



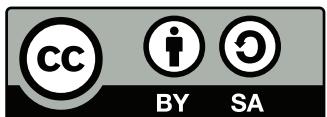
Universidad Complutense de Madrid

---

Laboratorio de Computación Científica  
Laboratory for Scientific Computing

Juan Jiménez  
Héctor García de Marina  
Lía García

2 de septiembre de 2024



El contenido de estos apuntes está bajo licencia Creative Commons Atribución-ShareAlike 4.0  
<http://creativecommons.org/licenses/by-sa/4.0/>  
©Juan Jiménez

# Índice general

<b>1. Introducción al software científico</b>	
<b>Introduction to scientific software</b>	<b>21</b>
1.1. Introducción a los computadores . . . . .	23
1.1. Introduction to computers . . . . .	23
1.1.1. Niveles de descripción de un ordenador . . . . .	23
1.1.1. Computer description levels . . . . .	23
1.1.2. El modelo de computador de Von Neumann . . . . .	27
1.1.2. The Von Neumann's computer model . . . . .	27
1.1.3. Representación binaria . . . . .	29
1.1.3. Binary coding . . . . .	29
1.2. Aplicaciones de Software Científico . . . . .	34
1.2. Scientific software application . . . . .	34
<b>2. Introducción a la programación en Python</b>	
<b>Introduction to Python Programming</b>	<b>37</b>
2.1. Un entorno de programación para Python . . . . .	38
2.1. A python's development environment . . . . .	38
2.1.1. Anaconda. Una distribución de herramientas de computación de software abierto . . . . .	38
2.1.1. Anaconda. A free software computing tools distribution . . . . .	38
2.1.2. Spyder . . . . .	42
2.1.2. Spyder . . . . .	42
2.1.3. Variables . . . . .	45
2.1.3. Variables . . . . .	45
2.2. Operaciones aritméticas, relaciones y lógicas. . . . .	54
2.2. Arithmetical, relational and logical operations . . . . .	54
2.2.1. Operaciones aritméticas . . . . .	54
2.2.1. Arithmetic operations . . . . .	54
2.2.2. Precedencia de los operadores aritméticos . . . . .	56
2.2.2. Arithmetic operator precedence . . . . .	56
2.2.3. Operaciones Relacionales y lógicas. . . . .	58
2.2.3. Relational and logical operations. . . . .	58
2.3. Scripts en Python . . . . .	64
2.3. Scripts in Python . . . . .	64
2.3.1. El editor de textos de Spyder. . . . .	65
2.3.1. The Spyder's text editor . . . . .	65
2.3.2. Ejecución de scripts . . . . .	68
2.3.2. Running a script . . . . .	68

2.4. Funciones en Python . . . . .	69
2.4. Functions in Python . . . . .	69
2.5. Los <i>NameSpace</i> en Python y el ámbito de las variables . . . . .	74
2.5. Namespace and scope in Python . . . . .	74
2.6. Depuración . . . . .	80
2.6. Debugging . . . . .	80
2.7. Control de flujo . . . . .	85
2.7. Flow control . . . . .	85
2.7.1. Flujo condicional. . . . .	85
2.7.1. Conditional flow. . . . .	85
2.7.2. Bucles . . . . .	93
2.7.2. Loops . . . . .	93
2.7.3. Funciones recursivas. . . . .	103
2.7.3. recursive functions. . . . .	103
2.7.4. Algoritmos y diagramas de flujo. . . . .	105
2.7.4. Algorithms and flow charts. . . . .	105
2.8. Exportar e importar datos en Python . . . . .	110
2.7.5. Reading and writing data in Pyhton . . . . .	110
2.9. Ejercicios . . . . .	118
<b>3. Introducción a Numpy</b>	
<b>Introduction to Numpy</b>	<b>123</b>
3.1. Numpy: un paquete de Python para cálculo numérico . . . . .	123
3.1. Numpy: a Python library for scientific computing . . . . .	123
3.1.1. Vectores y matrices en Numpy. . . . .	126
3.1.1. Vectors and matrices in Numpy. . . . .	126
3.2. Operaciones matriciales . . . . .	134
3.2. Matrix Operations . . . . .	134
3.2.1. Funciones incluidas en Numpy. . . . .	155
3.2.1. Functions included in Numpy . . . . .	155
3.3. Operadores vectoriales . . . . .	155
3.3. Vectorial operators . . . . .	155
3.4. Tipos de matrices empleados frecuentemente. . . . .	168
3.4. Kinds of matrices frequently used. . . . .	168
3.5. Factorización de matrices . . . . .	170
3.5.1. Factorization LU . . . . .	170
3.5. Matrix factorization. . . . .	170
3.5.1. LU factorization. . . . .	170
3.5.2. Factorización de Cholesky . . . . .	173
3.5.2. Cholesky factorisation . . . . .	173
3.5.3. Diagonalización . . . . .	174
3.5.3. Diagonalisation . . . . .	174
3.5.4. Factorización QR . . . . .	180
3.5.4. QR Factorisation . . . . .	180
3.5.5. Factorización SVD . . . . .	181
3.5.5. SVD Factorisation . . . . .	181
3.6. Ejercicios . . . . .	183

<b>4. Introducción a matplotlib</b>	
<b>Introduction to matplotlib</b>	<b>187</b>
4.1. Dibujar en 2D . . . . .	188
4.1.1. Trabajar con múltiples figuras . . . . .	190
4.1.1. Working with multiple figures . . . . .	190
4.1.2. Ejes no lineales . . . . .	193
4.1.2. Nonlinear axis . . . . .	193
4.1.3. Gráfica en coordenadas polares . . . . .	195
4.1.3. Polar coordinates graph . . . . .	195
4.1.4. Otras representaciones gráficas . . . . .	196
4.1.4. Other graphical representations . . . . .	196
4.2. Dibujar en 3D . . . . .	197
4.2. 3D plots . . . . .	197
4.2.1. Superficies . . . . .	199
4.3. Animaciones . . . . .	203
4.3. Animations . . . . .	203
4.4. Ejercicios . . . . .	205
4.5. Test del curso 2020/21 . . . . .	206
<b>5. Aritmética del Computador y Fuentes de error</b>	
<b>Computer Arithmetics and Errors</b>	<b>209</b>
5.1. Representación binaria y decimal . . . . .	209
5.1. Binary and denary representations . . . . .	209
5.2. Representación de números en el ordenador . . . . .	211
5.2. Computer number representation . . . . .	211
5.2.1. Números no enteros: Representación en punto fijo y en punto flotante . . . . .	214
5.2.1. Non-integer numbers: fixed-point and floating-point representations. . . . .	214
5.3. Errores en la representación numérica. . . . .	227
5.3. Numerical representations Errors. . . . .	227
5.3.1. Error de redondeo unitario . . . . .	228
5.3.1. round-off error . . . . .	228
5.3.2. Errores de desbordamiento . . . . .	232
5.3.2. Underflow and Overflow errors . . . . .	232
5.4. Errores derivados de las operaciones aritméticas . . . . .	233
5.4. Errors derived from arithmetic operation . . . . .	233
5.4.1. Acumulación de errores de redondeo . . . . .	234
5.4.1. Accumulation of round-off errors. . . . .	234
5.4.2. Anulación catastrófica . . . . .	239
5.4.2. Catastrophic cancellation. . . . .	239
5.4.3. Errores de desbordamiento . . . . .	240
5.4.3. Overflow and underflow errors . . . . .	240
<b>6. Cálculo de raíces de una función</b>	
<b>Methods for calculating the roots of a function</b>	<b>245</b>
6.1. Raíces de una función . . . . .	245
6.1. Roots of a function . . . . .	245
6.2. Metodos iterativos locales . . . . .	248
6.2.1. Método de la bisección . . . . .	248
6.2. Local iterative methods . . . . .	248

6.2.1.	Bisection Method . . . . .	248
6.2.2.	Método de interpolación lineal o ( <i>Regula falsi</i> ) . . . . .	252
6.2.2.	False position method or ( <i>Regula falsi</i> ) . . . . .	252
6.2.3.	Método de Newton-Raphson . . . . .	255
6.2.3.	Newton-Raphson method . . . . .	255
6.2.4.	Método de la secante . . . . .	259
6.2.4.	Secant method . . . . .	259
6.2.5.	Método de las aproximaciones sucesivas o del punto fijo . . . . .	264
6.2.5.	Method of successive approximations or fixed point iteration . . . . .	264
6.3.	Cálculo de raíces de funciones con Python. . . . .	273
6.3.1.	La función <code>fsolve</code> . . . . .	273
6.3.	Python root finding . . . . .	273
6.3.1.	Function <code>fsolve</code> . . . . .	273
6.3.2.	Cálculo de raíces de polinomios. . . . .	274
6.3.2.	Polynomials roots . . . . .	274
6.4.	Cálculo de raíces con la precisión máxima . . . . .	277
6.4.	Computing roots at maximum precision. . . . .	277
6.5.	Ejercicios . . . . .	279
6.5.	Problems . . . . .	279
6.6.	Test del curso 2020/2021 . . . . .	283
6.6.	2020/21 Test . . . . .	283
<b>7.</b>	<b>Sistemas de ecuaciones lineales</b>	
	<b>Linear equation systems</b>	<b>291</b>
7.1.	Introducción . . . . .	291
7.1.	Introduction . . . . .	291
7.2.	Condition number . . . . .	296
7.2.	Condicionamiento . . . . .	297
7.3.	Métodos directos . . . . .	299
7.3.1.	Sistemas triangulares . . . . .	299
7.2.1.	Direct methods . . . . .	299
7.2.2.	Triangular systems . . . . .	299
7.3.2.	Métodos basados en las factorizaciones . . . . .	302
7.2.3.	Methods based on factorisations . . . . .	302
7.3.3.	El método de eliminación de Gauss. . . . .	310
7.2.4.	The Gaussian elimination method. . . . .	310
7.3.4.	Gauss-Jordan y matrices en forma reducida escalonada . . . . .	318
7.2.5.	Gauss-Jordan and matrices in stepwise reduced form . . . . .	318
7.4.	Métodos iterativos . . . . .	320
7.3.	Iterative methods . . . . .	320
7.4.1.	El método de Jacobi. . . . .	322
7.3.1.	Jacobi's method . . . . .	322
7.4.2.	El método de Gauss-Seidel. . . . .	328
7.3.2.	The Gauss-Seidel method . . . . .	328
7.4.3.	Amortiguamiento. . . . .	332
7.3.3.	Weighted methods. . . . .	332
7.4.4.	Análisis de convergencia . . . . .	335
7.3.4.	Convergence analysis . . . . .	335
7.5.	Ejercicios . . . . .	340
7.6.	Test del curso 2020/21 . . . . .	343

<b>8. Interpolación y ajuste de funciones</b>	<b>345</b>
8.1. El polinomio de Taylor . . . . .	346
8.1.1. The Taylor's polynomial . . . . .	346
8.2. Interpolación polinómica. . . . .	350
8.2.1. Polynomial interpolation . . . . .	350
8.2.1.1. La matriz de Vandermonde . . . . .	352
8.2.1.2. The Vandermonde's matrix . . . . .	352
8.2.1.3. El polinomio interpolador de Lagrange . . . . .	353
8.2.1.4. Lagrange Interpolating Polynomial . . . . .	353
8.2.1.5. Diferencias divididas. . . . .	355
8.2.1.6. Divided differences. . . . .	355
8.2.1.7. El polinomio de Newton-Gregory . . . . .	359
8.2.1.8. The Newton-Gregory polynomial . . . . .	359
8.3. Interpolación por intervalos. . . . .	362
8.3.1. Piecewise interpolation . . . . .	362
8.3.1.1. Interpolación mediante splines cúbicos . . . . .	364
8.3.1.2. Cubic spline interpolation . . . . .	364
8.3.1.3. Funciones propias de Python para interpolación por intervalos . . . . .	371
8.3.1.4. Python functions for piecewise interpolation . . . . .	371
8.4. Ajuste polinómico por el método de mínimos cuadrados . . . . .	374
8.4.1. Least squared error method for polynomial data fitting . . . . .	374
8.4.1.1. Mínimos cuadrados en Python. . . . .	379
8.4.1.2. Least squares in Python. . . . .	379
8.4.1.3. Análisis de la bondad de un ajuste por mínimos cuadrados. . . . .	383
8.4.1.4. Analyzing goodness of fit using least squares method. . . . .	383
8.5. Curvas de Bézier . . . . .	385
8.5.1. Bézier Curves . . . . .	385
8.6. ejercicios . . . . .	395
8.6.1. exercises . . . . .	395
8.7. Test del curso 2020/21 . . . . .	397
8.7.1. Course 2020/21 test . . . . .	397
<b>9. Diferenciación e Integración numérica</b>	
<b>Numerical differentiation and integration</b>	<b>401</b>
9.1. Diferenciación numérica. . . . .	402
9.1.1. Numerical differentiation. . . . .	402
9.1.1.1. Diferenciación numérica basada en el polinomio de interpolación. . . . .	403
9.1.1.1.1. Numerical differentiation base on the interpolation polynomial. . . . .	403
9.1.1.2. Diferenciación numérica basada en diferencias finitas . . . . .	404
9.1.1.3. Numerical differentiation base on finite differences . . . . .	404
9.2. Integración numérica. . . . .	408
9.2.1. Numerical Integration. . . . .	408
9.2.1.1. La fórmula del trapezio. . . . .	410
9.2.1.1.1. Trapezium formula . . . . .	410
9.2.1.1.2. Las fórmulas de Simpson. . . . .	412
9.2.1.1.3. Simpson's formulae . . . . .	412
9.3. Problemas de valor inicial en ecuaciones diferenciales . . . . .	418

9.3.	Differential equations. Initial value problems . . . . .	418
9.3.1.	El método de Euler. . . . .	420
9.3.1.	The Euler's method. . . . .	420
9.3.2.	Métodos de Runge-Kutta . . . . .	428
9.3.2.	The Runge-Kutta methods . . . . .	428
9.4.	Ejercicios . . . . .	432
9.4.	Exercises . . . . .	432
9.5.	Test del curso 2020/21 . . . . .	435
9.5.	Course 2020/21 test. . . . .	435

# Índice de figuras

1.1. Descripción por niveles de un computador . . . . .	24
1.1. Computer level description . . . . .	24
1.2. Modelo de Von Neumann . . . . .	27
1.2. Von Neumann's model . . . . .	27
2.1. Ventana de Anaconda Navigator. Se ha señalado en rojo el icono correspondiente a Spyder. . . . .	39
2.2. Ventana de Anaconda-Navigator, se ha señalado en rojo el botón Environments. . . . .	40
2.3. Ventana de Anaconda-Navigator, se ha señalado en rojo el botón Environments. . . . .	40
2.4. Ventana de Anaconda-Navigator, se ha señalado en rojo el botón Environments. . . . .	42
2.4. Anaconda-Navgator window. The Environments button has been encircled in red . . . . .	42
2.5. Ventana de Anaconda-Navigator, se ha señalado en rojo el botón Environments. . . . .	43
2.5. Anaconda-Navgator window. The Environments button has been encircled in red . . . . .	43
2.6. Spyder: Entorno de desarrollo integrado para Python . . . . .	44
2.6. Spyder: An integer development environment for Python . . . . .	44
2.7. Tipos de datos en Python. . . . .	48
2.7. Data types in Python . . . . .	48
2.8. Editor de textos para Python incluido en Spyder . . . . .	66
2.8. Text editor for Python included in Spyder . . . . .	66
2.9. Relación de inclusión entre los Namesapces de Python . . . . .	78
2.9. Inclusion relationship among Python Namespaces . . . . .	78
2.10. Un ejemplo de error de sintaxis (falta cerrar un paréntesis . . . . .	81
2.10. A syntax error example (a closing bracket has been omitted . . . . .	81
2.11. Breakpoint activo señalado con una flecha azul, y menú desplegable <i>debug</i> . . . . .	83
2.11. Active Breackpoint pointed by a blue arrow and <i>debug</i> drop-down menu . . . . .	83
2.12. Parada de programa en un breakpoint y herramientas de depuración . . . . .	84
2.12. program stop on a breakpoint and debugging tools . . . . .	84
2.13. Esquema general de la estructura de flujo condicional <b>if</b> los términos escritos entre paréntesis son opcionales. . . . .	90
2.13. General outline of an <b>if</b> conditional flow structure. Terms enclosed in parenthesis are optional . . . . .	90
2.14. Esquema general de la estructura de un bucle <b>for</b> los términos escritos entre paréntesis son opcionales. . . . .	94
2.14. <b>for</b> structure General outline. Terms enclosed in parentheses are optional. . . . .	94
2.15. Esquema general de la estructura de un bucle <b>while</b> los términos escritos entre paréntesis son opcionales. . . . .	101
2.15. General structure for a while loop. Terms enclosed in parentheses are optional. . . . .	101
2.16. Símbolos empleados en diagramas de flujo . . . . .	106

2.16. Symbols used in flowcharts . . . . .	106
2.17. Diagrama de flujo para el problema de los números primos . . . . .	108
2.17. Flow chart for the prime numbers problem . . . . .	108
 3.1. Representación gráfica de vectores en el plano . . . . .	125
3.1. Graphic of vectors on the plane . . . . .	125
3.2. Representación gráfica de vectores en el espacio 3D . . . . .	126
3.2. Graphic of vector in the 3D space . . . . .	126
3.3. interpretación geométrica de la norma de un vector . . . . .	144
3.3. Geometrical interpretation of the norm of a vector. . . . .	144
3.4. efecto del producto de un escalar por un vector . . . . .	156
3.4. effect of the product of a vector by a scalar . . . . .	156
3.5. Representación gráfica de los vectores $a = (1, 2)$ , $b = (-1, 1)$ y algunos vectores, combinación lineal de $a$ y $b$ . . . . .	157
3.5. A graphic representations of Vectors $a = (1, 2)$ , $b = (-1, 1)$ and some vectors obtained from a lineal combination of $a$ and $b$ . . . . .	157
3.6. Representación gráfica de los vectores $a = (1, -2, 1)$ , $b = (2, 0, -1)$ , $c = (-1, 1, 1)$ y del vector $a - b + c$ . . . . .	158
3.6. Vectors $a = (1, -2, 1)$ , $b = (2, 0, -1)$ ), $c = (-1, 1, 1)$ and vector $a - b + c$ ; a graphic representation . . . . .	158
3.7. Representación gráfica del vector $a$ , en la base canónica $\mathcal{C}$ y en la base $\mathcal{B}$ . . . . .	161
3.7. Graphic representation of vector $a$ , in the canonical basis $\mathcal{C}$ and in $\mathcal{B}$ basis. . . . .	161
3.8. Transformaciones lineales del vector $a = [1, 2]$ . $D$ , dilatación/contracción en un factor 1.5/0.5. $R_x$ , reflexión respecto al eje x. $R_y$ , reflexión respecto al eje y. $R_\theta$ rotaciones respecto al origen para ángulos $\theta = \pi/6$ y $\theta = \pi/3$ . . . . .	164
3.8. Linear transformation of vector = [1, 1]. $D$ , factor 1.5/0.5 dilation/contraction. $R_y$ reflection on y-axis. $R_\theta$ $\theta = \pi/6$ and $\theta = \pi/3$ angles rotations around the origin. . . . .	164
3.9. Formas cuadráticas asociadas a las cuatro matrices diagonales: $ a_{11}  =  a_{22}  = 1$ , $a_{12} = a_{21} = 0$ . . . . .	167
3.9. Quadratic forms obtained using the four diagonal matrices: $ a_{11}  =  a_{22}  = 1$ , $a_{12} = a_{21} = 0$ . . . . .	167
 4.1. Ejemplo sencillo de pyplot. . . . .	189
4.1. Simple pyplot example. . . . .	189
4.2. Ejemplos de dibujos con diferente formato . . . . .	191
4.2. Different uses of plot . . . . .	191
4.3. Dos gráficas en una misma figura usando subplot . . . . .	192
4.3. Two graphs in one figure using subplot . . . . .	192
4.4. Figura con texto en etiquetas . . . . .	193
4.4. Figure with text and labels . . . . .	193
4.5. Gráfica de la función $y = e^x$ con ejes en escala lineal y logarítmica . . . . .	195
4.5. Graph of the function $y = e^x$ with axes on linear and logarithmic scale . . . . .	195
4.6. Espiral en coordenadas polares . . . . .	196
4.6. Spiral plot using polar coordinates . . . . .	196
4.7. Histogramas del número de automóviles por cada 1000 habitantes . . . . .	198
4.7. . . . .	198
4.8. Representación en 3 dimensiones usando scatter . . . . .	199
4.8. 3D plot using scatter . . . . .	199
4.9. Retícula plana . . . . .	201
4.9. Flat grid . . . . .	201

4.10. Comparación entre <code>wireframe</code> y <code>surface</code> . . . . .	202
4.10. Comparison between <code>wireframe</code> and <code>surface</code> . . . . .	202
4.11. vista de la figura con los ocho subplots . . . . .	208
 5.1. Posición relativa de un número no máquina $x$ y su redondeo a número máquina por truncamiento $x_T$ y por exceso $x_E$ . Si redondeamos al más próximo de los dos, el error es siempre menor o igual a la mitad del intervalo $x_E - x_T$ . . . . .	229
5.1. Location of a non-machine number in relation to its rounding to a machine number through truncation ( $x_T$ ) and excess ( $x_E$ ). When we round the number to the nearest of the two, the error is always less than half the interval $x_E - x_T$ . . . . .	229
5.2. Ilustración del cambio de precisión con la magnitud de los números representados.	232
5.2. Graphic illustration of precision change with the magnitude of represented numbers	232
5.3. Números representables y desbordamientos en el estándar IEEE 754 de precisión simple. . . . .	233
5.3. Representable numbers and over/under-flows in the IEEE 754 simple precision standard. . . . .	233
 6.1. Ejemplo de ecuación de Kepler para $a = 40$ y $b = 2$ . . . . .	246
6.1. An Example of Kepler's equation is taking $a = 40$ and $b = 2$ . . . . .	246
6.2. Ilustración del teorema de Bolzano . . . . .	248
6.2. Bolzano's theorem. . . . .	248
6.3. Diagrama de flujo del método de la bisección . . . . .	249
6.4. Flowchart of the bisection method . . . . .	250
6.5. proceso de obtención de la raíz de una función por el método de la bisección. . . . .	251
6.5. process of obtaining the root of a function by the bisection method. . . . .	251
6.6. Obtención de la recta que une los extremos de un intervalo $[a, b]$ que contiene una raíz de la función . . . . .	253
6.6. Obtaining the line joining the extremes of an interval $[a, b]$ containing a root of the function . . . . .	253
6.8. Flowchart of false position method . . . . .	254
6.7. Diagrama de flujo del método de interpolación lineal . . . . .	254
6.9. Proceso de obtención de la raíz de una función por el método de interpolación lineal	256
6.9. Process of obtaining the root of a function by the method of linear interpolation .	256
6.10. Recta tangente a la función $f(x)$ en el punto $x_0$ . . . . .	257
6.10. Tangent line to the function $f(x)$ at the point $x_0$ . . . . .	257
6.12. Flowchart of Newton-Raphson method . . . . .	258
6.11. Diagrama de flujo del método de Newton-Raphson . . . . .	258
6.13. Proceso de obtención de la raíz de una función por el método de Newton . . . . .	260
6.13. Process of obtaining the root of a function by Newton's method . . . . .	260
6.14. Recta secante a la función $f(x)$ en los puntos $x_0$ y $x_1$ . . . . .	261
6.14. Secant line to the function $f(x)$ at points $x_0$ and $x_1$ . . . . .	261
6.16. Flowchart of the secant method . . . . .	262
6.15. Diagrama de flujo del método de la secante . . . . .	262
6.17. proceso de obtención de la raíz de una función por el método de la secante . . . . .	263
6.18. Obtención gráfica del punto fijo de la función, $g(x) = -\sqrt{e^x}$ . . . . .	265
6.18. Obtaining the fixed point of the function graphically, $g(x) = -\sqrt{e^x}$ . . . . .	265
6.19. Diagrama de flujo del método del punto fijo. Nótese que la raíz obtenida corresponde a la función $f(x) = g(x) - x$ . . . . .	269
6.19. Flowchart of fixed point iteration. Root corresponds to $f(x) = g(x) - x$ . . . . .	269
6.20. $g(x) = \pm\sqrt{e^x}$ , Solo la rama negativa tiene un punto fijo. . . . .	270

