencia. Implementamos la definición del sistema de ecuaciones en la función `f(x)` de las líneas 23–25, que tiene como argumento una lista de dos elementos ($x \mapsto$ `x[0]`, $y \mapsto$ `x[1]`), y devuelve también una lista con dos elementos que representa cada función evaluada en los argumentos.

En la línea 28 definimos la lista `x_guess` que contiene tres pares de valores iniciales con los que vamos a iniciar la búsqueda de soluciones. Estos valores fueron elegidos inspeccionado visualmente la figura 1.1 de modo que estén "cerca" de los lugares donde se cruzan las curvas de nivel cero de f_1 y f_2. Si encotramos soluciones, las iremos almacenando en la lista `sols`.

En el bucle de las líneas 29–33 utilizamos cada valor inicial como argumento de la función `root` del submódulo `optimize`, al que le pasamos también la función `f` que representa las ecuaciones y la cadena `'hybr'` al argumento `method`, que es el método por defecto de `root` (y que por lo tanto puede ser omitido). `root` devuelve un objeto que guardamos en `sol` del tipo `OptimizeResult` que contiene varios atributos, y del que imprimimos `sol.message` (que describe la causa de la finalización), `sol.x` que muestra la solución del sistema y `sol.f` que nos permite verificar el valor de las funciones en la solución encontrada. Al ejecutarse este programa la salida es:

```
❯ ./sistema_no_lineal.py
The solution converged.
x: [-2.169, 6.574] -- root: [2.5308866113959994e-11, 9.906742093335197e-12]
The solution converged.
x: [0.000, 0.000] -- root: [0.0, 0.0]
The solution converged.
x: [1.562, 2.137] -- root: [9.992007221626409e-16, -1.4210854715202004e-14]
Proporción de no convergencia: 0.2510
```

El objeto `sol` del tipo `OptimizeResult` tiene otros atributos que no mostramos en este programa, pero que resulta útil inspeccionar (se puede ver la lista de atributos en este enlace). Particularmente el atributo booleano `success` que es `True` cuando el algoritmo converge a una solución. A continuación, en las líneas 35–37 marcamos en nuestra figura en construcción las raíces encontradas, con estrellas de colores rojo, azul y verde respectivamente. Utilizaremos este código de colores asignado a cada solución para representar las regiones de convergencia hacia cada una.



(A) Método híbrido de Powell.

(B) Método de Levenberg-Marquardt.

FIGURA 1.2: Regiones de convergencia.

Para determinar estas regiones de convergencia armamos una grilla de `n_trial = 100` puntos en cada variable (representada por los arrays `xs` y `ys`), y para cada punto de esta grilla (que recorremos en el bucle doble de las líneas 44 y 45) intentamos obtener una solución utilizando el método híbrido de Powell (`'hybr'`). Si hay convergencia, determinamos a cuál de las soluciones llegó el algoritmo a partir del valor inicial (línea 47), y marcamos este valor inicial con un pequeño círculo con el color correspondiente (línea 48). En caso de no haber convergencia, incrementamos el valor de `n_div` que dará cuenta del número total de valores iniciales que no convergieron a ninguna solución.

Tal como muestra la última línea de la salida anterior mostrada, al ejecutar el algoritmo con el método híbrido de Powell hay un 25,1 % de puntos que no convergen a ninguna solución, al menos con el máximo número de llamadas a la función o tolerancia establecidos por defecto. Esto se visualiza en la figura 1.2a como zonas blancas. Por el contrario, si ejecutamos nuevamente el programa pero cambiando el argumento `method` de `root` en la línea 45 por el de Levenberg-Marquardt `'lm'`, la proporción de no convergencia se reduce al 0,03 %, mostrando que este método es más robusto. Se puede ver en la figura 1.2b que casi no hay zonas en blanco.

1.4. Lecturas recomendadas

- Para profundizar en cuestiones de álgebra lineal, recomendamos el clásico libro de Strang: Gilbert Strang. *Linear Algebra and Its Applications, 4th Edition*. 4th. Brooks Cole, 2006, y también Stephen H. Friedberg, Arnold J. Insel y Lawrence E. Spence. *Linear Algebra*. 4th ed. Pearson, 2014.

- Se puede ver una lista detallada de métodos de resolución de sistemas de ecuaciones lineales en R.K. Gupta. *Numerical Methods. Fundamentals and Applications*. Cambridge University Press, 2019.

- El libro de Burden, Faires y Burden tiene una excelente introducción a los métodos de resolución de ecuaciones en una y varias variables: Richard L. Burden, J. Douglas Faires y Annette M. Burden. *Numerical Analysis*. 9th ed. Cengage Learning, 2016.

Parte III
Apéndices

A | Zen de Python

Incluimos aquí las frases traducidas correspondientes al Zen de Python [6].

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Espaciado es mejor que denso.
- La legibilidad es importante.
- Los casos especiales no son lo suficientemente especiales como para romper las reglas.
- Sin embargo la practicidad le gana a la pureza.
- Los errores nunca deberían pasar silenciosamente.
- A menos que se silencien explícitamente.
- Frente a la ambigüedad, evitar la tentación de adivinar.
- Debería haber una, y preferiblemente solo una, manera obvia de hacerlo.
- A pesar de que esa manera no sea obvia a menos que seas Holandés.
- Ahora es mejor que nunca.
- A pesar de que nunca es muchas veces mejor que *ahora* mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres son una gran idea, ¡tengamos más de esos!

Bibliografía

[1] URL: https://creativecommons.org/licenses/by-nc-sa/4.0/deed.es.

[2] Gilbert Strang. *Linear Algebra and Its Applications, 4th Edition*. 4th. Brooks Cole, 2006.

[3] Stephen H. Friedberg, Arnold J. Insel y Lawrence E. Spence. *Linear Algebra*. 4th ed. Pearson, 2014.

[4] R.K. Gupta. *Numerical Methods. Fundamentals and Applications*. Cambridge University Press, 2019.

[5] Richard L. Burden, J. Douglas Faires y Annette M. Burden. *Numerical Analysis*. 9th ed. Cengage Learning, 2016.

[6] Tim Peters. *The Zen of Python*. 19 de ago. de 2004. URL: https://www.python.org/dev/peps/pep-0020/.