6.20. $g(x) = \pm\sqrt{e^x}$ , only negative branch has a fixed point. . . . .	270
6.21. proceso de obtención de la raíz de la función $f(x) = e^x - x^2$ aplicando el método del punto fijo sobre la función $g(x) = -\sqrt{e^x}$ . . . . .	286
6.21. Fixed point iteration to compute the root of $f(x) = e^x - x^2$ using the function $g(x) = -\sqrt{e^x}$ . . . . .	286
6.22. primeras iteraciones de la obtención de la raíz de la función $f(x) = e^x - x^2$ aplicando el método del punto fijo sobre la función $g(x) = \ln(x^2)$ . . . . .	287
6.22. first iterations of obtaining the root of the function $f(x) = e^x - x^2$ by applying the fixed point method on the function $g(x) = \ln(x^2)$ . . . . .	287
6.23. proceso de obtención de la raíz de la función $f(x) = e^x - x^2$ aplicando el método del punto fijo sobre la función $g(x) = \ln(x^2)$ , el método oscila sin converger a la solución. . . . .	288
6.23. Fixed point iteration to compute the root of function $f(x) = e^x - x^2$ using the function $g(x) = \ln(x^2)$ , iteration oscillates without converging to the solution. . . . .	288
6.24. proceso de obtención de la raíz de la función $f(x) = e^x - x^2$ aplicando el método del punto fijo sobre la función $g(x) = \frac{e^x}{x}$ , el método diverge rápidamente. . . . .	289
6.24. fixed point iteration to compute the root of function $f(x) = e^x - x^2$ using the function $g(x) = \frac{e^x}{x}$ , the iteration diverges quickly. . . . .	289
6.25. Polinomio dibujado usando <code>linspace</code> . . . . .	290
6.25. Plotting a polynomial using <code>linspace</code> . . . . .	290
6.26. Diagrama de flujo del método de la bisección con precisión máxima . . . . .	290
7.1. Sistema de ecuaciones con solución única . . . . .	294
7.1. Equation system with unique solution . . . . .	294
7.2. Sistema de ecuaciones sin solución . . . . .	296
7.2. Equation system without solution . . . . .	296
7.3. Sistema de ecuaciones con infinitas soluciones . . . . .	297
7.3. Equation system with infinite solutions . . . . .	297
7.4. Diagrama de flujo general de los métodos iterativos para resolver sistemas de ecuaciones. La función $M(x)$ es la que especifica en cada caso el método. . . . .	321
7.4. General flowchart of iterative methods for solving systems of equations. The function $M(x)$ is the one that specifies the method in each case. . . . .	321
7.5. evolución de la tolerancia (módulo de la diferencia entre dos soluciones sucesivas) para un mismo sistema resuelto mediante el método de Gauss-Seidel y el método de Jacobi . . . . .	332
7.5. evolution of the tolerance (modulus of the difference between two successive solutions) for the same system solved by the Gauss-Seidel method and the Jacobi method	332
7.6. Evolución de la tolerancia para un mismo sistema empleando el metodo de Jacobi (diverge) y el de Jacobi amortiguado (converge). . . . .	339
7.6. Tolerance evolution for the same system using the Jacobi method (diverges) and the weighted Jacobi method (converges). . . . .	339
8.1. Comparación entre resultados obtenidos para polinomios de Taylor del logaritmo natural. (grados 2, 3, 5, 10, 20) . . . . .	349
8.1. A comparison among the results achieved using Taylor polynomials to approach the logarithm. (2, 3, 5, 10, 20 degrees) . . . . .	349
8.2. Polinomios de Taylor para las funciones coseno y seno . . . . .	351
8.2. Taylor polynomial for cosine and sine functions . . . . .	351
8.3. Polinomio de interpolación de grado nueve obtenido a partir de un conjunto de diez datos . . . . .	363
8.3. Nine degree interpolating polynomial obtained using a set of ten data . . . . .	363

8.4. Interpolaciones de orden cero y lineal para los datos de la figura 8.3 . . . . .	364
8.4. Zero-order and linear interpolation for the figure data . . . . .	364
8.5. Interpolación mediante spline cúbico de los datos de la figura . . . . .	372
8.5. Cubic Spline interpolation for the data represented in the figure . . . . .	372
8.6. Polinomio de mínimos cuadrados de grado 0 . . . . .	376
8.6. 0-degree least squared error polynomial . . . . .	376
8.7. Ejemplo de cálculo de ajuste por mínimos cuadrados empleando la función fit de Polynomial y nuestra función lse. . . . .	382
8.7. An example of least squares polynomial fit, using the function fit from Polynomial and our function lse. . . . .	382
8.8. Comparación entre los residuos obtenidos para los ajustes de mínimos cuadrados de un conjunto de datos empleando polinomios de grados 1 a 4. . . . .	384
8.8. Comparison among the residuals resulting from a dataset least squares fitting, using polynomials of degrees 1 to 4. . . . .	384
8.9. Curvas de Bézier trazadas entre los puntos $P_0 = (0, 0)$ y $P_n = (2, 1)$ , variando el número y posición de los puntos de control. . . . .	387
8.9. Bézier curves traced between points $P_0 = (0, 0)$ y $P_n = (2, 1)$ , variating the control points number and positions . . . . .	387
8.10. Curvas de Bézier equivalentes, construidas a partir de una curva con tres puntos de control. . . . .	392
8.10. Equivalent curves, built from a three control point Bézier's curve. . . . .	392
8.11. Curva de Bézier y su derivada con respecto al parámetro del polinomio de Bernstein que la define: $t \in [0, 1]$ . . . . .	393
8.11. Bézier's curve and its derivative with respect to the Bernstein's polynomial parameter which defines the curve $t \in [0, 1]$ . . . . .	393
8.12. Interpolación de tres puntos mediante dos curvas de Bézier . . . . .	395
8.12. interpolating three point using two Bezier's curves . . . . .	395
9.1. Variación del error cometido al aproximar la derivada de una función empleando una fórmula de diferenciación de dos puntos. . . . .	406
9.1. Function derivative approximation error, using a a two-points differentiation formula. Dependency on the difference $h$ value . . . . .	406
9.2. Comparación entre las aproximaciones a la derivada de una función obtenidas mediante las diferencias de dos puntos adelantada y centrada. . . . .	407
9.2. Comparison between the approximations achieved for a function derivative using two point forward difference and two point central difference. . . . .	407
9.3. Interpretación gráfica de la fórmula del trapecio. . . . .	411
9.3. Graphical interpretation of trapezium rule. . . . .	411
9.4. Interpretación gráfica de la fórmula extendida del trapecio. . . . .	412
9.4. Extended trapezium formula graphical interpretation . . . . .	412
9.5. Interpretación gráfica de la fórmula 1/3 de Simpson. . . . .	413
9.5. Graphical interpretation of Simpson's 1/3 formula . . . . .	413
9.6. Interpretación gráfica de la fórmula 3/8 de Simpson. . . . .	414
9.6. Graphical interpretation of Simpson's 3/8 formula . . . . .	414
9.7. Circuito RC . . . . .	423
9.8. Comparacion entre los resultados obtenidos mediante el método de Euler para dos pasos de integración $h = 0.5$ y $h = 0.25$ y la solución analítica para el voltaje $V_o$ de un condensador durante su carga. Hemos tomado $C = R = 1$ y $V_i = 10$ . . . . .	424

9.8. Comparison among the results obtained using Euler's method with two different integration steps $h = 0.5$ y $h = 0.25$ , and the analytic solution for voltage $V_o$ across a capacitor. We took $C = R = 1$ and $V_i = 10$ . . . . .	424
9.9. Masa suspendida en vertical de un muelle con rozamiento. . . . .	425
9.9. A mass vertically hanging of a sprint with a damping resistance . . . . .	425
9.10. Resultado de aplicar el método de Euler a una masa suspendida en vertical de un muelle con rozamiento. . . . .	427
9.10. Results achieved using Euler's method to solve the damped oscillator problem. . . .	427
9.11. Resultado de aplicar el método de Euler a una masa suspendida en vertical de un muelle con rozamiento $k = 100$ . . . . .	428
9.11. Result achieved using Euler's method to solve the damped oscillator problem. $K = 100$	428
9.12. Resultado de aplicar el método de Euler a una masa suspendida en vertical de un muelle con rozamiento $k = 100$ y paso $h = 0.001$ . . . . .	429
9.12. Result achieved using Euler's method to solve the damped oscillator problem. $K = 100$ , $h = 0.001$ . . . . .	429
9.13. Sistema masa-muelle-plano inclinado. . . . .	436
9.13. Mass-spring-inclined plane system. . . . .	436

# Índice de cuadros

2.1.	Algunos métodos de las Listas en Python . . . . .	51
2.1.	Some methods of Python's List . . . . .	51
2.2.	Operadores aritméticos definidos en Python . . . . .	55
2.2.	Arithmetic operators defined in python . . . . .	55
2.3.	Operadores relacionales definidos en Python . . . . .	59
2.3.	Relational operators defined in Python . . . . .	59
2.4.	Operadores lógicos entre valores y variables . . . . .	60
2.4.	Logical operators on values and variables . . . . .	60
3.1.	Algunas funciones matemáticas en Numpy de uso frecuente . . . . .	154
3.1.	frequently used mathematical functions included in Numpy . . . . .	154
4.1.	Colores de los puntos . . . . .	189
4.1.	Point colors . . . . .	189
4.2.	Algunos de los marcadores de puntos . . . . .	190
4.3.	Tipos de línea . . . . .	190
4.2.	Some point markers . . . . .	190
4.3.	Line types . . . . .	190
5.1.	Representación en complemento a dos para un registro de 4 bits . . . . .	213
5.1.	2's complement representation for a 4-bit register . . . . .	213
5.2.	Representación <i>en exceso a 127</i> , para un exponente de 8 bits. . . . .	221
5.2.	127- <i>biased</i> representation for a 8-bit exponent . . . . .	221
5.3.	Comparación entre los estándares del IEEE para la representación en punto flotante. ( $b_s$ bit de signo, $m_i$ bit de mantisa, $e_i$ bit de exponente) . . . . .	225
5.3.	Comparison between the IEEE standards for floating point representation. ( $b_s$ sign bit, $m_i$ mantissa bit, $e_i$ exponent bit) . . . . .	225
8.1.	$f(x) = \operatorname{erf}(x)$ . . . . .	346
8.2.	Tabla de diferencia divididas para cuatro datos . . . . .	357
8.2.	four data divided difference table . . . . .	357
8.3.	Tabla de diferencias para el polinomio de Newton-Gregory de cuatro datos . . . . .	361
8.3.	Table of differences for a Newton-Gregory polynomial of four data . . . . .	361
9.1.	Fórmulas de derivación basadas en diferencias finitas . . . . .	408
9.1.	Derivative formulae based on finite differences . . . . .	408



# Preface

# Prefacio

'Fair Lady!' Said Frodo again after a while. 'Tell me, if my asking does not seem foolish, who is Tom Bombadil?'

'He is' said Goldberry, staying her swift movements and smiling. (...) 'he is as you have seen him'. He is the Master of wood, water and hill.

'Then all this strange land belongs to him?'

'No indeed!' she answered, and her smile faded. 'That would indeed be a burden,' she added in a low voice, as if to herself. 'The trees and the grasses and all things growing or living in the land belong each to themselves. Tom Bombadil is the Master.'

---

J.R.R Tolkien, The Lord of the Rings.

Estos apuntes cubren de forma aproximada el contenido del *Laboratorio de computación científica* del primer curso del grado en física. La idea de esta asignatura es introducir al estudiante a las estructuras elementales de programación y al cálculo numérico, como herramientas imprescindibles para el trabajo de investigación.

Los capítulos uno al cuatro ofrecen una introducción a la programación en python, incluyendo algunas de las librerías más empleadas en computación científica. El resto de los capítulos describen técnicas clásicas para la resolución de problemas numéricos.

These lecture notes cover the contents of the *scientific computing lab*: a first course in scientific computing teaching during the first semester of the degree in physics. The aim is to introduce the student to computer programming and numerical calculus, which are invaluable tools in scientific research.

Chapters one to four present an introduction to Python programming, including some of the most common used libraries for scientific computing. The remaining chapters describe classical techniques to solve numerical problems.

Los párrafos encerrados entre 'brujas'



contienen material avanzado, pueden saltarse y son para aquellos que quieran ampliar conocimientos.

Casi todos los métodos que se describen en estos apuntes fueron desarrollados hace siglos por los grandes: Newton, Gauss, Lagrange, etc. Métodos que no han perdido su utilidad y que, con el advenimiento de los computadores digitales, han ganado todavía más si cabe en atractivo e interés. Se cumple una vez más la famosa frase atribuida a Bernardo de Chartres:

“Somos como enanos a los hombres de gigantes. Podemos ver más, y más lejos que ellos, no por que nuestra vista sea más aguda, sino porque somos levantados sobre su gran altura.”

En cuanto a los contenidos, ejemplos, código, etc. Estos apuntes deben mucho a muchas personas. En primer lugar a Manuel Prieto y Segundo Esteban que elaboraron las presentaciones de la asignatura *Introducción al cálculo científico y programación* de la antigua licenciatura en físicas, de la que el laboratorio de computación científica es heredera.

En segundo lugar a mis compañeros de los departamentos de *Física de la Tierra, Astronomía y Astrofísica I y Arquitectura de computadores y Automática* que han impartido la asignatura durante estos años:

Rosa González Barras, Belén Rodríguez Fonseca, Maurizio Matessini, Pablo Zurita, Vicente Carlos Ruiz Martínez, Encarna Serrano, Carlos García Sánchez, Jose Antonio Martín, Victoria López López, Alberto del Barrio, Blanca Ayarzagüena, Javier Gómez Selles, Nacho Gómez Pérez, Marta Ávalos, Iñaqui Hidalgo, Daviz sánchez, Juan Rodriguez, María Ramírez, Álvaro de la Cámara, Marta Martín (Espero no haberme olvidado de nadie). Muchas gracias a todos por tantas horas de trabajo compartidas

Paragraphs enclosed in 'witches',



contain advanced material. They are optional for those interested in deepening their knowledge.

Almost every method described in these notes was developed, centuries ago, by the *big ones*: Newton, Gauss, Lagrange, etc. But they are methods that are still useful and, with the coming up of digital computers, they are more interesting than ever. We can indeed quote the famous sentence from The scholar Bernardo de Chartres:

“We are like dwarfs sitting on the shoulders of giants. We see more, and things that are more distant, than they did, not because our sight is superior or because we are taller than they, but because they raise us up, and by their great stature add to ours.”

The contents, examples, code, etc. of this notes, are the results of the effort of many people. First, I would like to mention Manuel Prieto and Segundo Esteban, which prepared the slides for *Introducción al Cálculo Científico y Programación*, the predecessor of the Scientific Computing Laboratory in the old degree of Physics.

Second, I also want to thank to my colleagues from *Física de la Tierra, Astronomía y Astrofísica I* and *Arquitectura de computadores y Automática* Who have taught the subject since the beginning :

Rosa González Barras, Belén Rodríguez Fonseca, Maurizio Matessini, Pablo Zurita, Vicente Carlos Ruiz Martínez, Encarna Serrano, Carlos García Sánchez, Jose Antonio Martín, Victoria López López, Alberto del Barrio, Blanca Ayarzagüena, Javier Gómez Selles, Nacho Gómez Pérez, Marta Ávalos, Iñaqui Hidalgo, Daviz sánchez, Juan Rodriguez, María Ramírez, Álvaro de la Cámara, Marta Martín (I hope don't forget anybody). Thank you very much for sharing so many work hours.

Juan Jiménez, Lia García y Héctor García de Marina.



# Capítulo/Chapter 1

## Introducción al software científico Introduction to scientific software

”Begin at the beginning”, the King said, gravely, ”and go on till you came to the end; then stop”

---

Lewis Carroll, Alice in Wonderland

En la actualidad, el ordenador se ha convertido en una herramienta imprescindible para el trabajo de cualquier investigador científico. Su uso ha permitido realizar tareas que sin su ayuda resultarían sencillamente imposibles de acometer. Entre otras, distinguiremos las tres siguientes:

- Adquisición de datos de dispositivos experimentales.
- Análisis y tratamiento de datos experimentales.
- Cálculo Científico.

La primera de éstas tareas queda fuera de los contenidos de esta asignatura. Su objetivo es emplear el ordenador para recoger datos automáticamente de los sensores empleados en un dispositivo experimental. El procedimiento habitual es emplear dispositivos electrónicos que traducen las lecturas de un sensor (un termómetro, un manómetro, un caudalímetro, una cámara etc.) a un voltaje. El voltaje es digitalizado, es decir, convertido a una secuencia de ceros y unos, y almacenado en un ordenador para su posterior análisis o/y direc-

Computers have become an essential tool in the daily work of every Scientific researcher. They are used for doing tasks that could not be carried out without their help. We can point out the following tasks, among others:

- Data acquisition from experimental devices.
- Experimental data analysis and processing.
- Scientific computing.

The first of these tasks is beyond the scope of this course. It aims to use the computer to get data automatically from the sensors attached to an experimental device. Usually, data supplied by a sensor (a thermometer, a manometer, a flow meter, an optical Camera, etc.) are converted to voltages by some electronic device. Then, the voltages are digitized —i.e., converted to a sequence of zeros and ones— and stored in a computer for later analysis. Also, they can be shown (monitored) on a computer screen. In many cases, the computer can also interact with the experimental device: start or stop an experiment, control

tamente monitorizado, es decir, mostrado en la pantalla del ordenador. En muchos casos el ordenador es a su vez capaz de interactuar con el dispositivo experimental: iniciar o detener un experimento, regular las condiciones en que se realiza, disparar alarmas si se producen errores, etc.

De este modo, el investigador científico, queda dispensado de la tarea de adquirir por sí mismo los datos experimentales. Tarea que en algunos casos resultaría imposible, por ejemplo si necesita medir muchas variables a la vez o si debe medirlas a gran ritmo; y en la que, en general, es relativamente fácil cometer errores.

El análisis y tratamiento de datos experimentales, constituye una tarea fundamental dentro del trabajo de investigación científica. Los ordenadores permiten realizar dichas tareas, de una forma eficiente y segura con cantidades de datos que resultarían imposibles de manejar hace 50 años. Como veremos más adelante, una simple hoja de cálculo puede ahorrarnos una cuantas horas de cálculos tediosos. El análisis estadístico de un conjunto de datos experimentales, el cálculo –la estimación– de los errores experimentales cometidos, la posterior regresión de los datos obtenidos a una función matemática que permita establecer una ley o al menos una relación entre los datos obtenidos, formar parte del trabajo cotidiano del investigador, virtualmente en todos los campos de la ciencia.

Por último el cálculo. Cabría decir que constituye el núcleo del trabajo de investigación. El científico trata de explicar la realidad que le rodea, mediante el empleo de una descripción matemática. Dicha descripción suele tomar la forma de un modelo matemático más o menos complejo. La validez de un modelo está ligada a que sea capaz de reproducir los resultados experimentales obtenidos del fenómeno que pretende explicar. Si el modelo es bueno será capaz de obtener mediante cálculo unos resultados similares a los obtenido mediante el experimento. De este modo, el modelo queda validado y es posible emplearlo para predecir cómo se comportará el sistema objeto de estudio en otras condiciones.

the experimental conditions, trigger an alarm in case of error, etc.

In this way, the scientific researcher is released from getting the experimental data by himself. A task that could sometimes be impossible to do. For instance, when he needs to measure many variables simultaneously or when the measurements must be taken quickly. Besides, it is easy to make mistakes when the measurements are manually taken.

Experimental data analysis and processing are fundamental tasks in scientific work. The computers carry out such tasks efficiently and safely, working with data amounts that were impossible to deal with fifty years ago. As we shall see later, a simple data sheet can save us many hours of tedious calculations. Statistical analysis of experimental data, Estimation of experimental errors, and regression of data to a mathematical function allows us to establish a law or at least find a relationship among the data. All these are part of researchers' daily work, virtually in any field of science.

Lastly, computing . It can be said that scientific computing is the kernel of scientific work. The researcher tries to explain the real world using a mathematical description. Such a description usually takes the form of a mathematical model, which can be more or less complex. A model is valid because it can reproduce the same results as the original experiment, which the model tries to explain. If the model is good enough, it can be obtained by computing similar results from the experiment. Then, the model becomes tested and can be used to forecast the system's behaviour under study in many different conditions.

## 1.1. Introducción a los computadores.

Más o menos todos estamos familiarizados con lo que es un computador, los encontramos a diario continuamente y, de hecho, hay muchos aspectos de nuestra vida actual que serían inimaginables sin los computadores. En términos muy generales, podemos definir un computador como una máquina que es capaz de recibir instrucciones y realizar operaciones (cálculos) a partir de las instrucciones recibidas. Precisamente es la capacidad de recibir instrucciones lo que hace del ordenador una herramienta versátil; según las instrucciones recibidas y de acuerdo también a sus posibilidades como máquina, el ordenador puede realizar tareas muy distintas, entre las que cabe destacar como más generales, las siguientes:

- Procesamiento de datos
- Almacenamiento de datos
- Transferencias de datos entre el computador y el exterior
- Control de las anteriores operaciones

El computador se diseña para realizar funciones generales que se especifican cuando se programa. La programación es la que concreta las tareas que efectivamente realiza un ordenador concreto.

### 1.1.1. Niveles de descripción de un ordenador.

La figura 1.1 muestra un modelo general de un computador descrito por niveles. Cada nivel, supone y se apoya en el nivel anterior.

1. **Nivel Físico.** Constituye la base del *hardware* del computador. Está constituido por los componentes electrónicos básicos, diodos, transistores, resistencias, etc. En un computador moderno, no es posible separar o tan siquiera observar dichos componentes: Se han fabricado directamente sobre un cristal semiconductor, y forman parte de un dispositi-

## 1.1. Introduction to computers.

Everybody is somehow familiar with computers. We find them daily, and we depend on them in such a way that our current lives would be unimaginable without them. In general, a computer can be defined as a machine able to get instructions and data and use the instructions to perform calculations with the data. The capacity to get instruction makes the computer a versatile tool; according to the instruction received and the specific machine capacity, the computer can carry out very different tasks. Among them, we can highlight the following:

- data processing
- data storing
- data transfer between the computer and the outside.
- Control of these operations just listed.

The computer is designed to perform general functions which are specified when the computer is programmed. Programming is the way to define the tasks the computer will carry out.

Figure 1.1 shows a general computer layout described by levels. Each level lays and assume the previous level.

### 1.1.1. Computer description levels.

1. **Physical Level.** It's the ground of the computer hardware. Basic electronic components, diodes, transistors, resistances, etc, make it up. In a modern computer, it is impossible to split or even to watch such components: they are built directly in a semiconductor Crystal and part of an electronic device Known as *integrated circuit*.

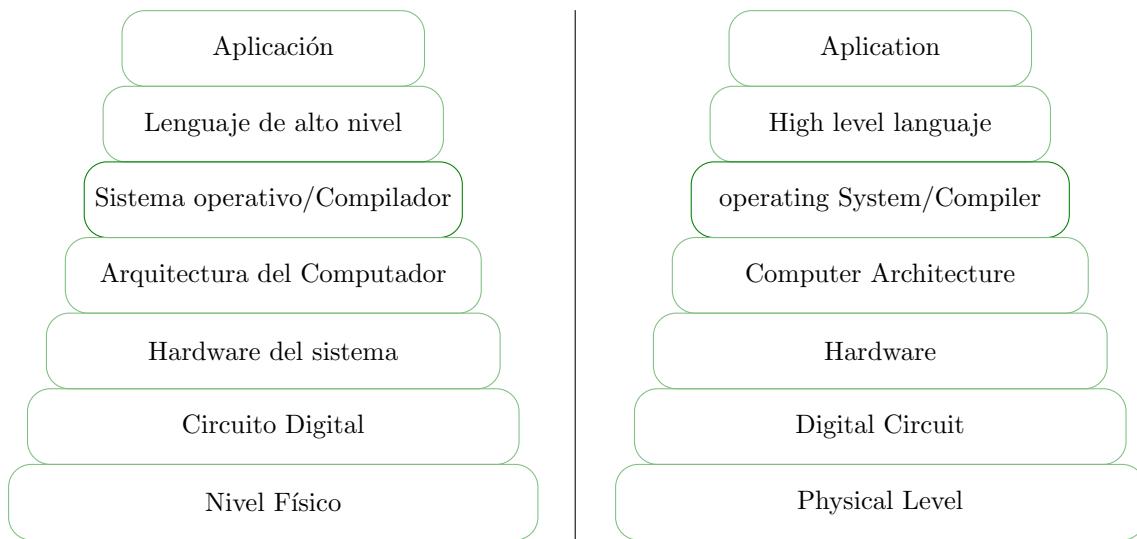


Figura 1.1: Descripción por niveles de un computador

vo electrónico conocido con el nombre de circuito integrado.

2. **Círculo Digital.** Los componentes del nivel físico se agrupan formando circuitos digitales, (En nuestro caso circuitos digitales integrados). Los circuitos digitales trabajan solo con dos niveles de tensión ( $V_1, V_0$ ) lo que permite emplearlos para establecer relaciones lógicas:  $V_1=\text{verdadero}$ ,  $V_2=\text{falso}$ . Estas relaciones lógicas establecidas empleando los valores de la tensión de los circuitos digitales constituyen el soporte de todos los cálculos que el computador puede realizar.
3. **Organización Hardware del sistema.** Los circuitos digitales integrados se agrupan y organizan para formar el *Hardware* del ordenador. Los módulos básicos que constituyen el *Hardware* son la unidad central de procesos (CPU), La unidad de memoria y las unidades de entrada y salida de datos. Dichos componentes están conectados entre sí mediante un bus, que transfiere datos de una unidad a otra.
4. **Arquitectura del computador.** La ar-

Figure 1.1: Computer level description

2. **Digital Circuit.** Physical level components are grouped, forming digital circuits. (In our case, Integrated digital circuits). They work with only two voltage levels ( $V_1, V_0$ ). This allows us to use them for defining logical relationships:  $V_1=\text{true}$ ,  $V_2=\text{false}$ . This logical relationship obtained using two voltage levels of digital circuits is, in turn, the basis for every computing the computer can carry out.
3. **Computer Hardware organisation.** Digital circuits are grouped and organized to build up the computer Hardware. Basic hardware modules are the central processing unit(CPU), the memory unit and the input and output units. These components are connected by a bus, which transfers data from one unit to another.
4. **Computer Architecture.** The architecture defines how the computer works. Thus, it is highly related to the hardware organization of the system; bit architecture works at a higher abstraction level. It defines how to get access to the memory register, arbitrate the use of the buses with links to the different compo-

arquitectura define cómo trabaja el computador. Por tanto, está estrechamente relacionada con la organización hardware del sistema, pero opera a un nivel de abstracción superior. Establece cómo se accede a los registros de memoria, arbitra el uso de los buses que comunican unos componentes con otros, y regula el trabajo de la CPU.

Sobre la arquitectura se establece el lenguaje básico en el que trabaja el ordenador, conocido como lenguaje máquina. Es un lenguaje que emplea todavía niveles lógicos binarios (ceros o unos) y por tanto no demasiado apto para ser interpretado por los seres humanos. Este lenguaje permite al ordenador realizar operaciones básicas como copiar el contenido de un registro de memoria en otro, sumar el contenido de dos registros de memoria, etc.

El lenguaje máquina es adecuado para los computadores, pero no para los humanos, por eso, los fabricantes suministran junto con el computador un repertorio básico de instrucciones que su máquina puede entender y realizar en un lenguaje algo más asequible. Se trata del lenguaje ensamblador. Los comandos de este lenguaje son fácilmente traducibles en una o varias instrucciones de lenguaje máquina. Aún así se trata de un lenguaje en el que programar directamente resulta una tarea tediosa y propensa a cometer errores.

5. **Compiladores y Sistemas Operativos** Los Compiladores constituyen un tipo de programas especiales que permiten convertir un conjunto de instrucciones, escritas en un lenguaje de alto nivel en lenguaje máquina. El programador escribe sus instrucciones en un fichero de texto normal, perfectamente legible para el ser humano, y el compilador convierte las instrucciones contenidas en dicho fichero en secuencias binarias comprensibles por la máquina.

Los computadores primitivos solo eran

nents and regulates the CPU work.

The primary language the computer works with is established according to its architecture. This primary language is called Machine language. It still uses binary logical levels (zeros and ones), so it is unsuitable for being understood by human beings. The computer uses machine language to perform basic operations such as copying the content from one memory register to another, adding the contents of two memory registers, etc.

Machine language is suitable for computers but not for human beings. Thus, manufacturers supply the computer with a basic repertory of instructions that their machine can understand and carry out, written in a more accessible language. It is known as assembler language. The commands of this language are accessible to translate to one or several machine language instructions. Writing code right in assembler language is tedious and prone to mistakes.

5. **Compilers y operating Systems** Compilers are special programs that allow us to translate instructions written in a high-level language into machine language. The programmer writes instructions in plain text, readable for a human being. Then, the compiler translates the file's contents into binary sentences that the machine can interpret.

Early computers were only able to run a program at a time. As new and more sophisticated computers were built, it arises the idea of making programs which were able to take over the basic tasks: Managing the information flow, dealing with peripheral devices, etc. These programs are called operating systems. Modern computers load an operating system at the boot time, which controls the running of the remaining programs. Some examples of operating systems are DOS (Disk Operating System), UNIX and the UNIX version for personal computers LINUX.

capaces de ejecutar un programa a la vez. A medida que se fueron fabricando ordenadores más sofisticados, surgió la idea de crear programas que se encargaran de las tareas básicas: gestionar el flujo de información, manejar periféricos, etc. Estos programas reciben el nombre de sistemas operativos. Los computadores modernos cargan al arrancar un sistema operativo que controla la ejecución del resto de las aplicaciones. Ejemplos de sistemas operativos son DOS (Disk Operating System), Unix y su versión para ordenadores personales Linux.

6. **Lenguajes de alto nivel.** Los lenguajes de alto nivel están pensados para facilitar la tarea del programador, desentendiéndose de los detalles de implementación del hardware del ordenador. Están compuestos por un conjunto de comandos y unas reglas sintácticas, que permiten describir las instrucciones para el computador en forma de texto.

De una manera muy general, se pueden dividir los lenguajes de alto nivel en lenguajes compilados y lenguajes interpretados. Los lenguajes compilados emplean un compilador para convertir los comandos del lenguaje de alto nivel en lenguaje máquina. Ejemplos de lenguajes compilados son C, C++ y Fortran. Los lenguajes interpretados a diferencia de los anteriores no se traducen a lenguaje máquina antes de ejecutarse. Si no que utilizan otro programa –el intérprete– que va leyendo los comandos del lenguaje y convirtiéndolos en instrucciones máquina a la vez que el programa se va ejecutando. Ejemplos de programas interpretado son Basic, Python y Java.

7. **Aplicaciones.** Se suele entender por aplicaciones programas orientados a tareas específicas, disponibles para un usuario final. Habitualmente se trata de programas escritos en un lenguaje de alto nivel y presentados en un formato fácilmente comprensible para quien los usa.

Existen multitud de aplicaciones, entre

6. **High level languages** High-level languages are intended to make the programmer's work easier, ignoring the hardware implementation details. They comprise a set of commands and syntactic rules, allowing the programmer to describe computer instruction in plain text.

Generally, High-level languages can be divided into compiled and interpreted languages. Compiled languages employ a compiler to translate the command from the high-level language to machine language. Some examples are C, C++ and FORTRAN. On the contrary, interpreted languages do not translate to machine language. They use a second program known as the interpreter. While a program in the interpreted language is running, the interpreter reads the program commands one by one and translates them to machine language. Examples of interpreted languages are BASIC, Python or Java.

7. **Programmes.** A Programme is a piece of code intended to perform specific tasks. They are available to the final users of the computer. Usually, programs are written using a high-level language and are user-friendly, i.e. they have an interface that is easy to use.

There are many different kinds of programs, according to their purpose. We can find Internet web browsers like Google Chrome, Mozilla or Bing, text editors like Word, Emacs or Latex. E-mail clients (Mail User Agents) like Outlook or Mozilla Thunderbird. The list of available programs would be endless.

las más conocidas cabe incluir los navegadores para Internet, como Bing, Mocilla o Google Crome, los editores de texto, como Word, las hojas de cálculo como Excel o los clientes de correo como Outlook. En realidad, la lista de aplicaciones disponibles en el mercado sería interminable.

### 1.1.2. El modelo de computador de Von Neumann

Los computadores modernos siguen, en líneas generales, el modelo propuesto por Von Neumann. La figura 1.2 muestra un esquema de dicho modelo.

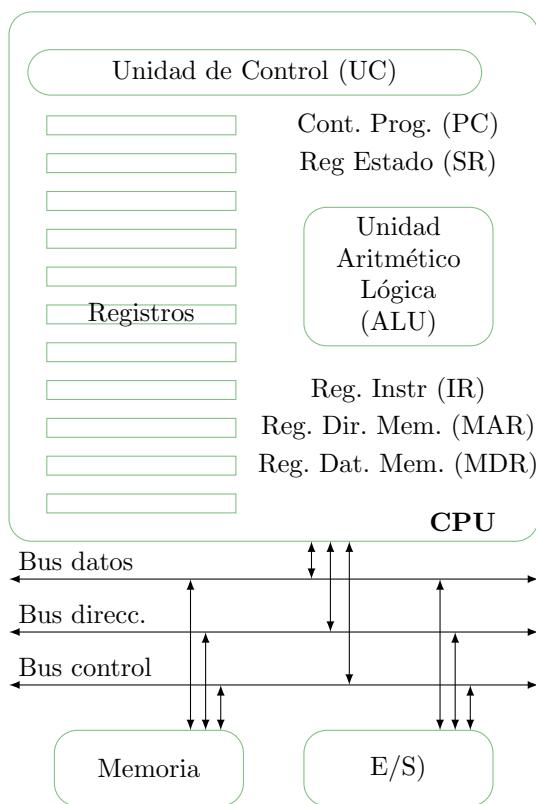


Figura 1.2: Modelo de Von Neumann

En el modelo de Von Newman se pueden distinguir tres módulos básicos y una serie de elementos de interconexión. Los módulos básicos son:

### 1.1.2. The Von Neumann's computer model

Modern computers generally follow the model proposed by Von Neumann. Figure 1.2 shows a schematic view of Von Neumann's model.

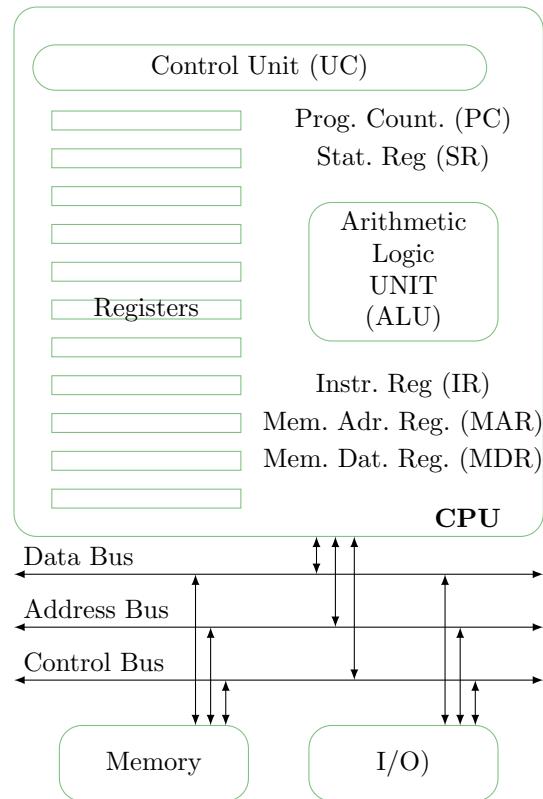


Figura 1.2: Von Neumann's model

Von Neuman's model has been divided into three basic models and several interconnection elements. The basic modules are:

- The Central Processing Unity. CPU

- **La Unidad Central de Procesos.** CPU (*Central process unit*) , esta unidad constituye el núcleo en el que el ordenador realiza las operaciones.

Dentro de la CPU pueden a su vez distinguirse las siguientes partes:

- La unidad de proceso ó ruta de datos: Está formada por La Unidad Aritmético Lógica (ALU), capaz de realizar las operaciones aritméticas y lógicas que indican las instrucciones del programa. En general las ALUs se construyen para realizar aritmética entre enteros, y realizar las operaciones lógicas básicas del algebra de Boole (AND, OR, etc). Habitualmente, las operaciones para números no enteros, representados en *punto flotante* se suelen realizar empleando un procesador específico que se conoce con el nombre de Coprocesador matemático. La velocidad de procesamiento suele medirse en millones de operaciones por segundo (MIPS) o millones de operaciones en punto flotante por segundo (MFLOPS).
- El banco de registros: Conjunto de registros en los que se almacenan los datos con los que trabaja la ALU y los resultados obtenidos.
- La unidad de control (UC) o ruta de control: se encarga de buscar las instrucciones en la memoria principal y guardarlas en el registro de instrucciones, las decodifica, las ejecuta empleando la ALU, guarda los resultados en el registro de datos, y guarda las condiciones derivadas de la operación realizada en el registro de estado. El registro de datos de memoria, contiene los datos que se están leyendo de la memoria principal o van a escribirse en la misma. El registro de direcciones de memoria, guarda la dirección de la memoria principal a las que esta accediendo la ALU,

This Unity is the kernel where the computer performs operations.

Inside the CPU, it is possible to difference the following parts:

- The processing unit or data router comprises the Arithmetic Logic Unit (ALU), . The ALU can perform the arithmetical and logical operations described in the program instructions. They are built to perform arithmetic between integer numbers and the basic Boole's algebra logical operations (AND, OR, etc). Non-integer numbers are usually represented using a unique format called *floating point representation*. Operations between floating point numbers are performed using a specific processor known as a coprocessor. The processing speed is measured in millions of instructions per second (MIPS) or millions of floating point operations per second (MFLOPS).
- The register bank: A set of registers for storing the data ALU is working with and the results of ALU operations.
- The control unit (UC) or control route: It fetches the instructions from the main memory and stores them in the instruction register. It also decoded the instructions, executed them using the ALUs, and stored the results in the data register. Once the operation is finished, the UC stores the conditions derived from the operation in the state register. The memory data register holds the data read from the main memory or those ready to be written there. Memory Address register holds the main memory address the ALU is accessing for writing or reading. The programme counter, or the instruction pointer, is a special register. It stores the current position at which the CPU is located inside a

para leer o escribir. El contador del programa, también conocido como puntero de instrucciones, es un registro que guarda la posición en la que se encuentra la CPU dentro de la secuencia de instrucciones de un programa.

- **La unidad de memoria.** Se trata de la memoria principal o primaria del computador. Está dividida en bloques de memoria que se identifican mediante una dirección. La CPU tiene acceso directo a dichos bloques de memoria.

La unidad elemental de información digital es el bit (0,1). La capacidad de almacenamiento de datos se mide en Bytes y en sus múltiplos, calculados siempre como potencias de 2:

$$\begin{aligned}1 \text{ Byte} &= 8 \text{ bits} \\1 \text{ KB} &= 2^{10} \text{ bits} = 1024 \text{ B} \\1 \text{ MB} &= 2^{20} \text{ bits} = 1024 \text{ KB} \\1 \text{ GB} &= 2^{30} \text{ bits} \\1 \text{ TB} &= 2^{40} \text{ bits}\end{aligned}$$

- **Unidad de Entrada/Salida.** Transfiere información entre el computador y los dispositivos periféricos.

Los elementos de interconexión se conocen con el nombre de *Buses*. Se pueden distinguir tres: En bus de datos, por el que se transfieren datos entre la CPU y la memoria ó la unidad de entrada/salida. El bus de direcciones, para especificar una dirección de memoria o del registro de E/S. Y el bus de Control, por el que se envían señales de control, tales como la señal de reloj, la señal de control de lectura/escrituras entre otras.

### 1.1.3. Representación binaria

Veamos con algo más de detalle, cómo representa la información un computador. Como se explicó anteriormente, La electrónica que constituye la parte física del ordenador,

program instructions sequence.

- **Memory Unit** It is the main or primary memory of the computer. It is divided into memory blocks. Each memory block is identified by its address. The CPU has direct access to memory blocks.

The elemental unit of digital information is the *bit* (0,1). Data storing capacity is gauged in Bytes and multiples of Byte, represented as powers of two:

$$\begin{aligned}1 \text{ Byte} &= 8 \text{ bits} \\1 \text{ KB} &= 2^{10} \text{ bits} = 1024 \text{ B} \\1 \text{ MB} &= 2^{20} \text{ bits} = 1024 \text{ KB} \\1 \text{ GB} &= 2^{30} \text{ bits} \\1 \text{ TB} &= 2^{40} \text{ bits}\end{aligned}$$

- **Input/Output Unit.** This unit transfers information between the computer and the peripheral devices.

The interconnection elements are called *Buses*. We can define three: the data bus transfers data between the CPU and the main memory or the input/output unity. The address bus is used for transmitting a memory address or an input/output unit. The control bus for sending control signals, such as the clock signal and the reading/writing control signal, among others.

### 1.1.3. Binary coding

Let's see in more detail how a computer represents the information. As described before, the electronic stuff represents the computer's physical part. It works with two voltage le-

trabaja con dos niveles de voltaje. Esto permite definir dos estados, –alto, bajo– que pueden representarse dos símbolos 0 y 1. Habitualmente, empleamos 10 símbolos:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, es decir, empleamos una representación decimal. Cuando queremos representar números mayores que nueve, dado que hemos agotado el número de dígitos disponibles, lo que hacemos es combinarlos, agrupando cantidades de diez en diez. Así por ejemplo, el numero 16, representa seis unidades más un grupo de diez unidades y el número 462 representa dos unidades más seis grupos de diez unidades más cuatro grupos de 10 grupos de 10 unidades. Matemáticamente, esto es equivalentes a emplear sumas de dígitos por potencias de diez:

$$13024 = 1 \times 10^4 + 3 \times 10^3 + 0 \times 10^2 + 2 \times 10^1 + 4 \times 10^0$$

Si recorremos los dígitos que componen el número de izquierda derecha, cada uno de ellos representa una potencia de diez superior, porque cada uno representa la cantidad de grupos de 10 grupos, de grupos ... de diez grupos de unidades. Esto hace que potencialmente podemos representar cantidades tan grandes como queramos, empleando tan solo diez símbolos. Esta representación, a la que estamos habituados recibe el nombre de representación en base 10 . Pero no es la única posible.

Volvamos a la representación empleada por el computador. En este caso solo tenemos dos símbolos distintos el 0 y el 1. Si queremos emplear una representación análoga a la representación en base diez, deberemos agrupar ahora las cantidad en grupos de dos. Así los únicos números que admiten ser representados con un solo dígito son el uno y el cero. Para representar el número dos, necesitamos agrupar: tendremos 0 unidades y 1 grupo de dos, con lo que la representación del número dos en base dos será 10. Para representar el número tres, tendremos una unidad más un grupo de dos, por lo que la representación será 11, y así sucesivamente. Matemáticamente esto es equivalente emplear sumas de dígitos por potencias de 2:

$$10110 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

vels, which can be associated with two states —high and low— and, in turn, these levels can define two symbols, 0 and 1. Usually, we employ 10 symbols (digits):

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, i.e. we use a *decimal* representation. Whenever we want to represent a number greater than nine, we combine several symbols, gathering quantities in groups of ten because we have exhausted the ten available digits. So, for instance, the number 16 represents six units plus a group of ten units, and the number 462 represents two units plus six groups of ten units, plus four groups of ten groups of 10 units. In mathematics, this is equal to using sums of products of digits by powers of ten:

If we look at the digits that compose the number from left to right, each one represents an upper power of ten, i.e., each represents the number of groups of ten groups of groups ... of ten groups of unities. This means we can represent amounts as significant as we wish, using only ten symbols. We are used to this numerical representation, which is known as a decimal representation or representation in base 10. But it is not the only one.

Returning to the computer's numerical representation, we have only two digits: 0 and 1. If we want to define a representation that resembles the base 10 representation, we must now gather the quantities in groups of two. So, the number two is represented in base 2 as 10. To represent the number three, we have a group of two plus one unit 11. In mathematics, this is equal to using sums of products of digits by powers of two;

Esta representación recibe el nombre de representación binaria o en base 2. La expansión de un número representado en binario en potencias de 2, nos da un método directo de obtener su representación decimal. Así, para el ejemplo anterior, si calculamos las potencias de dos y sumamos los resultados obtenemos:

$$1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 16 + 0 + 4 + 2 + 0 = 22$$

que es la representación en base 10 del número binario 10110.

Para números no enteros, la representación tanto en decimal como en binario, se extiende de modo natural empleando potencias negativas de 10 y de 2 respectivamente. Así,

This representation is known as binary coding. If we expand the binary encoding of a number as powers of two, we obtain a direct way to obtain its decimal representation. For instance, If we take the last example, calculate the power of two and add the results we obtain,

$$835.41 = 8 \times 10^2 + 3 \times 10^1 + 5 \times 10^0 + 4 \times 10^{-1} + 1 \times 10^{-2}$$

y para un número en binario,

$$101.01 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2}$$

De nuevo, basta calcular el término de la derecha de la expresión anterior para obtener la representación decimal del número 101.01.

¿Cómo transformar la representación de un número de decimal a binario? De nuevo nos da la clave la representación en sumas de productos de dígitos por potencias de dos. Empecemos por el caso de un número entero. Supongamos un número D, representado en decimal. Queremos expandirlo en una suma de potencias de dos. Si dividimos el número por 2, podríamos representarlo como:

which is the representation on base 10 for the binary number 10110

For non-integer numbers, decimal and binary representations of a number extend straightforwardly using the negative powers of 10 and 2, respectively. So,

and for a binary number,

Again, it is enough to compute the previous expression's right-side term to obtain the decimal representation for the number 101.01

How to convert a number representations from decimal to binary? One more time, the key is to look at the number representation as a digits multiplied by powers of two. Let us start with an integer number. Suppose we have a number D, represented in decimal basis. We can expand it a sum of powers of two. If we divide the number by two, we could represent it as:

$$D = 2 \cdot C_1 + R_1$$

donde  $C_1$  representa el cociente de la división y  $R_1$  el resto. Como estamos dividiendo por dos, el resto solo puede valer cero o uno. Supongamos ahora que volvemos a dividir el cociente obtenido por dos,

where  $C_1$  represents the division quotient and  $R_1$  the remaining after division. As we are dividing by two, the remainder can only be zero or one. Suppose that we divide again the quotient by two,

$$C_1 = 2 \cdot C_2 + R_2$$

Si sustituimos el valor obtenido para  $C_1$  en la ecuación inicial obtenemos,

If we replace the value we got for  $C_1$  in the initial equation, we get,

$$D = 2 \cdot (2 \cdot C_2 + R_2) + R_1 = 2^2 \cdot C_2 + R_2 \cdot 2^1 + R_1 \cdot 2^0$$

Si volvemos a dividir el nuevo cociente obtenido  $C_2$  por dos, y volvemos a sustituir,

If we divide the newly obtained quotient  $C_2$  by two and replace the value again,

$$C_2 = 2 \cdot C_3 + R_3$$

$$D = 2^2 \cdot (2 \cdot C_3 + R_3) + R_2 \cdot 2^1 + R_1 \cdot 2^0 = 2^3 \cdot C_3 + R_3 \cdot 2^2 + R_2 \cdot 2^1 + R_1 \cdot 2^0$$

Supongamos que tras repetir este proceso  $n$  veces, obtenemos un cociente  $C_n = 1$ . Lógicamente no tiene sentido seguir dividiendo ya que a partir de este punto, cualquier división posterior que hagamos nos dará cociente 0 y resto igual a  $C_n$ . Por tanto,

$$D = 1 \cdot 2^n + R_n \cdot 2^{n-1} \cdots + R_3 \cdot 2^2 + R_2 \cdot 2^1 + R_1 \cdot 2^0$$

La expresión obtenida, coincide precisamente con la expansión en potencias de dos del número binario  $1R_n \cdots R_3R_2R_1$ .

Como ejemplo, podemos obtener la representación en binario del número 234, empleando el método descrito: vamos dividiendo el número y los cocientes sucesivos entre dos, hasta obtener un cociente igual a uno y a continuación, construimos la representación binaria del número colocando por orden, de derecha a izquierda, los restos obtenidos de las sucesivas divisiones y añadiendo un uno más a la izquierda de la cifra construida con los restos:

Dividendo Dividend	Cociente $\div 2$ Quotient $\div 2$	Resto Remainder
234	117	0
117	58	1
58	29	0
29	14	1
14	7	0
7	3	1
3	1	1

Por tanto, la representación en binario de 234 es 11101010.

Supongamos ahora un número no entero, representado en decimal, de la forma  $0.d$ . Si lo multiplicamos por dos:

$$E_1.d_1 = 0.d \cdot 2 \quad (1.1)$$

Donde  $E_1$  representa la parte entera y  $d_1$  la parte decimal del número calculado. Podemos entonces representar  $0.d$  como,

$$0.d = (E_1.d_1) \cdot 2^{-1} = E_1 \cdot 2^{-1} + 0.d_1 \cdot 2^{-1} \quad (1.2)$$

After repeating the process  $n$  times, suppose we get a quotient  $C_n = 1$ . There is no point in keeping on dividing because, after this point, any division we do will yield a 0 quotient and a remainder equal to  $C_n$ . Therefore,

The expression we have obtained fits the binary number  $1R_n \cdots R_3R_2R_1$  expansion in powers of two.

For instance, we can compute the binary expression of 234 using the method already described. We divide the number and the successive quotients by two until we get a quotient equal to one. Then, we build the number binary representation, placing the remainders obtained from the consecutive divisions in order, from left to right, and adding a one on the left of the remainders.

Therefore, the binary representation of 234 is 11101010.

Suppose now, that we have non-integer number in denary representation as  $0.d$ . If we multiply the number by two:

Where  $E_1$  represents the integer part and  $d_1$  the decimal part of the computed number. We can then represent  $0.d$  as,

Si volvemos a multiplicar  $0.d_1$  por dos,

If we multiply  $0.d_1$  by two again,

$$E_2 \cdot d_2 = 0.d_1 \cdot 2 \quad (1.3)$$

$$0, d_1 = E_2 \cdot 2^{-1} + 0, d_2 \cdot 2^{-1} \quad (1.4)$$

y sustituyendo en 1.2

and, after substituting in 1.2

$$0.d = E_1 \cdot 2^{-1} + E_2 \cdot 2^{-2} + 0, d_2 \cdot 2^{-2} \quad (1.5)$$

¿Hasta cuando repetir el proceso? En principio hasta que obtengamos un valor cero para la parte decimal,  $0.d_n = 0$ . Pero esta condición puede no cumplirse nunca. Puede darse el caso –de hecho es lo más probable– de que un número que tiene una representación exacta en decimal, no la tenga en binario. El criterio para detener el proceso será entonces obtener un determinado número de decimales o bien seguir el proceso hasta que la parte decimal obtenida vuelva a repetirse. Puesto que los ordenadores tienen un tamaño de registro limitado, también está limitado el número de dígitos con el que pueden representar un número decimal. Por eso, lo habitual será truncar el número asumiendo el error que se comete al proceder así. De este modo, obtenemos la expansión del número original en potencias de dos,

$$0.d \cdot 2 = E_1 \cdot 2^{-1} + E_2 \cdot 2^{-2} + \cdots + E_n \cdot 2^{-n} + \cdots \quad (1.6)$$

Donde los valores  $E_1 \dots E_n$  son precisamente los dígitos correspondientes a la representación del número en binario:  $0.E_1 E_2 \dots E_n$ . (Es trivial comprobar que solo pueden valer 0 ó 1).

Veamos un ejemplo de cada caso, obteniendo la representación binaria del número 0.625, que tiene representación exacta, y la del número 0.626, que no la tiene. En este segundo caso, calcularemos una representación aproximada, tomando 8 decimales.

Para construir la representación binaria del primero de los números, nos basta tomar las partes enteras obtenidas, por orden, de derecha a izquierda y añadir un 0 y la coma decimal a la izquierda. Por tanto la representación binaria de 0.625 es 0.101. Si expandimos su valor en potencias de dos, volvemos a recuperar el número original en su representación

When do we stop the consecutive multiplication process? Basically, we stop when the decimal part becomes zero,  $0.d_n = 0$ . However, it's unlikely that we will reach exactly zero. It's possible that a number with an exact decimal representation does not have an exact binary representation. To stop the process, we can choose to get a predefined number of decimals or continue until the computed decimal part starts repeating. Since computers have limited register size, they can only represent a limited number of decimal digits. Therefore, we often truncate the number, accepting the error it introduces. Thus, we get the original number expansion in powers of two,

Where values  $E_1 \dots E_n$  are precisely the digits corresponding to the binary number representation:  $0.E_1 E_2 \dots E_n$ . (It is trivial to check that they can only be 0 or 1).

Let us see an example of each case, computing the binary representation of the number 0.625 which has an exact binary representation and of the number 0.626 which does not have it. In the second case, we will compute an approximate representation, taking 8 decimals.

For building the binary representation for the first number, it is enough to take the computed integer parts by order, from left to right, and add on the left a zero and the decimal point. Therefore, the binary representation of 0.625 is 0.101. If we expand its value in powers of two, we recover the original number in its decimal representation.

P decimal Decimal P.	$\times 2$ $\times 2$	P entera Integer P.	P decimal Decimal P.	$\times 2$ $\times 2$	P entera Integer P.
0,625	1,25	1	0,623	1,246	1
0,25	0,5	0	0,246	0,492	0
0,5	1,0	1	0,492	0,984	0
			0,984	1,968	1
			0,968	1,936	1
			0,936	1,872	1
			0,872	1,744	1
			0,744	1,488	1

decimal.

En el segundo caso, la representación binaria, tomando nueve decimales de 0,623 es 0.10011111. Podemos calcular el error que cometemos al despreciar el resto de los decimales, volviendo a convertir el resultado obtenido a su representación en base diez,

$$0 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} + 1 \cdot 2^{-5} + 1 \cdot 2^{-6} + 1 \cdot 2^{-7} + 1 \cdot 2^{-8} = 0,62109375$$

El error cometido es, en este caso: Error =  $0,623 - 0,62109375 = 0,00190625$ .

## 1.2. Aplicaciones de Software Científico

Dentro del mundo de las aplicaciones, merecen una mención aparte las dedicadas al cálculo científico, por su conexión con la asignatura.

Es posible emplear lenguajes de alto nivel para construir rutinas y programas que permitan resolver directamente un determinado problema de cálculo. En este sentido, el lenguaje FORTRAN se ha empleado durante años para ese fin, y todavía sigue empleándose en muchas disciplinas científicas y de la Ingeniería. Sin embargo, hay muchos aspectos no triviales del cálculo con un computador, que obligarían al científico que tuviera que programar sus propios programas a ser a la vez un experto en computadores. Por esta razón, se han ido desarrollando aplicaciones específicas para cálculo científico que permiten al investigador centrarse en la resolución de su problema y no en el desarrollo de la herramienta adecuada para resolverlo.

In the second case, the binary representation, taking nine decimals, of the number 0.623 is 0.10011111. We can compute the error made by neglecting the remaining decimals, converting back the number from binary to decimal,

The error we made is in this case,

## 1.2. Scientific software application

mong the computer software applications, those devoted to scientific computing are worth mentioning due to their close connection with the subject.

It is possible to use high-level language to write programs and routines that allow solving specific computation problems. Languages such as FORTRAN have been used for decades to do it, and they are still used in many disciplines of science and engineering. However, many non-trivial software and hardware computing details should force the scientist who writes their own programs to be an expert in computer science. For this reason, specific software applications for scientific computing have been developed, allowing researchers to focus on solving their problems without worrying about creating a suitable tool to solve them.

Some of these applications are ad-hoc, designed specifically for a particular scientific

En algunos casos, se trata de aplicaciones a medida, relacionadas directamente con algún área científica concreta. En otros, consisten en paquetes de funciones específicos para realizar de forma eficiente determinados cálculos, como por ejemplo el paquete SPSS para cálculo estadístico.

Un grupo especialmente interesante lo forman algunos paquetes de software que podríamos situar a mitad de camino entre los lenguajes de alto nivel y las aplicaciones: Contienen extensas librerías de funciones, que pueden ser empleadas de una forma directa para realizar cálculos y además permiten realizar programas específicos empleando su propio lenguaje. Entre estos podemos destacar Mathematica, Maple , Matlab, Octave y Scilab y Python.

area. Others are packages of specific functions, intended to carry out computations efficiently. For example, the software package SPPS is tailored for statistical computation, providing a high level of efficiency for researchers.

There is a particularly interesting group of software packages that bridge the gap between high-level languages and applications. These packages contain extensive function libraries that can be used for a wide range of computations. They also allow the creation of specific programs using their own languages. Among them, we can highlight Mathematica, Maple, Matlab, Octave, Scilab, and Python.



## Capítulo/Chapter 2

# Introducción a la programación en Python

## Introduction to Python Programming

### UN PARACAIDISTA SE DESNUCA

Por darle una sorpresa y a escondidas, su hacendosa, tierna e inocente esposa le cambió a última hora el paracaídas de reglamento por otro de punto, amorosamente tejido por ella, de malla ancha.

---

Julio Ceron. Diario ABC,  
2.03.1986. p.55

Este capítulo presenta una introducción general a la programación. Para su desarrollo, vamos a emplear uno de los lenguajes de programación que más aceptación ha tenido en los últimos años: Python. Este lenguaje fue desarrollado por Guido van Rossum a finales de los años 80 del siglo pasado. Es un lenguaje de alto nivel de propósito general.

El porqué emplear python está relacionado con la existencia de un gran número de *módulos* especialmente diseñados para el cálculo científico. Además, al tratarse de un lenguaje interpretado, es posible centrarse en el estudio de la programación en sí, sin tener que preocuparse de la compilación. En resumen, emplea-

This chapter presents a general introduction to programming. We will use Python, one of the currently most used programming languages, along their pages. Guido van Rossum developed Python in the late 1980s. It is a high-level, general-purpose programming language.

Why Python? Well, it has many useful *modules* that allow us to develop code for scientific computing efficiently and reliably. Moreover, being an interpreted language facilitates focusing on programming skills without bothering with the compiling process. We will use Python to learn basic general programming methods and solve numerical problems.

remos Python tanto para aprender a programar como para resolver problemas de cálculo científico.

Este capítulo no pretende ser exhaustivo, –cosa que por otro lado resulta imposible en el caso de Python–, sino tan solo dar una breve introducción a su uso. Afortunadamente, Python cuenta con una muy buena documentación, accesible a través de la Red.

## 2.1. Un entorno de programación para Python

En primer lugar antes de empezar a describir el lenguaje de programación, vamos a introducir las herramientas que usaremos. Es frecuente que los lenguajes de programación cuenten con lo que se conoce como un entorno de desarrollo integrado o, abreviadamente IDE (acrónimo tomado de su nombre en inglés: *integrated development environment*). Un IDE suministra un habitualmente un entorno gráfico y un conjunto de herramientas tales como ayuda en línea, un depurador o un editor que facilitan la construcción y el depurado del código. Nosotros vamos a emplear un IDE específicamente diseñada para cálculo científico, no es el único, ni es necesariamente el mejor. Pero es adecuado para empezar a trabajar con Python. Se trata de *Spyder*. Veamos en primer lugar como conseguirlo y como instalarlo en un ordenador personal.

### 2.1.1. Anaconda. Una distribución de herramientas de computación de software abierto

Entre los distintos medios en los que podemos instalar Python y algunas de sus herramientas de desarrollo, hemos seleccionado Anaconda porque es software libre, fácil de instalar e incluye Spyder. La Manera más sencilla de instalarlo es descargar el instalador de Anaconda de su página Web:

<https://www.anaconda.com/download>

This chapter is by no means exhaustive. It would be impossible in the case of a language like Python. It is just an introduction to its use. Fortunately, Python has excellent online documentation and plenty of examples on the Web.

## 2.1. A python's development environment

Before describing the programming language, we will introduce the tools we should use. It is common for programming languages to provide an *integrated development environment* (IDE). An IDE usually includes a graphic environment and tools, such as online help, a debugger, or a text editor, that simplify code development and debugging. We are going to use an IDE specifically designed for scientific computing. It's not the only one available; neither is perhaps the best one, but it is suitable to begin working with Python. We refer to *Spyder*. First, Let's see how to get it and install it on a personal computer.

### 2.1.1. Anaconda. A free software computing tools distribution

Among the many ways you may follow to install Python and some of its development tools, We have selected Anaconda because it is free software, easy to install, and includes Spyder. The easy way to install Anaconda is by downloading it from its Web page:

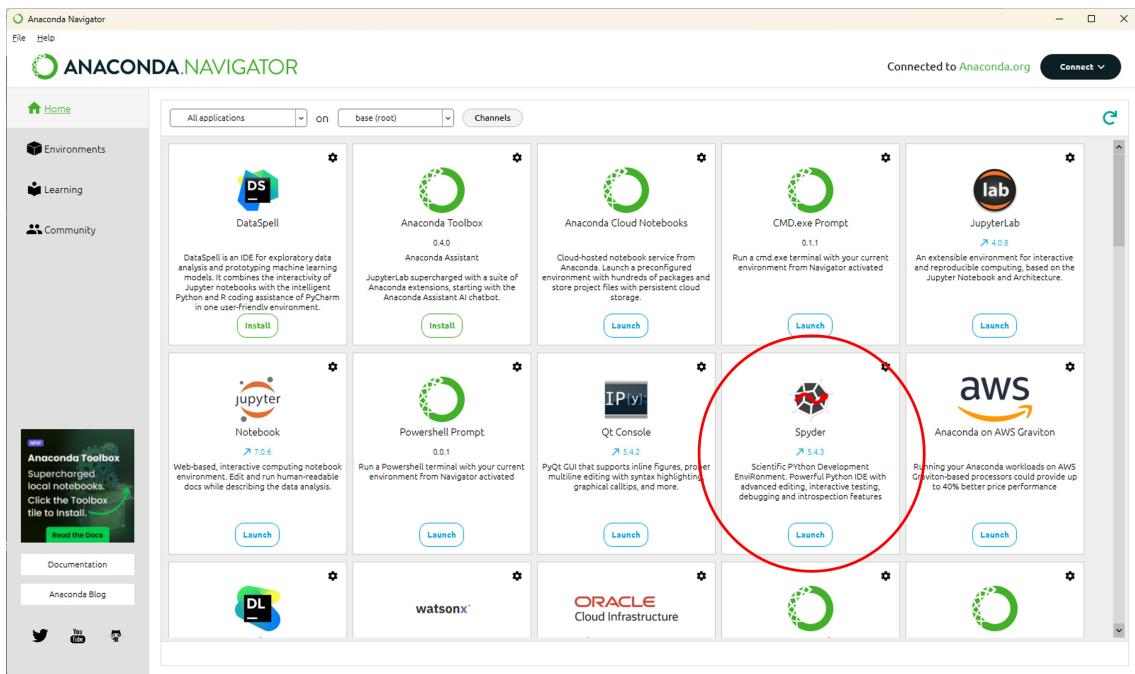


Figura 2.1: Ventana de Anaconda Navigator. Se ha señalado en rojo el ícono correspondiente a Spyder.

Hay instaladores de Anaconda disponibles para Windows, Linux y Mac. El proceso de instalación es en todos los casos bastante sencillo. La página web contiene además documentación en la que se explica detalladamente el funcionamiento de Anaconda.

Una vez instalado, Anaconda proporciona una aplicación (Navegador) desde la que manejarlo: *anaconda-navigator*. Si ejecutamos Anaconda-Navigator, obtendremos una ventana como la que se muestra en la figura 2.1.

El navegador de Anaconda, muestra un conjunto de iconos. Cada uno de ellos corresponde a una aplicación distinta. Muchas de ellas están relacionadas con IDEs especialmente diseñados para usar Python, otras están relacionadas con otros programas orientados al tratamiento de datos, las representaciones gráficas o la estadística. En algunos de los iconos aparece la palabra *launch*. Indica que la aplicación se encuentra instalada en nuestro ordenador,. Si pulsamos con el ratón sobre el botón *launch*, la aplicación se ejecuta. En otros casos, aparece la palabra *install*. Se trata de aplicaciones

Anaconda installers are available for Windows, Linux, and Mac. The installation process is straightforward. The Anaconda webpage offers complete information on installation and performance.

Once installed, it supplies an application *anaconda-navigator* to manage Anaconda. When we start Anaconda-Navigator, we get a window like that shown in figure 2.1.

Anaconda-Navigator shows a set of icons. Anyone of them is intended to launch a different application. Many of them are IDEs specifically designed to use Python. Others relate to other programs devoted to data processing, graphics, or statistics. The word *launch* is written in some of these icons. This means that the application has already been installed on our computer. Pressing the left-hand button of the mouse over the word *launch* opens the application. In other cases, the word *install* is written over the icon. In these cases, the application is not installed on our computer, but we can install it by pressing the left-hand button of the mouse over the word

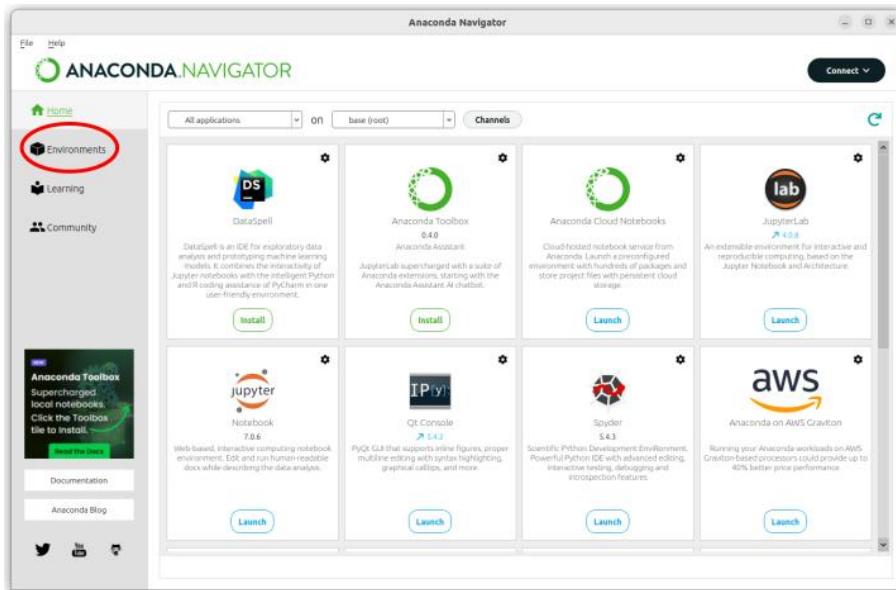


Figura 2.2: Ventana de Anaconda-Navigator, se ha señalado en rojo el botón Environments.

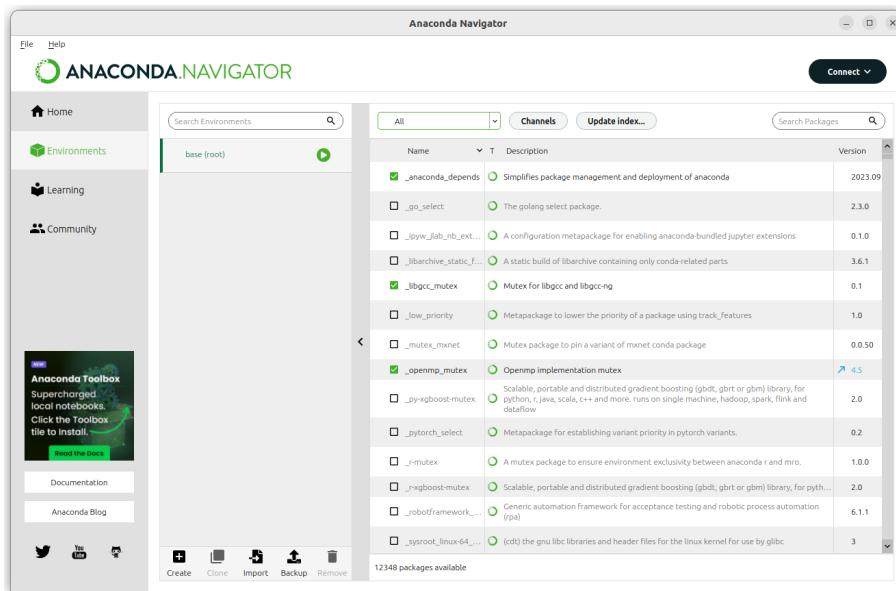


Figura 2.3: Ventana de Anaconda-Navigator, se ha señalado en rojo el botón Environments.

que no están instaladas en nuestro ordenador, pero que pueden ser instaladas si pulsamos en botón *install*.

En nuestro caso, vamos a emplear Spyder, que aparece entre las aplicaciones ya instaladas, y que aparece rodeada en la figura 2.1 con un círculo rojo.

Antes de entrar en la descripción de spyder, vamos a emplear Anaconda navigator para instalar un plug-in, que nos permitirá manejar desde spyder, otro de los entornos de programación de Python más empleados actualmente: *Jupyter Notebook*. Jupiter Notebook, es en realidad un IDE para programar en Python independiente de Spyder. Es muy útil porque permite combinar código en python con texto y gráficos. Más adelante explicaremos como usarlo.

**Instalación de spyder-notebook.** para instalar el plug-in de Spyder que permite manejar notas de jupyter-notes, pulsamos en la ventana de Anaconda-Navigator el botón *Environments*, (enmarcado en rojo en la figura 2.2). La ventana de Anaconda-Navigator nos muestra ahora los entornos de trabajo de Anaconda, (figura 2.3). Por defecto, estaremos en el entorno base (root). Este será además el único entorno disponible si acabamos de hacer una instalación de Anaconda nueva en el ordenador. Si seleccionamos *All* en el menu desplegable situado en la parte superior izquierda del panel de la derecha, obtenemos una relación de todos los paquetes disponibles en anaconda. Aquellos que ya están instalados aparecen marcados a la izquierda con un cuadro verde, mientras que para aquellos disponibles pero no instalados el cuadro aparece en blanco.

En la parte superior derecha de éste panel hay un buscador. Si introducimos en el la palabra spyder, El panel nos mostrará los paquetes relacionados con Spyder (figura 2.4). Seleccionamos el paquete spyder notebook y en la parte inferior derecha del panel nos apareceran dos nuevos botones. Si seleccionamos el botón *apply*, Anaconda comenzará el proceso de instalación del paquete. Lo primero que hace es abrir un pop-up y comprobar las dependencias, una vez terminada la compro-

*install*.

We are going to use Spyder, which has already been installed. In figure 2.1, the Spyder icon has been enclosed in a red circle.

Before starting with a Spyder description, we will use Anaconda to install a plug-in, allowing us to use another of Phyton's most currently used IDEs: *Jupyter Notes*. Jupyter Notes is an IDE for Python code development. It is independent of Spyder. It is a handy tool because it allows us to combine Python code with graphics and text. We will describe Jupyter Notes later on.

**Spyder-notebook installation.** To install the plugin that allows using Jupyter Notes from Spyder, we press the mouse left button on the *Environments* icon (framed in red in figure 2.2). Then, the Anaconda-Navigator window shows us Anaconda work environments (figure 2.3). If we have just installed Anaconda, the only environment already available should be the base (root) environment. We will focus on the right-side panel. The upper left corner has a drop-down menu; we select the option *all*. Then, the panel shows us all the software packages available in the Anaconda default channel. Those packages installed in the computer are ticked with a green square label before the package name; packages available but not installed are ticked with a white square.

There is a browser located in the upper right corner of the panel. We will write the word *spyder* or *spyder notebook*. The panel now shows those software packages related to Spyder (figure 2.4). We select the package spyder notebook by clicking the white square before the package name. A new pair of push buttons appear on the panel's lower side. Clicking on *apply* triggers the Spyder notebook installation process. First, Anaconda launches a pop-up window and checks Spyder Notebook software dependencies. The process may take some time, so be patient until it finishes the checking. Once the checking process is over, we will press the *apply* button on the pop-up window and let Anaconda install the software.

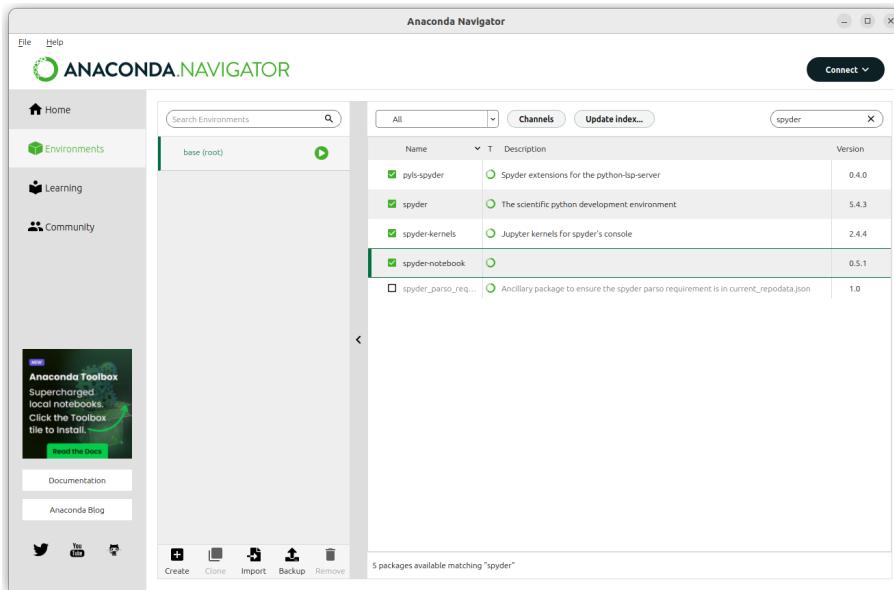


Figura 2.4: Ventana de Anaconda-Navigator, se ha señalado en rojo el botón Environments.

Figure 2.4: Anaconda-Navigator window. The Environments button has been encircled in red

bación, pulsamos el botón apply del pop-up y dejamos que Anaconda instale spyder notebook.

### 2.1.2. Spyder

Vamos a centrarnos en esta sección en describir Spyder. Como ya dijimos, Spyder es un entorno de programación integrado, especialmente pensado para trabajar con Python en computación científica. Para lanzar el entorno basta con que pulsemos el botón *launch* del ícono de Spyder, en la ventana de Anaconda-navigator (figura: 2.1). Al hacerlo, se nos abrirá una nueva ventana como la que se muestra en la figura 2.6.

En la ventana de Spyder podemos distinguir tres paneles distintos<sup>1</sup>. Además, en la parte superior de las ventanas tenemos una barra con menús desplegables y otra con botones. Vamos a describir brevemente algunas de las características principales del entorno a la vez que nos vamos introduciendo en las características fundamentales de la programación

<sup>1</sup>La configuración que Spyder que se describe aquí, es la configuración por defecto. El usuario puede cambiarla a su gusto modificando los valores por defecto.

### 2.1.2. Spyder

Let's focus on the Spyder description. As we said before, Spyder is an Integer Development Environment specially designed to use Python for scientific computing. To start Spyder, we push the *launch* button on the Spyder Icon, located in the Anaconda-Navigator window (figure 2.1). Then, a new window opens, as shown in figure 2.6.

This (Spyder) window has three different panels<sup>1</sup>. Besides, in the upper part of the window, there is a toolbar and a second bar that contains drop-down menus. We are going to give a brief description of the Spyder environment and, at the same time, some of Python programming's basic features. Anyway, we will not be exhaustive. An exhaustive description is far beyond the reach of these notes. The best way to learn to deal with Spyder is by

<sup>1</sup>We describe here the default Spyder configuration. The user may change this default configuration according to their preferences.

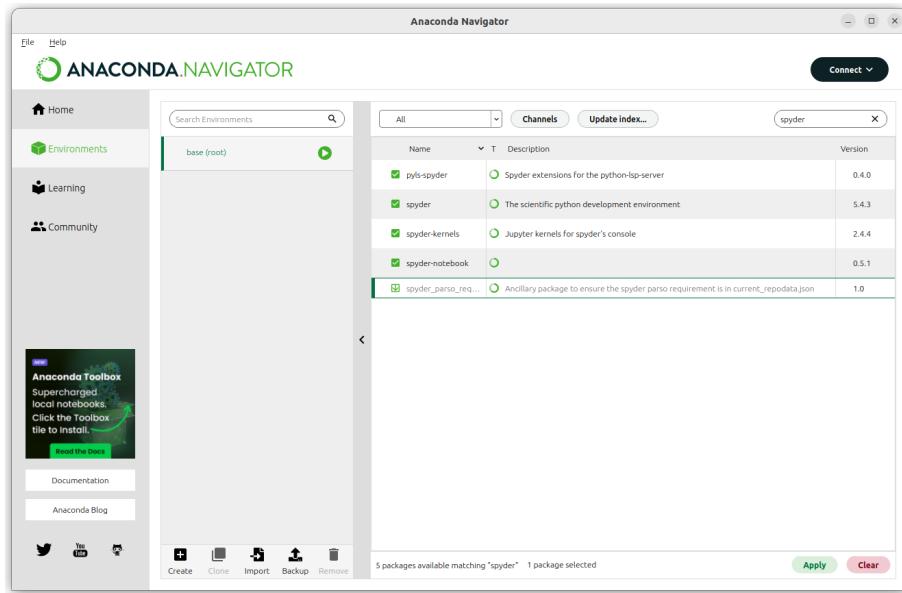


Figura 2.5: Ventana de Anaconda-Navigator, se ha señalado en rojo el botón Environments.  
Figure 2.5: Anaconda-Navgator window. The Environments button has been encircled in red

en Python. En cualquier caso, no vamos a dar una descripción exhaustiva. La mejor manera de aprender a manejar Spyder es usarlo y consultar la abundante documentación disponible.

**El terminal mejorado de Python** De los tres paneles mostrados en la figura 2.6, vamos a describir primero el situado abajo a la derecha. Este panel es un terminal, se conoce con el nombre de Ipython (Interactive python) e incluye muchas mejoras sobre el terminal estándar de python. Ipython nos muestra el símbolo `In [1]:`, que recibe el nombre de *prompt*, y a continuacion una barra vertical | parpadeante. El terminal permite al usuario interactuar directamente con Python; es decir, Python puede recibir instrucciones directamente a través del terminal, ejecutar las instrucciones, y devolver y/o guardar en memoria los resultados obtenidos. Veamos un ejemplo. Si escribimos en el terminal:

`In [1]: 2+2`

y pulsamos la tecla *intro*, Python calcula la suma pedida, muestra el resultado y nos des-

using it. Moreover, there is a vast amount of available documentation.

**The Python’s Enhance Terminal** Returning to the three panels shown in figure 2.6, we will focus first on the left-down one. This panel is a Python console. The console displays a symbol `In [1]:` known as the *prompt* followed by a flicking vertical bar. The console allows the user to interact with Python straight; i.e., Python can get direct instructions through the terminal, run the instructions, and show and/or save in memory the achieved results. Let’s see an example. If we write in the console:

`In [1]: 2+2`

and press the intro key; Python calculates the sum, shows the result using an output (*echo*) mark, `Out [1]:`, gives back a new clean prompt, and stops waiting for a new command:

`Out[1]: 4`

`In [2]:`

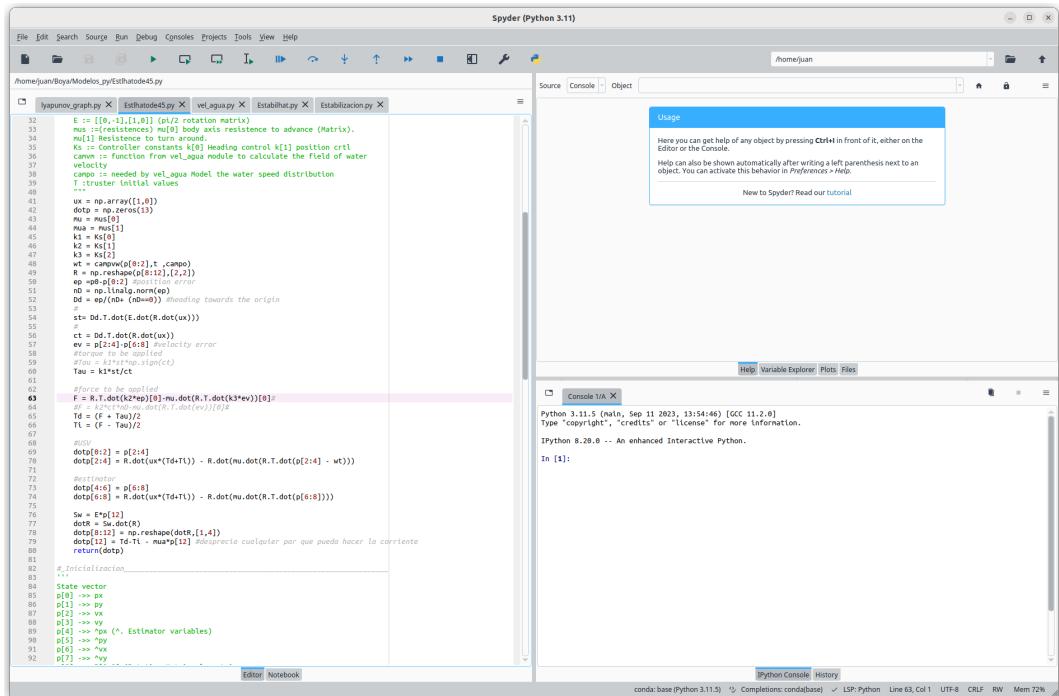


Figura 2.6: Spyder: Entorno de desarrollo integrado para Python  
Figure 2.6: Spyder: An integer development environment for Python

vuelve un nuevo prompt, esperando una nueva orden:

```
Out[1]: 4
In [2]:
```

Es interesante notar que el resultado nos lo ha mostrado empleando una marca de salida (*echo*) **Out[1]:**. En programación, cuando una orden nos muestra por pantalla el resultado de la operación realizada, se dice que el ordenador ha hecho *echo*. Como veremos más adelante esto no es lo más habitual, en la mayoría de los casos, el ordenador ejecuta la orden recibida y cuando termina nos devuelve el prompt **In [2]:** para indicarnos que ha terminado y está listo para recibir una nueva instrucción. De este modo, podemos emplear Python de modo análogo a como empleamos una calculadora. De hecho, la sintaxis es prácticamente la misma.

In programming slang, it is usual to call the answer the computer shows on the screen an *echo*. As we shall see later on, it is not usual for the computer to show us the result of the operations. Most of the time, it just executes the commands, and when it finishes, it shows us the prompt **In[2]:** again to tell us that it is ready for a new command. In this way, we can use Python as a calculator. The syntax is, indeed, quite similar.

### 2.1.3. Variables

El uso de Python como si fuera una calculadora no tiene demasiado interés. Una vez ejecutada la orden, tras pulsar la tecla *intro* el ordenador no guarda ninguna información sobre la operación realizada. Si queremos realizar un cálculo complejo descomponiéndolo en operaciones más sencillas, deberemos anotar los resultados parciales y volver a copiarlos en el terminal si queremos emplearlos de nuevo. Por supuesto, hace ya mucho tiempo que alguien encontró una buena solución, para este y otros problemas similares: el uso de variables.

Podemos ver una variable como una región de la memoria del computador, donde un programa guarda una determina información: números, letras, etc. Una característica fundamental de una variable es su nombre, ya que permite identificarla. Como nombre para una variable se puede escoger cualquier combinación de letras y números, empezando siempre con una letra, en el caso de Python<sup>2</sup>. Se puede además emplear el signo “\_”. Python distingue entre mayúsculas y minúsculas, por lo que si elegimos como nombres de variable Pino, PINO y PiNo, Python las considerará como variables distintas.

El método más elemental de crear o emplear una variable es asignarle la información para la que se creó. Para hacerlo, se emplea el símbolo de asignación , que coincide con el signo = empleado en matemáticas. Como veremos más adelante la asignación en programación y la igualdad en matemáticas no representan exactamente lo mismo. La manera de asignar directamente información a una variable es escribir el nombre de la variable, a continuación el signo de asignación y, por último, la información asignada, `variable_1 = 18`. Si escribimos dicha expresión en la terminal de Python y pulsamos la tecla *intro*:

```
In [2]: variable_1 = 18
In [3]:
```

---

<sup>2</sup>Como se verá más adelante, Python tiene un conjunto de nombres de instrucciones y comandos ya definidos. Se debe evitar emplear dichos nombres, ya que de hacerlo se puede perder acceso al comando de Python que representan

### 2.1.3. Variables

Nevertheless, handling Python as a calculator is hardly ever enjoyable. Once a command has been executed after pressing the *intro* key, the computer keeps no memory of the operation. If we wish to make a complex computation, splitting it into simpler operations, we should take note of the partial results and copy them again on the console to be used again. Of course, a long time ago, somebody found a good solution for this and other similar problems: using variables.

We can consider a variable as a computer memory region where a program has allocated specific information: numbers, characters, etc. A variable fundamental characteristic is its name because it permits one to identify it univocally. . We can take as a variable name whatever combination of lowercase and uppercase letters and numbers, provided that the first character is always a letter. When using Python<sup>2</sup>, We may also use the symbol “\_”. For Python, uppercase and lowercase letters are different symbols. Thus, if we choose variable names, such as Oack, OACK, and OaCk, they represent different variables for Phyton.

The most straightforward method to create or use a variable is to assign the information we want to contain. To do this, we use the assignment symbol , which coincides with the mathematical symbol =. As will be seen later, programming assignment and mathematical equality are not the same concept. To assign some piece of information to a variable, we write the variable name, then the assignment symbol, and, eventually, the assigned information `variable_1 = 18`. If we write this expression on the console and press the *intro* key, we get:

```
In [2]: variable_1 = 18
In [3]:
```

In this case, the computer doesn't echo the result. Anyway, the variable has been saved in

---

<sup>2</sup>As we shall see later on, Python has a set of command names and keywords already defined. We should avoid using these names or keywords as names for our variables. Otherwise, we could lose access to the corresponding Python command or keyword.

En este caso, Python no hace eco, no nos muestra por pantalla ningún resultado. Sin embargo, la variable ha quedado guardada en la memoria del ordenador. Podemos pedirle a Python que nos la muestre,

```
In [3]: variable_1
Out[3]: 18
```

```
In [4]: print(variable_1)
18
```

In [5]:

En el primer caso, hemos escrito directamente el nombre de la variable en el prompt de IPython. En el segundo, hemos hecho uso de una función de Python `print()`. Mas adelante veremos con detalle las funciones en Python. Por el momento, es suficiente con decir que una función es un objeto de programación que toma una variable de entrada y opera sobre dicha variable y nos devuelve un resultado. La función `print()` toma como variable de entrada, una variable cualquiera y nos imprime en el terminal de Ipython su contenido.

Podemos asignar también a una variable el resultado de una operación aritmética,

```
In [5]: variable_2 = 2 * 5
```

Para saber qué variables tiene guardadas el ordenador en memoria, podemos emplear algunos de los comandos especiales de la consola de Ipython,

In [5]: %whos	Type	Data/Info
Variable		
variable_1	int	18
variable_2	int	10

```
In [6]: %who
variable_1
variable_2
```

En el primer caso, hemos empleado el comando de Ipython `whos`. Delante del comando hemos puesto el carácter %, que sirve para indicar al ordenador que se trata de un comando de la Consola y no de Python. El ordena-

the computer's memory. We may ask Python to show it,

```
In [3]: variable_1
Out[3]: 18
```

```
In [4]: print(variable_1)
18
```

In [5]:

In the first case, we have written the name of our variable straightforwardly after the Ipython prompt. In the second case, we use the Python function `print()`. Later on, we will see the concept of function in Python in more detail. Meanwhile, it is enough to say that a function is a programming object that takes an input variable, operates it and returns a result. In our case, the function `print()` takes a variable whatsoever and prints in the Ipython console the content of the variable.

We can also assign to a variable the result of an arithmetic operation,

```
In [5]: variable_2 = 2 * 5
```

To know which variables are saved in the computer memory, we can use some special Ipython console commands:

In [6]: %who

variable\_1

In the first case, we used the Ipython command `whos`. Notice that we have written the symbol % just before the command. This tells the computer we are introducing a console (Ipython) command, not an ordinary Python

dor nos muestra todas las variables que hemos definido, el tipo de variable y el dato que contienen. En el segundo caso hemos empleado el comando `who`, que nos da simplemente una lista de las variables contenidas en memoria.

Acabamos de mencionar, el concepto de tipo de una variable. En algunos lenguajes, es preciso indicar al ordenador qué tipo de información se guardará en una determinada variable, antes de poder emplearlas. Esto permite manejar la memoria del computador de una manera más eficiente, asignando zonas adecuadas a cada variable, en función del tamaño de la información que guardarán. A este proceso, se le conoce con el nombre de *declaración* de variables. En Python no es necesario declarar las variables antes de emplearlas. El tipo de dato se asigna directamente cuando la variable se crea, de acuerdo con la información asignada. Para saber el tipo de dato que contiene una variable se le aplica la función `type()`. Vamos a ver los tipos de datos estándar o *built-in* de Python.

**Numeric.** Se trata de datos que corresponden a valores o cantidades numéricas. Dentro de los datos numéricos, Python define tres tipos distintos:

*Integer.* Permite definir números enteros positivos y negativos. El tipo se representa mediante la abreviatura `int`. Una característica específica de los enteros en Python es que no tienen limitación de tamaño.

```
In [92]: a = 35
In [93]: type(a)
Out[93]: int
```

*Float.* Permite definir números en coma flotante, es decir, una representación aproximada de un número real. (ver el capítulo 5). Se pueden introducir separando la parte entera de la decimal mediante un punto y también en notación científica. El tipo se representa mediante la abreviatura `float`.

Es interesante notar que, `a = 3` Creará una variable entera mientras que `a=3.` creará una variable real.

command. The computer displays the variables we have defined until this point, the variable Type, and the data contained. In the second case, we have used the command `who`, which shows us the bare list of the variables saved in the computer memory.

We just mentioned the *type* of a variable. In some programming languages, it is necessary to explicitly say the kind of information a variable will store before using it. The sort of information stored defines the type of the variable. This helps to manage the computer memory more efficiently, assigning memory zones according to the variable size. This process is known as variable *declaration*. In Python, it is not necessary to declare variables. Python sets the type to a variable when it is created. To know the data type of a variable, we use the function `type()`. Let's see standard or built-in types in Python.

**Numeric.** Data which represent numerical quantities. Inside the numeric data, Python defines three different types:

*Integer.* It allows for representing whole numbers, positives, and negatives. The type is represented by the abbreviation `int`. A specific feature of integers in Python is that they have no size limitation.

*Float.* Floating point numbers, i.e., approximated representations of real numbers. (See chapter 5). We use a point to split the integer and decimal parts of the number. It is also possible to write a floating point number using scientific notation. This type is represented by the word `float`.

Notice that `a = 3` creates an integer variable, but `a=3.` creates a real variable.

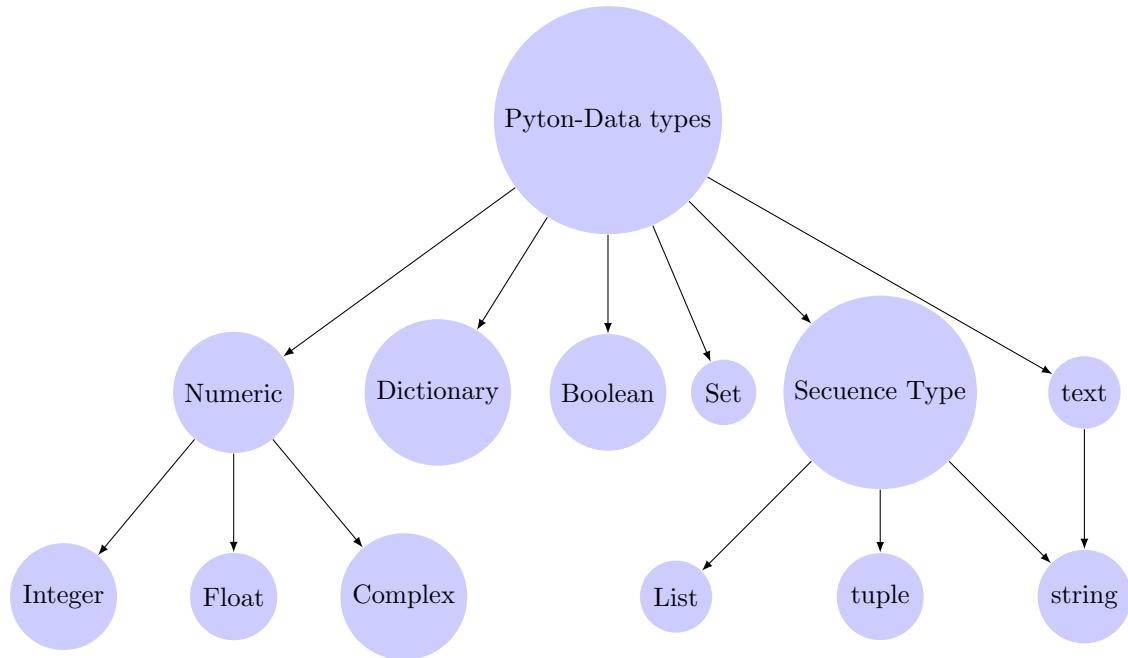


Figura 2.7: Tipos de datos en Python.

Figure 2.7: Data types in Python

```

In [95]: b = -3.5
In [96]: type(b)
In [94]: type(a)
Out[94]: intOut[96]: float
In [98]: c = 3e -4
In [99]: type(c)
Out[99]: float
  
```

*Complex.* Por último, es posible tambien manejar en Python números complejos. Para ello hay que definirlos como la suma de su parte real más su parte imaginaria que va siempre seguida del simbolo *j*. Aunque introduzcamos las partes reales e imaginarias de un número complejo como enteros, Python siempre los considera números en coma flotante.

Finally, it is possible in Python to deal with complex numbers. To define them, we write the real part of the number, the addition symbol, and the imaginary part, followed by the symbol *j*. Notice that Python considers a complex number's real and imaginary parts as floating point numbers even if you write them as integers.

```

In [100]: d = 2+3j
In [101]: type(d)
Out[101]: complex
In [102]: d.real #parte real
#del número d
Out[102]: 2.0
In [103]: d.imag #parte imaginaria del número d
  
```

**Boolean.** Como veremos más adelante, es muy frecuente en programación necesitar saber si una condición se cumple (es verdadera) o no (es falsa) esto lleva a que en muchos lenguajes de programación, existe un tipo de variable que solo toma dos valores: `True` (verdadero) o `False` (falso).

```
In [1]: D = True
In [2]: type(D)
Out[2]: bool
```

**Sequence Type** Todos los tipos incluidos en Sequence Type, así como los tipos dictionary y set, son en realidad estructuras de datos. Podemos verlos como contenedores, que nos permiten almacenar y manipular varios (muchos) datos distintos empleando una sola variable.

*String.* Es el formato propio para crear variables que contengan texto. Se construyen a partir de caracteres UNICODE, encerrados entre comillas,

```
In [3]: tex1 = 'a'
In [4]: type(tex1)
Out[4]: str
In [5]: tex2 ='perro'
In [6]: type(tex2)
Out[6]: str
```

Una propiedad común a todos los tipos sequence es que son indexables. La indexación es una propiedad muy importante que vamos a emplear habitualmente en programación. La variable `tex2` del ejemplo anterior contiene en realidad una cadena de caracteres; las letras que componen la palabra *perro*. Python nos permite extraer individualmente dichos caracteres empleando para ello un índice correspondiente a la posición que ocupan en la cadena. Para ello, escribimos el nombre de la variable seguido del índice encerrado entre corchetes. Hay que tener en cuenta que al primer elemento de una secuencia le corresponde el índice cero.

As we will see later, it is widespread in programming to check if a condition fulfills (it is true) or not (it is false). This leads to defining a type of variable that can take just two possible values: `True` or `False`.

**Sequence Type** Every kind of variable included in Sequence Type, and also the types dictionary and set, are Data structures. We can see them as containers that allow us to deal with several (many) data using a single variable.

*String* This is the proper type to generate variables that contain text. They are built using UNICODE characters enclosed in brackets.

A common property of all types belonging to Sequence Type is that they are indexable. . The indexation is a fundamental property that we frequently use in programming. The variable `tex2` in the previous example is an array of characters, the letters that compound the work *perro* (Dog in Spanish HA, HA). Using an index, Python, allows us to extract such characters individually. The index represents the position the character we want to extract takes in the word. To extract a character from a variable of type String, we write the variable's name followed by the character's index enclosed in square brackets. It is important to note that Python begins to count the elements of a sequence in zero.

```
In [7]: letra_1 = tex[0]
In [8]: print(letra_1)
p
In [9]: letra_4 = tex[3]
In [10]: print(letra_4)
r
```

Python permite también indexar una secuencia, empezando por el final. En esta caso, al último elemento le corresponde el índice -1, al penúltimo -2, etc,

```
In [40]: tex[-1]
Out[40]: 'o'
In [41]: tex[-3]
Out[41]: 'r'
```

*Listas.* Las listas en python son colecciones ordenadas de datos. Los elementos de una lista pueden ser datos de cualquier tipo y no tienen por qué ser homogeneos. Para crear una lista basta escribir los elementos que la componen entre corchetes y separados por comas,

```
In [11]: L = ['carta', 23, 2.5, 1-2j]
In [12]: L
Out[12]: ['carta', 23, 2.5, (1-2j)]
In [13]: type(L)
Out[13]: list
```

Hemos construido una lista en la que el primer elemento es una cadena, el segundo un entero, el tercero un float y el cuarto un número complejo. Debido a su versatilidad, las listas son un tipo de variable muy empleado en Python. Pdemos copiar elementos de una lista a otra variable empleando índices, y tambien podemos asignar a un elemento de una lista un valor nuevo, por supuesto, perderemos el valor antiguo,

```
In [14]: L3 = L[3]
In [15]: print(L3)
(1-2j)
In [16]: L[2] = 0.
In [17]: print(L)
['carta', 0.0, 2.5, (1-2j)]
```

Dado que una lista no es más que una colección ordenada de variables, podemos anidar listas dentro de listas,

Python also permits indexing a sequence beginning by the end. In this case, the sequence's last element takes the index -1, the next-to-last takes -2, and so on.

Lists are ordered collections of data. The elements of a list can be data of whatever type, and they do not need to be homogeneous. To create a list, write the elements that built it up, enclosed in square brackets and separated by commas.

We had built a list in which the first element is a string, the second an integer, the third a real –float type– number, and the fourth is a complex number. Due to their versatility, the Lists are a very useful type in Python. We can copy elements of a list in another variable using indexes and assign a new value to an element of a list. Of course, in this case, we will lose the old value.

As far as a list is nothing more than an ordered collection of variables, we can nest a list inside another list,

append()	añade un elemento al final de la lista adds a element to the end of a list	L.append(-3)
copy()	Crea una copia de la lista Returns a copy of a list	L2 = L.copy()
clear()	Borra todos los elementos de una lista clears all elements from a list	L.clear
count()	Cuenta el número de elementos de una lista Counts the elements of a list	L.count()
insert()	Inserta un nuevo elemento en una posición específica Inserts an element at a specific position in a list	L.insert(i,-4)
pop()	Extrae el ultimo elemento de una lista. Si se añade un índice extrae el elemnto indicado extract the last element of a list. Addind and index, extract the specific element	a =L.pop() a=L.pop(i)

Tabla 2.1: Algunos métodos de las Listas en Python

Table 2.1: Some methods of Python's List

```
In [14]: LL = [1, 'p', ,L,-0.7]
In [15]: print(LL)
[1, 'p', ['carta', 23, 2.5, (1-2j)], -0.7]
```

Hemos incluido la lista L como el tercer elemento de la nueva lista LL. Podemos extraer un elemento de la lista interior, empleando dos índices; el primero para referirnos a la posición de la lista L dentro de la lista LL, y el segundo para indicar la posición del elemento deseado dentro de la lista L. Así por ejemplo, si queremos obtener la palabra 'carta',

```
In [16]: ctr = LL[2][0]
In [17]: print(ctr)
carta
```

Podemos cambiar datos de una lista, usando el simbolo de asignación,

```
In [18]: LL[3] = 'J'
In [19]: print(LL)
[1, 'p', ['carta', 23, 2.5, (1-2j)], 'J']
```

Una característica importante de las listas, son sus métodos. Los métodos son funciones especiales que nos permiten manipular las listas de una manera directa. La tabla 2.1 muestra algunos de los más usuales. La forma de emplearlos es escribir el nombre de la lista, seguido de un punto el nombre del método y, entre paréntesis los parámetros que toma. Si el método no toma ningún parámetro se deja el

We have included the list L as the third element of a new list LL. We can extract an element from the inner list using two indexes. The first one indicates the position of the list L inside the list LL, and the second one indicates the position of the wanted element inside the list L. For instance, if we want to get the word 'carta',

We can change list data using the assignment symbol,

An essential feature of Lists is their methods. The methods are special functions that allow us to manipulate the lists in a straight way. Table 2.1 shows some of the most used list methods. To use a method, we write the list name followed by a point and name of the method and, enclosed in brackets, the parameters the method takes. If the method doesn't use parameters, we write an empty bracket

paréntesis vacío, pero nunca se omite. Veamos algunos ejemplos:

```
In [12]: L = [1, 'a', -0.5, 'en', 2.5, 'a']
In [13]: print(L)
[1, 'a', -0.5, 'en', 2.5, 'a']
In [14]: L.append(-2)
In [15]: print(L)
[1, 'a', -0.5, 'en', 2.5, 'a', -2]
In [16]: L.pop()
Out[16]: -2
In [17]: print(L)
[1, 'a', -0.5, 'en', 2.5, 'a']
In [18]: L.insert(3, 'insrt')
In [19]: print(L)
[1, 'a', -0.5, 'insrt', 'en', 2.5, 'a']
In [20]: L.pop(3)
Out[20]: 'insrt'
In [21]: print(L)
[1, 'a', -0.5, 'en', 2.5, 'a']
In [22]: L.count('a')
Out[22]: 2
In [23]: L.count('s')
Out[23]: 0
```

*Tuplas* Las tuplas, son similares a las listas. La principal diferencia es que son inmutable. es decir, una vez que se han creado no se puede modificar el valores de los elementos que contiene. Para crearlas se encierran los elementos que constituyen la tupla entre paréntesis y separados con comas. En el ejemplo que sigue se observa como al intentar modificar un elemento de una tupla una vez creada, Python nos devuelve un aviso de error.

```
In [41]: T = (0, 'int', 2.4, [1, 4, 3.5])
In [42]: T
Out[42]: (0, 'int', 2.4, [1, 4, 3.5])
In [43]: T = (0, 'int', 2.4, [1, 4, 3.5])
In [44]: print(T)
(0, 'int', 2.4, [1, 4, 3.5])
In [45]: type(T)
Out[45]: tuple
In [46]: T[3]
Out[46]: [1, 4, 3.5]
In [47]: T[4] = 12
Traceback (most recent call last):
Cell In[47], line 1
    T[4] = 12
TypeError: 'tuple' object does not support item assignment
```

after the method name but never leave it off.  
Let's see some examples:

*tuples* Tuples are similar to List. The main difference between them is that Tuples are immutable, i.e., once created, the elements of a Tuple cannot be modified. To build a Tuple, we enclose their elements into brackets, separated by commas. The following example shows how Python throws an error message when we try to modify a Tuple element once it has been built.

**Dictionary.** Los diccionarios son estructuras en las que los datos aparecen asociados a claves. La forma más sencilla de crearlos es mediante pares clave:valor separados por comas y encerrados entre llaves. El símbolo ':' asocia cada elemento con su clave. Un diccionario puede contener dentro cualquier tipo de variable, incluidas listas u otros diccionarios. Para acceder a un elemento guardado en un diccionario se escribe el nombre del diccionario seguido de la clave del elemento deseado escrita entre corchetes.

```
In [63]: D = {'nombre': 'Pepe', 'día': 27, 'mes': 'Febrero', 'datos': [1, 3.5, 6, 0.0]}
In [64]: D
Out[64]: {'nombre': 'Pepe', 'día': 27, 'mes': 'Febrero', 'datos': [1, 3.5, 6, 0.0]}
In [65]: type(D)
Out[65]: dict
```

Para añadir nuevos elementos a un diccionario, se emplea el nombre del diccionario seguido de la clave que tendrá el elemento, escrita entre corchetes y se usa el símbolo de asignación para añadir el valor del elemento. Al igual que sucedía en el caso de las listas, es posibles anidar diccionarios dentro de otros diccionarios. Para acceder al diccionario interno, empleamos su clave. Para acceder a un elemento del diccionario interno empleamos la clave del diccionario interno seguida de la clave del elemento. Para borrar un elemento de un diccionario empleamos la orden de python `del()`. Igual que las listas, los diccionarios cuentan con un buen número de métodos propios. Los interesados pueden consultarlos en las páginas de referencia de Python.

A dictionary is a structure in which data are associated with keys. The simplest way to create them is by using a pair key: data separated by commas and enclosed in curly braces. The symbol ':' associates each element with its key. A dictionary can contain whatever type of variable, including lists and other dictionaries. To get access to data stored in a dictionary, we write the name of the dictionary followed by the key of the element we want to access, enclosed in square brackets.

To add new elements, we write the name of the dictionary followed by the key we want to assign to the element, enclosed in square brackets, and we use the assignment symbol to add the element value. As in the case of lists, it is possible to nest one dictionary into another. To access the inner dictionary, we use its key. To access data inside the inner dictionary, we use the inner dictionary key followed by the data key. (see In[78]: in the example below). To eliminate an element from a dictionary, we use the Python command `del()`. Similar to the Lists, there are quite a few dictionary methods. Readers interested should be addressed to Python reference pages.

```
In [75]: D['nuevo'] = {'calle': 'Atocha', 'num.': 18, 'Piso': '3D'}
In [76]: D
Out[76]:
{'nombre': 'Pepe',
 'día': 12,
 'mes': 'Febrero',
 'datos': [1, 3.5, 6, 0.0],
 'nuevo': {'calle': 'Atocha', 'num.': 18, 'Piso': '3D'}}
In [77]: D['día']
Out[77]: 12
In [78]: D['nuevo']['num.']}
```

```

Out[78]: 18
In [81]: del(D['día'])
In [82]: D
Out[82]:
{'nombre': 'Pepe',
 'mes': 'Febrero',
 'datos': [1, 3.5, 6, 0.0],
 'nuevo': {'calle': 'Atocha', 'num.': 18, 'Piso': '3D'}}

```

**Set.** Los conjuntos son colecciones datos no repetidos e inmutables. Para crear un conjunto en Python se escriben sus elementos separados por comas y encerrados entre llaves. Si hay elementos repetidos en la definición, el conjunto creado solo los contendrá una vez. No vamos a verlos en más detalle.

```

In [85]: C= {'L', 'M', 'X', 'J', 'V'}
In [86]: C
Out[86]: {'J', 'L', 'M', 'V', 'X'}
In [87]: C= {'L', 'M', 'X', 'J', 'V', 'M'}
In [88]: C
Out[88]: {'J', 'L', 'M', 'V', 'X'}
In [89]: C[1] = 23
Traceback (most recent call last):
Cell In[89], line 1
      C[1] = 23
TypeError: 'set' object does not support item assignment

```

## 2.2. Operaciones aritméticas, relacionales y lógicas.

### 2.2.1. Operaciones aritméticas

Una vez que sabemos como crear variables en Python, vamos a ver como podemos realizar operaciones aritméticas elementales con ellas. La sintaxis es muy sencilla, y podemos sintetizarla de la siguiente manera:

```

resultado = operando1operador1
           operando2operador2operando3 ...
           operadorn-1operandon

```

Es decir basta concatenar los operadores con los operandos y definir una variable en la que guardar el resultado. Por ejemplo,

**Set.** Sets are immutable non-repeated data collections. To make a set in Python we write its elements separated by commas and enclosed in curly braces. If there are repeated elements in its definition, the set created will contain a single instance of the repeated element.

## 2.2. Arithmetical, relational and logical operations

### 2.2.1. Arithmetic operations

Once we know how to create variables in Python, we will see how to perform basic arithmetic operations. The syntax is pretty simple, and we can synthesize it as follows:

```

result = operand1operator1
         operand2operator2operand3 ...
         operatorn-1operandn

```

That is, it is enough to concatenate operands and operators and define a variable that gets the result. For instance,

```
In [1]: a = 3
In [2]: b = 4
In [3]: c = 12.4
In [4]: d = a + b - c
In [5]: print(d)
-5.4
```

En este ejemplo los operandos son las variables `a`, `b`, `c`, los operadores empleados son el símbolo `+` que representa la operación suma y el símbolo `-` que representa la resta. `d` es la variable en la que se guarda el resultado, en este caso, de la suma de las dos primeras variables y su diferencia con la tercera.

Los operadores aritméticos disponibles en Python cubren las operaciones aritméticas habituales. La tabla 2.2 contiene los operadores definidos en python.

In this example, the operands are the variables `a`, `b`, `c`. the operator are the symbol `+` which represents the addition operation and the symbol `-` which represents the subtraction. `d` is the variable that holds the results. In this case, the addition of the two first variables and the result with the third one.

Python available arithmetics operators cover the usual arithmetic operations. Table 2.2 shows the arithmetic operators defined in Python.

Tabla 2.2: Operadores aritméticos definidos en Python  
Table 2.2: Arithmetic operators defined in python

Operación Operation	Símbolo Symbol	Uso Use	notas notes
Suma Addition	<code>+</code>	<code>r=a+b</code>	
Diferencia Subtraction	<code>-</code>	<code>r=a-b</code>	
Producto Product	<code>*</code>	<code>r=a*b</code>	
División division	<code>/</code>	<code>d=a/b</code>	
División entera Integer division	<code>//</code>	<code>d=a//b</code>	Calcula la división entera. Divide y redondea el cociente hacia cero. Calculate the division and round the quotient towards zero
Resto de la división entera Modullus	<code>%</code>	<code>d=a % b</code>	Calcula el resto de la división entera Calculate the remainder after integer division
Potenciación Power	<code>**</code>	<code>y=a ** b</code>	Potencia. Eleva <code>a</code> al exponente <code>b</code> : $a^b$ Raise <code>a</code> to the power of <code>b</code>

```
In [92]: a = 5
In [93]: b = 3
In [94]: print(a+b)
8
In [95]: print(a-b)
2
In [96]: print(a*b)
15
In [97]: a = 5
In [98]: b = 3
In [99]: suma = a+b
In [100]: print(suma)
8
In [101]: dif = a-b
In [102]: print(dif)
2
In [103]: prod = a*b
In [104]: print(prod)
15
In [105]: div = a/b
In [106]: print(div)
1.6666666666666667
In [107]: div_ent = a//b
In [108]: print(div_ent)
1
In [109]: rem = a%b
In [110]: print(rem)
2
In [111]: pot = a**b
In [112]: print(pot)
125
```

## 2.2.2. Precedencia de los operadores aritméticos

Los ejemplos anteriores muestran el uso básico de los siete operadores aritméticos definidos en Python.

Combinando operadores aritméticos, es posible elaborar expresiones complejas. Por ejemplo,

```
In[1]: R=5*3-6/3+2**3+2-4
```

La pregunta que surge inmediatamente es en qué orden realiza Python las operaciones indicadas. Para evitar ambigüedades, Python —como todos los lenguajes de programación— establece un orden de precedencia, que permite saber exactamente en qué orden se realizan

## 2.2.2. Arithmetic operator precedence

The previous examples show a basic use of the seven arithmetic operators defined in Python.

By combining arithmetic operators, it is possible to build up complex expressions. For instance,

```
In[1]: R=5*3-6/3+2**3+2-4
```

A question arises: in which order does Python carry out the operations involved in this expression? To avoid ambiguities, Python—as any other programming language—establishes a precedence order that allows knowing exactly in which order the operations will be carried out. Python precedence order is:

las operaciones. En Python el orden de precedencia es:

1. En primer lugar se calculan las potencias.
2. A continuación los productos y las divisiones, que tienen el mismo grado de precedencia.
3. Por último, se realizan las sumas y las restas.

Por tanto, en el ejemplo que acabamos de mostrar, Python calcularía primero,

$2^{**}3=8$

a continuación el producto y la división

$5*3=15$

$6/3=2$

Por último sumaría todos los resultados intermedios, y guardaría el resultado en la variable R

$15-2+8-4=17$

R=17

**Uso de paréntesis para alterar el orden de precedencia.** Cuando necesitamos escribir una expresión complicada, en muchos casos es necesario alterar el orden de precedencia. Para hacerlo, se emplean paréntesis. Sus reglas de uso son básicamente dos:

- La expresiones entre paréntesis tienen precedencia sobre cualquier otra operación.
- Cuando se emplean paréntesis anidados (unos dentro de otros) los resultados siempre se calculan del paréntesis más interno hacia fuera.

Por ejemplo,

```
In[1]: y=2+4/2
In[2] : print(y)
4
In[3]: y=(2+4)/2
In[4]: print(y)
3
```

1. First, it calculates the powers.
2. Then, products and divisions that share the same precedence degree.
3. Eventually, additions and subtractions are carried out.

Thus, in the example we have just shown, Python would calculate first,

$2^{**}3=8$

then, the product and the division

$5*3=15$

$6/3=2$

eventually, it would sum up all intermedia results and save the final result in the variable R.

$15-2+8-4=17$

R=17

**Using parentheses to modify the precedence order.** When evaluating a complex expression, we need to modify the order of precedence in many cases. To do it, we use parentheses. The rules of use are mainly two:

- Expressions enclosed in parenthesis have precedence over whatever other operation.
- When using nested paratheses (parenthesis enclosed in other parentheses). The results are always obtained from the inner parenthesis to the outer one.

For instance,

```
In[1]: y=2+4/2
In[2] : print(y)
4
In[3]: y=(2+4)/2
In[4]: print(y)
3
```

In the first operation, the precedence order makes Python divide 4 between 2 and then add 2 to the result. In the second case, the Parenthesis has precedence; Python first adds 2 and 4 and then divides the result between 2.

En la primera operación, el orden de precedencia de los operadores hace que Python divida primero 4 entre 2 y a continuación le sume 2. En el segundo caso, el paréntesis tiene precedencia; Python suma primero 2 y 4 y a continuación divide el resultado entre 2.

El uso correcto de los paréntesis para alterar la precedencia de los operadores, permite expresar cualquier operación matemática que deseemos. Por ejemplo calcular la hipotenusa de un triángulo rectángulo a partir de valor de sus catetos,

$$h = (c_1^2 + c_2^2)^{\frac{1}{2}}$$

Que en Python podría expresarse como,

```
In[1]: h=(c1^2+c2^2)**(1/2)
```

O la expresión general para obtener las raíces de una ecuación de segundo grado,

$$x = \frac{-b \pm (b^2 - 4 \cdot a \cdot c)^{\frac{1}{2}}}{2 \cdot a}$$

en este caso es preciso dividir el cálculo en dos expresiones, una para la raíz positiva,

```
In[2]: x=(-b+(b^2-a*c)**(1/2))/(2*a)
```

y otra para la raíz negativa

```
In[3]: x=(-b-(b^2-a*c)**(1/2))/(2*a)
```

Es necesario ser cuidadosos a la hora de construir expresiones que incluyen un cierto número de operaciones. Así, en el ejemplo que acabamos de ver, el paréntesis final `2*a` es necesario; si se omite, Python multiplicará por `a` el resultado de todo lo anterior, en lugar de dividirlo.

### 2.2.3. Operaciones Relacionales y lógicas.

Aunque son distintas, las operaciones relacionales y las lógicas estas estrechamente relacionadas entre sí. Al igual que en el caso de las operaciones aritméticas, en las operaciones relativas y lógicas existen operandos – variables sobre las que se efectúa la operación – y operadores, que indican cuál es la operación

Using parentheses to alter the operator's precedence allows building whatever mathematical expression we wish. For example, to calculate the hypotenuse of a rectangular triangle using the values of its catheti,

$$h = (c_1^2 + c_2^2)^{\frac{1}{2}},$$

we may express this in Python as,

```
In[1]: h=(c1^2+c2^2)**(1/2)
```

Or, as another example, the generic solution of a quadratic equation

$$x = \frac{-b \pm (b^2 - 4 \cdot a \cdot c)^{\frac{1}{2}}}{2 \cdot a},$$

in this case, it is necessary to split the result into two expressions: one for the positive root,

```
In[2]: x=(-b+(b^2-a*c)**(1/2))/(2*a)
```

and the second one for the negative root.

```
In[3]: x=(-b-(b^2-a*c)**(1/2))/(2*a)
```

Caution is needed when building expressions that contain a large number of operations. Take the example just shown: if we forget the last parenthesis `2*a`, Python would multiply by `a` the result of the remaining operation instead of dividing it by `a`.

### 2.2.3. Relational and logical operations.

Although they are not equal, relational and logical operations are strongly related. As we have seen in the case of arithmetic operations, relational and logic operations are built up using operands—variables on which we perform the operations—and operators that indicate which operations we carry out on the

que se efectúa sobre los operandos. La diferencia fundamental es que tanto en el caso de las operaciones relacionales como lógicas el resultado solo puede ser 1 (**True**) o 0 (**False**).

**Operadores relacionales.** La tabla 2.3 muestra los operadores relacionales disponibles en el entorno de Python. Su resultado es siempre la verdad o falsedad de la relación indicada.

Tabla 2.3: Operadores relacionales definidos en Python

Table 2.3: Relational operators defined in Python

operación operation	símbolo symbol	ejemplo example	notas notes
menor que minor than	<	r=a<b	El resultado es <b>True</b> si $a$ es menor que $b$ . En otro caso el resultado es <b>False</b> . The result is <b>True</b> if $a$ is minor than $b$ . Otherwise the result is <b>False</b>
mayor que greater than	>	r=a>b	El resultado es <b>True</b> si $a$ es mayor que $b$ . En otro caso el resultado es <b>False</b> . The result is <b>True</b> if $a$ is greater than $b$ . Otherwise the result is <b>False</b>
mayor o igual que greater than or equal to	>=	r=a>=b	El resultado es <b>True</b> si $a$ es mayor o igual que $b$ . En otro caso <b>False</b> . The result is <b>True</b> if $a$ is minor than $b$ or equal to $b$ . Otherwise the result is <b>False</b>
menor o igual que Less than or equal to	<=	r=a<=b	El resultado es <b>True</b> si $a$ es menor o igual que $b$ . En otro caso el resultado es <b>False</b> . The result is <b>True</b> if $a$ is greater than $b$ or equal to $b$ . Otherwise the result is <b>False</b>
igual a equal to	==	a==b	El resultado es <b>True</b> si $a$ es igual a $b$ . En otro caso el resultado es <b>False</b> . The result is <b>True</b> if $a$ is equal to $b$ . Otherwise the result is <b>False</b>
Distinto de not equal to	!=	a!=b	El resultado es <b>True</b> si $a$ es distinto de $b$ . En otro caso el resultado es <b>False</b> . The result is <b>True</b> if $a$ is not equal to $b$ . Otherwise the result is <b>False</b>

Es importante señalar que el operador relacional que permite comparar si dos variables son iguales es **==** (doble igual), no confundirlo con el igual simple **=** empleado como sabemos como símbolo de asignación.

**Operadores Lógicos** En Python se distinguen tres conjuntos de operadores lógicos según el tipo de variable sobre la que actúen. Aquí

operand. The main difference with arithmetic operators is that both relational and logic operations would only cast 1 (**True**) or 0 (**False**) as an operation result.

**Relational Operators.** Table 2.3 shows the relational operators available in Python. Their result is always the truth or falsehood of the relationship the operator represents.

It is important to note that the symbol double-equal **==** compares whether two variables are equal. Please do not mistake it for the assignation symbol **=**.

**Logical operator** Python distinguishes three sets of logical operators. We will only present one of them: Logical operators to compare variables.

vamos a ver solo uno de ellos: los operadores lógicos entre variables.

La tabla 2.4 muestra los operadores lógicos entre variables. El resultado, es siempre un (1) **True** o un (0) **False**.

Table 2.4 shows the logical operator defined in Python for variables. The result is always (1) **True** or (0) **False**.

Tabla 2.4: Operadores lógicos entre valores y variables  
Table 2.4: Logical operators on values and variables

operación operation	símbolo symbol	ejemplo example	notas notes
and	and	r=a and b	Operación lógica <i>and</i> entre las variables a y b Logical operation <i>and</i> between variables a and b
or	or	r=a or b	Operación lógica <i>or</i> entre las variables a y b Logical operation <i>or</i> between the variables a and b
negación not	not	r= not a	complemento de a (si a es <b>True</b> entonces <b>not</b> a es <b>False</b> ) a complement (if a is <b>True</b> then <b>not</b> a is <b>False</b> )

En cuanto a su funcionamiento, son los operadores típicos del álgebra de Bool. Así el operador **and** sigue la tabla de verdad propia de la operación *and*, el resultado solo es verdadero (1) si sus operandos son verdaderos (1)<sup>3</sup>,

<sup>3</sup>En realidad, Python considerará verdadero cualquier operando distinto de 0

A logical operator works following the standard of Bool's Algebra. So, the **and** operator follows the true table of the *and* operation in Bool's Algebra, the result is true (1) only if both its operands are true,<sup>3</sup>

<sup>3</sup>In fact, Python takes any variable a true, whenever it is not zero.

Tabla de verdad de la operación **and**  
Truth table for **and** operation

operando 1 operand 1	operando 2 operand 2	resultado result
1	1	1
1	0	0
0	1	0
0	0	0

el operador **or**, responde a la tabla de verdad del operación booleana *or*, el resultado es verdadero si cualquiera de sus operandos es verdadero o si ambos lo son.

Veamos a continuación como los operadores de lógicos de Python satisfacen las tablas de verdad,

The operator **or** follows the true table of the boolean operation *or*, and the result is true if any one of their operands is true or both of them are true.

Let's see how the logical operator in Python fullfil the truth tables,

Tabla de verdad de la operación `or`Truth table for `or` operation

operando 1 operand 1	operando 2 operand 2	resultado result
1	1	1
1	0	1
0	1	1
0	0	0

In [1]: `a = 1`In [2]: `b = 0`In [3]: `c = 1`In [4]: `d = 0`In [5]: `a and b`

Out[5]: 0

In [6]: `a and c`

Out[6]: 1

In [7]: `b and d`

Out[7]: 0

In [8]: `a and d`

Out[8]: 0

In [9]: `a or b`

Out[9]: 1

In [10]: `a or c`

Out[10]: 1

In [11]: `a or d`

Out[11]: 1

In [13]: `b or d`

Out[13]: 0

In [14]: `not a`Out[14]: `False`In [15]: `not b`Out[15]: `True`

Es importante destacar que para Python, las operaciones lógicas devuelven valores = 0, si el resultado es cierto ó  $\neq 0$  si es falso, mientras que las operaciones relaciones y el complemento (la negación) devuelven valores lógi-

It is important to remark that logical operations in Python cast a value = 0 if the result is true or  $\neq 0$  if the result is false, while relational operations and the complement (negation) cast boolean values *True* or *false*. That is

cos *True* o *False*. En realidad, esto no supone ningún problema, ya que el lenguaje maneja estos valores lógicos igual que sus equivalentes enteros 1 y 0. Otro aspecto interesante tiene que ver con el hecho de que Python considere como verdadera, cualquier variable que tome un valor distinto de cero. Esto no vas a permitir comentar algunos aspectos del modo en que se llevan a cabo las operaciones lógicas. observa los siguientes ejemplos,

```
In [25]: vdo = 7.5
In [26]: fls = 0
In [27]: vdo2 = 1
In [28]: vdo3 = 18
In [29]: vdo and fls
Out[29]: 0
In [30]: vdo or fls
Out[30]: 7.5
In [31]: vdo and vdo2
Out[31]: 1
In [32]: vdo2 and vdo
Out[32]: 7.5
In [33]: vdo or vdo2
Out[33]: 7.5
In [34]: vdo2 or vod
Out[34]: 1
```

En las líneas In[25] a In[26] hemos creado cuatro variables numérica. Para Python todas ellas son verdaderas excepto `fls`, que vale cero y por tanto es falsa. Si aplicamos el operador `and` a las variables `vdo` y `fls`, el resultado es cero, puesto que una de las variables es cero. Sin embargo, en la línea In[30] observamos que el resultado de la operación `or` realizada es 7.5, que es valor de la primera variable. Este resultado está relacionado con el modo en el que Python realizar las operaciones lógicas. Para una operación `or` es suficiente que uno de los dos operando sea distinto de cero (cierto). El programa comprueba que el primer operando cumpla con la condición, pero una vez que ha comprobado que la primera variable es distinta de cero, ya no necesita comprobar más, sea cual sea el valor de la segunda el resultado será cierto. así que, se límita a devolvernos el valor de la primera variable, distinto de cero y por tanto cierto.

Por contraste, en las líneas In[31] e In[32]

not a problem at all because Python manages these boolean values as their integer counterpart 1 and 0. The fact that Python considers true any value not equal to zero will help us to show how it carries out logical operations, get a lookout to the next examples,

We defined four numeric variables in lines In[25] to In[26]. All of them are true for Python except for `fls`, which is zero and, therefore, false. If we apply the operator `and` to variables `vdo` and `fls`, the result is zero because one of the variables is zero. However, in line In[30], we see that the result of the `or` operation carried out is 7.5, Which is the value of the first variable, i.e., `vdo`. This result is related to Python's way of performing logic operations. For an `or` operation to yield true, it is enough that one of the operands be not equal to zero (true). The program begins trying the first operand, and once it checks that it is true, it is not necessary to test the value of the second operand because the result of the operation will be true, no matter which value the second operand takes. So, once Python is sure the first operand is not equal to zero, it casts its value as the *true* result of the operation.

By contrast, in lines In[31] and In[32], we

se le pide a python que evalue la operación *and* aplicada a las variables, `vdo` y `vdo2`. En este caso, Python tiene necesariamente que comprobar que las dos variables cumplen la condición, puesto que aunque la primera variable sea distinta de cero, el resultado solo es cierto si también la segunda variable es distinta de cero.

Es interesante notar cómo el resultado de la operación cambia al cambiar el orden de los operandos. En los dos casos (`In[31]` e `In[32]`) el resultado de la operación es cierto, pero en cada caso devuelve como valor distinto de cero, el valor de la segunda variable, que es la última cuya validez ha comprobado Python. Mira el resto de los ejemplos, y comprueba que siempre se cumple el criterio descrito.

Por último, indicar que los operadores lógicos pueden combinarse entre sí con operadores relacionales y con operadores aritméticos. El orden de precedencia es el siguiente:

1. Paréntesis ()
2. Operadores aritméticos en su orden de precedencia
3. Operadores relacionales, todos tienen el mismo orden de precedencia por lo que se evalúan de izquierda a derecha
4. `and`
5. `or`

Es aconsejable el uso de paréntesis cuando se encadenan varias operaciones lógicas para asegurar su uso correcto y facilitar la lectura de las sentencias.

Por ejemplo,

```
In [3]: a = 12.5
In [4]: b = 3.6
In [5]: c = -3.2
In [6]: d = 0.

In [15]: a>b or d>c
Out[15]: True

In [16]: (a>b) or (d>c)
Out[16]: True
```

ask Python to evaluate the result of the *and* operation applied to `vdo` and `vdo2`. Now, Python has to check that both variables fulfil the condition. Although the first variable is not equal to zero, the result will only be true if the second variable is not zero.

It's worth noting that the result of this operation can change depending on the order of the operators. In both cases (`In[31]` and `[32]`), the operation yields true, but Python assigns the value of the second operand as the result because it is the last variable that Python has checked.

Lastly, we can combine logical operators with relational and arithmetic operators. The precedence order is as follows,

1. Parenthesis ()
2. Aritmétical operators in their order of precedence
3. Relational operator, all have the same precedence order; thus, they are evaluated from left to right.
4. `and`
5. `or`

Using parenthesis when several logical operations are linked is good practice to ensure their correct use and to ease the code reading. for instance,

En ambos casos, primero se ejecutan los operadores relacionales, es decir se comprueba si  $a > b$  y se comprueba si  $c > d$ , por último se aplica el operador lógico `or` a los resultados. Pero en el segundo caso es más fácil ver lo que se pretende calcular, gracias al uso de paréntesis.

Veamos otro ejemplo en el que el uso de paréntesis es necesario para obtener el resultado correcto. Se trata de la función lógica *or exclusivo* o *XOR*. Se trata de una operación que compara dos variables lógicas de modo que el resultado solo es verdadero si uno de los operandos es verdadero pero el otro es falso. La tabla de verdad de la operación *XOR* toma la forma,

In both cases, Python first computes the relational operators, i.e., it checks if  $a > b$  and  $c > d$ . Then, it applies the logical operator `or` to the previous results. But, in the second case, it is easier to understand the operation's objective, thanks to the parentheses.

Let's see another example where the parentheses are mandatory: the logical operation *exclusive or* o *XOR*. It is an operation that compares two logical variables and yields true only if one of the operands is true and the other is false. the *XOR* operation true table is as follows,

Tabla de verdad de la operación `xor`  
Truth table for `xor` operation

operando 1 operand 1	operando 2 operand 2	resultado result
1	1	0
1	0	1
0	1	1
0	0	0

Esta operación no está definida como tal en Python pero se puede implementar usando las operaciones `and`, `or` y `not`:

$$\text{XOR} \equiv (\text{a or b}) \text{ and not}(\text{a and b}).$$

Dejamos como un ejercicio comprobar que esta combinación de operaciones cumple con la tabla de verdad de la operación *XOR* y que pasaría que retiramos los paréntesis.

This operation is not defined in Python but can be easily implemented using the logical operations `and`, `or` y `not`:

$$\text{XOR} \equiv (\text{a or b}) \text{ and not}(\text{a and b}).$$

We left it as an exercise to check that such a combination of logical operations fulfils the *XOR* true table and what happens if we remove the parenthesis.

## 2.3. Scripts en python

Hasta ahora, hemos manejado siempre Python desde la línea de comandos. Es decir, hemos introducido las instrucciones de Python en la ventana de comandos. Este modo de emplear el programa es poco eficiente, ya que exige volver a introducir todos los comandos de nuevo cada vez que queremos repetir un cálculo.

Python puede emplear ficheros de texto en los que introducimos un conjunto de comandos, los guardamos, y volvemos a emplearlos

## 2.3. Scripts in Python

So far, we have always dealt with Python using the command line. That is, we have constantly introduced instructions using the command line. This method is rather inefficient because we need to rewrite the commands again anytime we want to repeat the same operations.

With Python, you can effortlessly save a set of instructions in a text file and reuse them anytime you need to carry out the same calculation. This is the usual way to work in

siempre que queramos. Esta es la forma habitual de trabajar no solo de Python, sino de otros muchos entornos de programación. Un fichero que contiene código de Python recibe el nombre genérico de *Script*. Un Script de Python no es más que un fichero de texto que contiene líneas formadas por comandos válidos de Python. Lo habitual es que cada línea contenga un comando. El fichero se guarda con un nombre y la extensión .py. Por ejemplo: `miprograma.py`. El nombre del fichero, puede contener números y letras, y algunos caracteres especiales como la barra baja (\_). En general es no es aconsejable emplear nombres que incluyan espacios en blanco u otros caracteres especiales, ya que es posible que den problemas al intentar ejecutarlos.

### 2.3.1. El editor de textos de Spyder.

Podemos emplear un editor de textos cualquiera, que genere texto en ASCII, como por ejemplo el 'block de notas', para escribir nuestros programas. Sin embargo, si trabajamos en el entorno de Spyder, lo ideal es emplear su propio editor de textos.

El editor de textos de Spyder ocupa el panel izquierdo de la ventana del IDE, (figura 2.6). En la figura 2.8 se muestra una vista más detallada de este panel.

La figura nos muestra un panel con varias pestañas. Cada una corresponde a un archivo de texto que contiene un programa escrito en Python. En la pestaña se indica el nombre del archivo. El archivo visible en la figura muestra el nombre `untitled0.py`. Se trata de un archivo nuevo que acabamos de abrir para escribir un nuevo programa. Para crear un nuevo Script en Spyder basta pulsar el icono situado a la izquierda en la parte superior del panel del editor de texto o bien desplegar el menú *File* situado justo encima y seleccionar la opción *New file*.

Si nos fijamos en la figura 2.8, vemos que el archivo numera en la parte derecha las líneas de código. Además aparecen ya unas líneas escritas,

Las líneas que aparecen escritas por defecto en el archivo de texto, no son comandos

Python and many other programming environments. A Python code file is usually called a *Script*. Thus, a Python Script is a text file that contains lines of valid Python commands. The file is saved using a name and the extension .py. So a valid name would be, for example, `myprogram.py`. The file name may contain numbers, letters, and special characters, such as the underline (\_). Using names that include blanks or other special characters is not advisable as a general criterion because we may have problems running them.

### 2.3.1. The Spyder's text editor

We can use whatever text editor we want to write our programs, provided it generates text in ASCII format. For instance, we can use the classical 'notepad'. Nevertheless, If we are working with Spyder, we should use its text editor.

The Spyder's text editor is located in the left panel of the IDE window (figure 2.6. Figure 2.8 shows a more detailed view.

The figure shows a panel with several tabs. Each one belongs to a different text file containing a Python program. The file name is written in the tab. The file shown in the figure has the name `untitled0.py`. It is a new file we have just opened to write a new program. To create a new Spyder script, it is enough to click the left icon on top of the editor panel or unfold the File menu located just above and select the option *New file*.

Looking at figure 2.8, we see how the code lines are numbered on the right side of the file how the code lines are numbered. Besides, some code lines are included.

These lines, written by default in the text file, are not Python commands; they are comments, i.e. lines that supply information on the file content to the reader. Usually, these lines contain the program's purpose, the type of variables it uses, the date of creation, the author, etc. Python accepts two kinds of

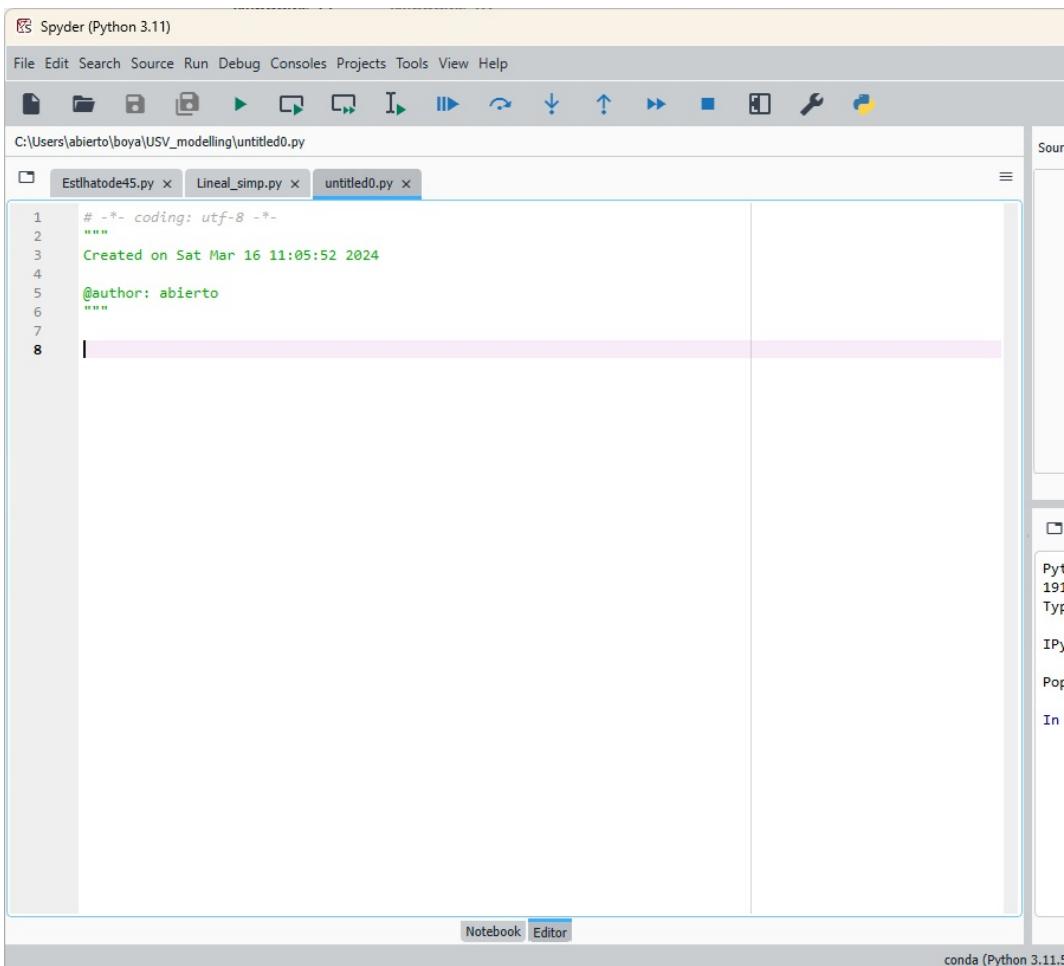


Figura 2.8: Editor de textos para Python incluido en Spyder  
Figure 2.8: Text editor for Python included in Spyder

de Python. Se trata de comentarios. Es decir, líneas en la que damos información a quien lee el fichero sobre su contenido, autor, fecha de creación etc. Python admite dos tipos básicos de comentarios:

- Líneas individuales cuyo primer carácter es una almohadilla `#`. Python interpreta que toda la línea es un comentario y se la salta.
- Grupos de líneas encerrados entre grupos de tres comillas `"""`. Se emplean para comentarios más largos, en los que se dan explicaciones más completas de lo

comments:

- Single comment lines. They start with a hash `#` symbol. Python considers the whole line a comment and skips it.
- A sequence of lines enclosed into groups of three quotation marks `"""`. They are used for lengthier comments, which contain longer descriptions of the code performance.

que hace el código.

---

```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Sat Mar 16 20:05:44 2024
4
5 @author: abierto
6 """
```

---

En nuestro caso, Spyder nos ha escrito un comentario de línea indicando la codificación con la que se guarda el texto del archivo (*utf8*). Después crea un área para un comentario del segundo tipo, (líneas 2 a 6 del código). Nos escribe la fecha en que se ha creado el archivo y el autor. Evidentemente estos valores pueden modificarse. Este espacio para comentarios creado por Python, es un buen sitio para que nosotros añadamos información sobre para qué sirve el fichero de código, el tipo e variables que define, etc.

Para seguir con el mismo ejemplo, vamos a escribir algunas líneas de código y vamos a guardar el fichero con un nombre nuevo. El fichero se guarda, bien pulsando el tercer ícono por la izquierda de los que se muestran encima del panel del editor de texto, o desplegando el menú File y pulsando la opción Save as. En nuestro caso, lo vamos a guardar con el nombre *ejemplo.py*.

In our case, Spyder has written a line comment showing the text codification (*utf8*) used to store the file's information. Below, Spyder creates a comment area –the second type of comment– using the code lines 2 to 6. There, it writes the file creation date and the name of the file author. Of course, this data can be changed by the user. This comment area created by Python is an excellent place to add information about the file features, the type of variables defined, etc.

We will continue with the example, adding some code lines and then save the file with a new name. To save a file, click the third icon from the left in the icon panel above the editor or unfold the File menu and select Save as. We will save our file with the name *ejemplo.py*.

---

```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Sat Mar 16 18:48:46 2024
4 Este Script es un simple ejemplo para mostrar el uso básico
5 de los Scripts de Python
6 This Script is a basic example just to show how to use Python's
7 Scripts
8 @author: abierto
9 """
10
11 #Primero creamos unas variables y mostramos su valor
12 #first we create some variables
13 a = [1,2,3,3,4,0,-8]
14 print(a)
15 b = 12
16 print(b)
17 c = [1,0,1,0]
```

```

18 print(c)
19 #Ahora realizamos algunas operaciones basicas con las variables
20 #Now we perform some basic operation with the variables created
21 a[3] = a[3] + 4.2
22 p = a.pop()
23 d = p+b-c[0]
24 #por ultimo mostramos los resultados de nuestras operaciones
25 #and eventually we show the results
26 print(a)
27 print(p)
28 print(d)

```

---

Si nos fijamos en los cambios introducidos, hemos añadido una líneas al comentario largo, explicando qué contiene el fichero. Además, hemos introducido comentarios de línea, explicando qué hace el código en cada paso. Es fácil ver que el programa crea dos listas **a** y **c** y una variable entera **b**. Además las imprime en el terminal para que veamos el valor que tomas. a continuación, suma 4.5 al valor del cuarto elemento de la lista **a**, extrae el último valor de dicha lista y lo guarda en una variable nueva **p**. En la línea 23 crea una nueva variable **d** sumando **p** y **b** y restando al resultado el último elemento de la lista **c**. Por último nos muestra el valor de las variables **a**, **p** y **d**.

### 2.3.2. Ejecución de scripts

Una vez que hemos terminado de escribir nuestro código, necesitamos ejecutarlo, para obtener los resultados del programa. Dentro del entorno de Spyder, la ejecución de un script se lleva a cabo pulsando el ícono señalado con un triángulo verde, situado en la parte superior del panel del script o bien seleccionando en el menú desplegable *Run* la opción *Run*.

Los resultados de la ejecución aparecen en la consola de Ipython,

```
In [4]: runfile('C:/Users/abierto/Documents/borrar/ejemplo.py',
              wdir='C:/Users/abierto/Documents/borrar')
[1, 2, 3.3, 4, 0, -8]
12
[1, 0, 1, 0]
[1, 2, 3.3, 8.2, 0]
-8
3
```

Focusing on the changes, we added some lines to the long comment explaining the file's contents. Besides, we have introduced some line comments now and then, describing what the code does in each step. It is easy to check that we have created two Lists **a** and **c** and an integer variable **b**. Moreover, python prints these new variables on the console to show their values. Then, it adds 4,5 to the current value of the fourth element of the list **a**, pops out the last values of this list and saves it in a new variable **p**. On line 23, a new variable, **d** is created, adding **p** and **b** and subtracting from the result the last element of the list **c**. Eventually, Python shows the values of variables **a**, **p** and **d**.

### 2.3.2. Running a script

Once we have finished our code edition, we need to run it to obtain the results. Using the Spyder environment, we run a script by clicking the green triangle icon located on top of the editor panel or unfolding the Run menu and selecting the option Run.

The results are shown on the Ipython console,

En realidad, al pulsar el icono de ejecución, lo que hacemos es ejecutar el comando `runfile()`, indicando entre paréntesis el fichero que queremos que se ejecute. Además, la consola de Ipython muestra los resultados de los comandos `print` que hemos definido en nuestro programa. A efectos prácticos, ejecutar el script es equivalente a escribir sus líneas de código una a una en la consola. De hecho, Python ha guardado en memoria las variables que hemos creado, de modo que podemos usarlas si introducimos nuevos comandos en la consola. Por ejemplo, podemos añadir un nuevo elemento al final de la lista `c`,

```
In [7]: c.append(31.2)
In [8]: print(c)
[1, 0, 1, 0, 31.2]
```

In [9]:

## 2.4. Funciones en Python

en la sección anterior hemos visto como emplear scripts para guardar nuestro código y poder reutilizarlo siempre que queramos. Aunque esto supone una mejora respecto a reescribir nuestros comandos de nuevo en la consola, todavía da lugar a código poco flexible y reutilizable. Vamos a introducir en esta sección una estructura de programación que va hacer de nuestro código una herramienta mucho más potente. Se trata de las funciones.

En Python podemos definir funciones directamente en la consola, pero esto no es, en general, una buena idea; cuando cerramos Spyder perdemos el trabajo hecho. Vamos por tanto a utilizar scripts para definir nuestras funciones.

La estructura general de una función en python toma la siguiente forma

```
def nombre_de_la_funcion(e1,e2,...):
    código de la función
    return, r1, r2, ...
```

La primera línea de la definición de una función recibe el nombre de cabecera. La cabecera de la función empieza siempre con la

When we click the icon Run, we execute the command `runfile()`, including, enclosing in parentheses, the name of the file we want to run. Then, Python shows the results of the `print` command that we have included in our program. From the point of view of the results, there is no difference between running the script or writing their lines one by one in the Ipython console. In fact, Python has saved the variables we created running the script in the computer memory, and we can use them if we introduce new commands in the console. For instance, we can add a new element at the end of list `c`,

```
In [7]: c.append(31.2)
In [8]: print(c)
[1, 0, 1, 0, 31.2]
```

In [9]:

## 2.4. Functions in Python

In the previous section, we learned about using scripts to save our code and reuse it later. While this is an improvement over repeatedly rewriting the same commands in the console, it still results in code that is not very flexible and reusable. In this section, we will introduce functions, a programming structure that enables our code to become a more powerful tool.

In Python, it is possible to define functions using the console. However, this is not advisable because all work will be lost once Spyder is exited. takes in Python the following general structure,

```
def funcion_name(e1,e2,...):
    function code
    return, r1, r2, ...
```

$$ax^2 + bx + c = 0$$

The first line of a function definition is known as the function header. It always starts with the keyword `def`. Following this, we write the name of the function. Valid function names follow the same rules as file names. After the function name, we enclose input variables

palabra clave `def`. A continuación se indica el nombre que tendrá la función. Los nombre válidos deben empezar por letras y en general no es buena idea emplear caractéres especiales. A continuación, entre paréntesis y separadas por comas se incluyen las variables de entrada, es decir, aquella variables que la función va a emplear para realizar sus cálculos. Si no necesita tomar variables de entrada, se deja es paréntesis vacío, pero no se omite. Por último se añade el símbolo ':' para indicar que se ha terminado de definir la cabecera de la función.

Deabajo de la cabecera, viene las líneas de código de la función, que constituyen el cuerpo de la misma. Una característica importante de Python es que todas las líneas de código pertenecientes a una función deben estar indentadas, es decir, deben empezar dejando un número igual de espacios en blanco al principio de todas las lineas. Una de las ventajas de emplear el editor de textos de Spyder es que, si definimos una línea de cabecera correctamente, y pulsamos la tecla intro, directamente nos indenta las líneas siguientes. La última linea duna función suele emplearse para devolver variables que contienen los resultados que desamos optener de las operaciones realizadas por la función. Empieza con la palabra clave `return` y va seguida de una lista de variables separadas por comas. Por supuesto dichas variables se tienen que haber definido antes en las líneas de código de la función. A continuación vamos a ver un ejemplo de una función construida para obtener las soluciones de una ecuación de segundo grado,

$$ax^2 + bx + c = 0$$

Cuyas soluciones toman la forma general,

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

in parentheses, separated by commas. These input variables are the variables that the function will use to perform its operations. When the function does not need input variables, we leave the parentheses empty but never omit them. Eventually, we write the symbol ':' to end the function header.

Beneath the header file comes the function body, that is, the code lines of the function. One important feature of Python language is that any line belonging to the body function should be indented. The function body lines should begin, leaving the same number of blank spaces from the beginning of the line. One advantage of using the Spyder text editor is that it automatically acknowledges the file header lines and indents the following lines. The last function line is usually used to recover variables that contain the function operations' relevant results. It starts with the keyword `return`, followed by a list of variables separated by commas. These variables should be defined in the function body.

Let's look at an example function for solving quadratic equations.

$$ax^2 + bx + c = 0$$

it has the following general solutions,

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

---

eq-seg-grado.py

---

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Mon Mar 18 18:33:43 2024
4 Este script contiene una función para obtener la solución de una
5 ecuación de segundo grado.
6 This script defines a function to solve a quadratic equation

```

```

7  @author: abierto
8  """
9  def soleq2(a,b,c):
10  """
11  Esta función calcula las soluciones de la ecuación general,
12  ax**2 + b*x + c = 0
13  Variables de entrada:
14  coeficientes a, b y c
15  Variables de salida:
16  xp Solucion para la raíz cuadrada positiva
17  xm solucion para la raiz cuadrada negativa
18
19  this function calculates the solutions of the general equation
20  a*x**2 + b*x + c = 0
21  input variables:
22  a,b and c coeficients
23  output variables;
24  xp solution using positive square root
25  xm solutions using negative square root
26  """
27  xp = (-b+(b**2-4*a*c)**(1/2))/2/a
28  xm = (-b-(b**2-4*a*c)**(1/2))/2/a
29  return xp, xm
30
31 #aquí añadimos un ejemplo de uso que no es parte de la función
32 #here we add a 'how to use' example that's not part of the function
33
34 solplus, solminus = soleq2(2, 3, 1)
35
36 #comprobacion
37 #checking
38 print('raíz positiva/positive root:', 2*solplus**2+3*solplus+1)
39 print('raiz negativa/negative root:', 2*solminus**2+3*solminus+1)

```

---

Para definir nuestra función hemos abierto un nuevo script en el editor de texto de Spyder y lo hemos guardado con el nombre `eq_seg_grado.py`. Evidentemente, se puede guardar con cualquier nombre, aunque es buena costumbre tratar de buscar uno que nos de alguna pista de qué contiene el script.

Tras, incluir un breve comentario sobre el contenido del fichero, hemos definido la cabecera de la función en la línea 9. Es interesante fijarse en la definición: el nombre de la función es `soleq2`, y ha continuación, entre paréntesis hemos definido como variables de entrada los tres coeficientes de la ecuación que queremos resolver. Cerramos la cabecera, añadiendo el símbolo 'dos puntos' final. Inmediantamente debajo, de la cabecera, hemos incluido

We have opened a new script in the Spyder text editor to define our function. We have saved the Script with the name `eq_seg_grado.py`. Obviously, you can save the file with whatever valid name you like, but it is a good idea to find a name that gives us some hints in the script content (`quad_eq.p` should be a nice name for this script).

After briefly commenting on the file's content, we define the function header on line number 9. It is interesting to look carefully at this definition: After writing the keyword `def`, we have given a name to the function `soleq2`, and, next, we have defined the input variables, enclosed in parentheses and separated by commas. These input variables will be the coefficients of the quadratic equation

un comentario amplio, describiendo qué hace la función cuáles son sus variables de entrada, y qué variables, con los resultados, nos va a devolver.

La línea 27 contiene las operaciones matemáticas para obtener la solución de la ecuación correspondiente a la solución con raíz cuadrada positiva y guarda el resultado en la variable `xp`, mientras que la línea 28 contiene la solución correspondiente a la raíz cuadrada negativa y guarda el resultado en la variable `xm`. ¡Compruébalo! Por último la línea 29 devuelve las variables `xp`, `xm` con los resultados calculados.

Aquí termina la definición de la función. Por eso, las líneas siguientes de código, que no forman parte de la función, no están indentadas. Las hemos añadido simplemente para mostrar cómo utilizar la función `soleq2` y comprobar que los resultados son correctos.

En la línea 34 se muestra cómo emplear (llamar) a la función que acabamos de crear. En primer lugar definimos dos variables en los que guardar los resultados `solplus` y `solminus`. Python asociará, por orden, estas variables separadas por comas, a las variables definidas tras la palabra clave `return`. Es decir, copiará el contenido de la variable `xp` en la variable `solplus` y el de la variable `xm` en la variable `solminus`. Evidentemente, podemos elegir cualquier nombre de variable válido para guardar los resultados. A continuación escribimos el símbolo de asignación `=`, seguido del nombre de la función. Por último, damos valores a las variables de entrada `a`, `b`, `c`.

Las líneas 38 y 39 comprueban que las dos raíces obtenidas satisfacen la ecuación de segundo grado,  $2x^2 + 3x + 1 = 0$ . Hemos añadido al comando `print`, un texto entre comillas para que nos indique qué resultado de las comprobaciones corresponde a cada solución obtenida.

Si ejecutamos el script que hemos creado, pulsando el icono con el triángulo verde, Obtenemos en la consola de Ipython el siguiente resultado,

```
In [1]: runfile('C:/Users/abrierto/boya/USV_modelling/eq_seg_grado.py',
wdir='C:/Users/abrierto/boya/USV_modelling')
raíz positiva/positive root: 0.0
```

we wish to solve. We close the function header, adding the final colon. Just beneath the header, we have added a rather lengthy comment explaining the purpose of the function and describing its input and output variables.

We have written the mathematical operations for the quadratic equation's positive and negative squared roots in lines 27 and 28, respectively. Check it! In the first case, we have saved the result in variable `xp` and, in the second case, we have saved the result for the negative squared root in variable `xm`. Lastly, line number 29 returns variables `xp`, `xm` with the operation results.

The function definition finishes here. The remaining code lines do not belong to the function, so they are not indented. We have added them to show how to use the function `soleq2` and check if the results are correct.

In line 34, we show how to use (call) the already created function. First, we define two variables, `solplus` and `solminus` for saving the function results. These variables, separated by commas, are orderly associated with those returned by the function,i.e. the variables included in the function code after the keyword `return`. Thus, Python will copy the content of variable `xp` in variable `solplus` and the content of variable `xm` in variable `solminus`. It is unnecessary to say that we can give whatever valid name we want to the variables we use to save the function results. After the variables above, we write the assignation symbol followed by the function name. lastly, we write values for input variables, `a`, `b`, `c`.

Lines 38 and 39 check that both obtained solutions fulfil the quadratic equation  $2x^2 + 3x + 1 = 0$ . We have added a quoted text to the `print` command that helps to determine which result is up to each solution.

If we click on the green triangle icon and run the script we have written, we get the following result in the Ipython console,

```
raiz negativa/negative root: 0.0
```

In [2]:

Es importante recordar que un script no es más que una secuencia de instrucciones que se ejecutan en el terminal. Por tanto, si seguimos la secuencia del script del ejemplo, lo primero que hemos hecho es *crear* la función que hemos definido y guardarla en la memoria del ordenador. Por eso, podemos ahora si queremos seguir utilizándola directamente en el terminal,

```
In [12]: coef = [3,-2,2]
In [13]: sp, sm = soleq2(coef[0],coef[1],coef[2])
In [14]: print(sp)
(0.333333333333334+0.7453559924999299j)
In [15]: print(sm)
(0.333333333333333-0.7453559924999299j)
In [16]: print(solplus)
-0.5
In [17]: print(solminus)
-1.0
```

En este caso, hemos agrupado los coeficientes de la ecuación de segundo grado que queremos resolver en una lista `coef`. Despues, llamamos a la función. El procedimiento es el mismo de antes, definimos unas variables de salida para guardar los resultados `sp`, `sn` escribimos el signo de asignación, el nombre de la función y damos valores a las variables de entrada. Tal y como las hemos escrito, la función identificará el coeficiente `a` con el primer elemento de la lista `coef[0]`, el segundo con coeficiente con el segundo elemento de la lista, etc.

A continuación empleamos el comando `print`, para observar los valores de las variables de salida. En esta caso, la ecuación no tiene soluciones reales, y el resultado son dos números complejos conjugados.

Las dos últimas líneas In[16] e In[17] las hemos empleado para imprimir el valor de las variables `solplus` y `solminus`. Esta variables se crearon cuando se ejecutó el script, pero Python las conserva todavía en memoria.

It is important to note that a script is just a sequence of instructions that run in the console. Thus, if we track the sequence of the example script, we first *create* the function defined in the script and save it in the computer memory. Hence, as long as we don't quit Spyder, we can keep on using it directly in the Ipython terminal,

Notice that we have collected the quadratic equation coefficients in a Python list in this example. Then, we call the function. The procedure is the same as before; we define output variables to save the results `sp`, `sn`, write the assignation symbol and the function name and give values to the input variables. These last have been written so that the function should identify the quadratic equation `a` coefficient with the first element of the list, the second coefficient with the second element of the list, etc.

Next, we use the command `print` to watch the output values of the output variables. In this case, the equation does not have real solutions, and the results are two complex conjugated numbers.

The last two lines, In[16] and In[17], are used to show the value of `solplus` and `solminus`. these variables were created when we ran the script, but Python still holds them in the computer memory.

## 2.5. Los *NameSpace* en Python y el ámbito de las variables

Acabamos de ver que python conserva en la memoria las variables creadas para emplearlas posteriormente. Es importante entender cómo python lleva a cabo la gestión de las variables creadas en memoria. Esto nos lleva al concepto de *namespace*. Por *namespace* entendemos un espacio de memoria, reservado por Python donde los programas y funciones puedes acceder para buscar variables, funciones y, en general, objetos de programación.

En Python se definen cuatro *namespaces* diferentes. Su definición es dinámica, es decir, tienen una duración en tiempo limitada: se crean cuando son necesarios y se destruyen una vez que una vez que dejan de ser necesarios. Habitualmente varios *namespaces* coexisten a la vez. Veamos cuales son:

**Built-in Namespace.** Contiene todas las variables y funciones propias de Python. Dichas variables son accesibles siempre que tenemos abierta una consola de Python. Podemos listar sus nombres escribiendo en la consola de Python el comando `dir(__builtins__)`. Por ejemplo, la función `print`, pertenece a este *namespace*. Podemos acceder a las funciones y variables contenidas en Built-in, desde cualquier función o programa de Python.

**Global namespace.** Podríamos decir que este *namespace* contiene todas las variables y funciones que hemos definido en un script. Sin embargo, para entender mejor a qué nos estamos refiriendo es preciso introducir el conceptos de importación. Volvamos por un momento al ejemplo que vimos más arriba en el que creamos un script que contenía una función `soleq2` que calculaba las soluciones de una ecuación de segundo grado. Hemos visto que si ejecutamos el script, en Spyder la función quedaba en la memoria del ordenador y podíamos emplearla para resolver cualquier ecuación de segundo grado directamente en la consola de Ipython.

## 2.5. Namespace and Scope in Python

[Variable! Namespace] We have seen that Python holds the created variables in the computer memory so that they can be used later. Understanding how Python manages the variables saved in the computer memory is essential. This takes us to the idea of Python *namespaces*. A *namespace* could be considered a memory space reserved by Python, where programs and functions can access to find variables, functions and, in general, programming objects.

Python defines four different *namespaces*. They are dynamically defined, meaning they last for a limited time. They are created when necessary and destroyed when they are no longer needed. Usually, several *namespaces* co-exist. They are:

**Built-in Namespace.** This namespace stores all Python Built-in variables and functions. We can access these variables whenever we have a Python terminal opened. We list their names by writing the command `dir(__builtins__)` on the Ipython terminal. For instance, the function `print` belongs to this *namespace*. We can access every function and variable stored in the Built-in namespace from whatever function or program we write in Python.

**Global Namespace.** We could say that this *namespace* contains all functions and variables defined in a script. Nevertheless, we will introduce the importation concept to understand better what we mean by this. Let's return to the example above, where we wrote a script with a function `soleq2`, allowing us to get a quadratic equation's roots. We have seen that if we run the script in Spyder, the function `soleq2` is stored in the computer memory, and we can use it straightforwardly on the Ipython terminal to solve any quadratic function.

But what if we want to use the function `soleq2` in another script? Do we need to write down the equation again in the new script? The answer is not. Python supplies a method

Pero, ¿Qué pasa si queremos emplear dicha función en otro script? ¿Debemos volver a escribirla de nuevo? La respuesta es no. Para emplear la función `soleq2`, podemos *importar* nuestro primer script en el segundo mediante el comando `import`. Veamos un ejemplo,

---

Example\_import.py

---

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Thu Mar 28 17:25:30 2024
4 Esto es un ejemplo para ver el comando import
5 This is an example on the use of the import command
6 @author: abierto
7 """
8 #importamos el script entero esto no siempre es una buena idea
9 #we import the whole script although not always it is a good idea
10 import eq_seg_grado
11 e = 2
12 f = -4
13 g = 1
14 solplus, solminus = eq_seg_grado.soleq2(e, f, g)
15
16 print('Example_import:', solplus)
17 print('Example_import:', solminus)
18
19 print('eq_seg_grado:', eq_seg_grado.solplus)
20 print('eqseg_grado:', eq_seg_grado.solminus)
```

---

La línea 10 de ejemplo `Example_import.py`, usa el comando `import` para *importar* el contenido del script `eq_seg_grado.py`. Importante: se escribe el nombre del archivo sin la extensión `.py`. Cuando se ejecuta esta línea de código, se ejecuta el script `eq_seg_grado.py` y se crea un namespace global que contiene todas las funciones y variables creadas por dicho script.

En las líneas 11 – 13 definimos tres variables, para usarlas como coeficientes de una ecuación de segundo grado y en la línea 14 llamamos a la función `soleq2` para resolver dicha ecuación. Esta función está contenida en el namespace global de `eq_seg_grado.py`, para acceder a ella escribimos:

`eq_se_grado.soleq2`

Es decir: primero indicamos el namespace y, separado por un punto, el nombre de la función. Esta es la forma en que podemos ac-

to access the functions and variables of one script from another. The method is known as *importation*; we import one script into another using the command `import`. The following script shows an example,

Line number 10 on `Example_import.py` example, uses the command `import` to *import* the content of script `eq_seg_grado.py`. Attention: we write the name of the file without the extension `.py`. When program line number 10 is executed, the script `eq_seg_grado.py` is thoroughly executed, and a global namespace is created to save the variables and functions created by this script.

In lines 11 – 13, we define three variables to be used as coefficients for a quadratic equation. Later, in line 14, we call function `soleq2` to solve the equation. But `soleq2` belongs to the global namespace created when the script `eq_seg_grado` was imported. So, to access it, we write:

`eq_se_grado.soleq2`

We indicate the namespace first and then the function name, separated by a point. This is how we access the variables and functions

ceder a cualquiera de las funciones o variables definidas en un script a través de otro script.

¿por qué definir las variables y funciones en namespaces separados? Una razón es que permite repetir nombres de variables y funciones en distintas partes de nuestro código sin que accidentalmente unos borren a otros. En el ejemplo anterior, en la línea 14 hemos guardo los resultados de las soluciones de la ecuación de segundo grado en las variables `solplus` y `solminus`. Pero estos nombres de variable ya existían. Hemos dicho que al importar `eq_seg_grado`, este script se ejecuta. Si revisamos sú código, vemos que en la línea 34 `solplus, solminus = soleq2(2, 3, 1)`, Se llama a la función `soleq2`, y se emplean exactamente los mismos nombres de variable para guardar los resultados. Aunque ambos resultados se guardan con los mismos nombres, pertenecen a namespaces distintos. Si ejecutamos el script `Example_import.py` obtenemos en el terminal de Ipython los siguiente resultados,

En términos generales, cuando se importa un script, lo que se crea es una estructura de programación, que en Python se conoce con el nombre de módulo. El módulo toma el nombre del script que lo genera, exceptuando la extensión. Para acceder a las funciones contenidas en el módulo, basta anteponer al nombre de la variable o función el nombre del módulo separado por un punto.

Si descontamos las funciones y variables definidas en el Built-in, todo el resto de código escrito en Python, está pensado para crear módulos. Lo habitual es que estos módulos se agrupen en paquetes de software que engloban funciones y objetos orientados a tareas específicas. A lo largo de estos apuntes, veremos algunos de ellos como `numpy`, orientado al cálculo científico o `matplotlib`, dedicado a la realización de gráficos.

defined in one script from another script.

Why is it convenient to define variables and functions in separated namespaces? One reason is that it allows repeating the same variable and function names in different locations of our code, avoiding them clashing accidentally. In the last example, in line 14, we saved the quadratic equation solutions in variables `solplus` y `solminus`. But these variable names have already been used. We said we were running it when we imported `eq_seg_grado`. If we review the code of this script we can see that in line 34

`solplus, solminus = soleq2(2, 3, 1)`, the program calls `soleq2` and the results are saved in variables with identical names, `solplus` y `solminus`. Although we are saving the solutions in both cases with the same names, they belong to different namespaces. After running the script `Example_import.py`, we achieve the following result,

Generally, a programming structure is created whenever we import a script. In Python, these program structures are called modules. A script and the module generated when the script is imported share the same name (except for the extension). To access the functions and variables held in the module, it is enough to put the name of the module before the name of the function or variable separated with a point.

Except for the functions and variables defined in the Built-in namespace, the remaining code written in Python is intended to create modules. Usually, the modules are grouped into software packages which enclose functions and programming objects devoted to specific tasks. In these lecture notes, we will meet some of them, like `numpy`, oriented to scientific computing or `matplotlib`, devoted to graphical representation.

```
In [1]: runfile('C:/Users/abierto/Documents/borrar/Example_import.py',
              wdir='C:/Users/abierto/Documents/borrar')
raíz positiva/positive root: 0.0
raiz negativa/negative root: 0.0
Example_import: 1.7071067811865475
Example_import: 0.2928932188134524
eq_seg_grado: -0.5
eqseg_grado: -1.0
```

In [2]:

Al importar `eq_seg_grado`, se ejecuta dicho script. Se calcula por tanto la solución de la ecuación de segundo grado:  $x^2+2x+3 = 0$  y se muestran en el terminal las comprobaciones de los resultados. Una vez terminada la importación, todas las variables y funciones creadas por `eq_seg_grado`, están disponibles en su namespace global. despues, observamos en el terminal los resultados de las dos ecuaciones de segundo grado resueltas. Las dos primeras, corresponden a la ecuación  $2x^2+3x+1 = 0$ , y las dos últimas a la ecuación  $x^2+2x+3 = 0$ . Para imprimir estas últimas, hemos empleado `eq_seg_grado.solplus`  
`eq_seg_grado.solminus`,

para indicar que nos referimos a las variables guardadas en el namespace de `eq_seg_grado`.

**Local Namespace.** Este *namespace* es propio de las funciones. Cuando se ejecuta una función en Python, se crea un namespace específico (local) para dicha función en la que se guardan las variables que la función crea y utiliza durante su ejecución. Una vez que la función termina, el namespace local asociado se destruye, y las variables dejan de estar accesibles.

Si nos fijamos en el código de la función `soleq2`, vemos que la función crea unas variables `a,b,c` en su cabecera. Dentro de la función se emplean dichas variables para resolver una ecuación de segundo grado, y se guardan las soluciones en las variables `xp,xm`. Estas cinco variables pertenecen al namespace local de la función `soleq2` y solo el código de dicha función puede acceder a ellas.

Cuando llamamos a la función `soleq2` tanto si lo hacemos desde un módulo como desde la línea de comandos, empleamos el signo de asignación para copiar las variables de salida y damos valores a las variables de entrada. Por ejemplo si nos fijamos en la línea 14 del módulo `Example_import.py`,

```
solplus, solminus = eq_seg_grado.soleq2(e, f, g)
```

As we have already said, this script is run when we import `eq_seg_grado`. Thus, the program calculates first the solution of the quadratic function:  $x^2 + 2x + 3 = 0$  and shows the checking of the solutions on the Ipython terminal. Once the importation is over, all created functions and variables are available in the script's global namespace. Then, we see the results of the two quadratic equations solved on the Ipython terminal. First, the solutions of  $2x^2 + 3x + 1 = 0$  and then the solutions of  $x^2 + 2x + 3 = 0$ . To print these last solutions, we use

```
eq_seg_grado.solplus  
eq_seg_grado.solminus,
```

To indicate that, we want to print the variables held in the global namespace `eq_seg_grado`.

**Local Namespace.** This *namespace* is used by Python functions. Whenever a function is executed, a specific namespace (local) is created for the function. Every variable created by the function is saved in its local namespace, and it is available for the function to use during execution. Once the function ends, its associated local namespace is destroyed, and the variables are no longer accessible.

Focus on function `soleq2` code, we can see how the function creates variables `a,b,c` in its header. In the function body, these variables are used as the coefficients of the quadratic function to be solved. The solutions are saved in variables `xp,xm`. These five variables belong to the local namespace of the function `soleq2`, and only this function code can access them.

When we call function `soleq2`, whether from another module or the command line, we use the assignation symbol to copy the output variables and give values to the input variables. For instance, recall the line number 14 of the `Example_import.py` module,

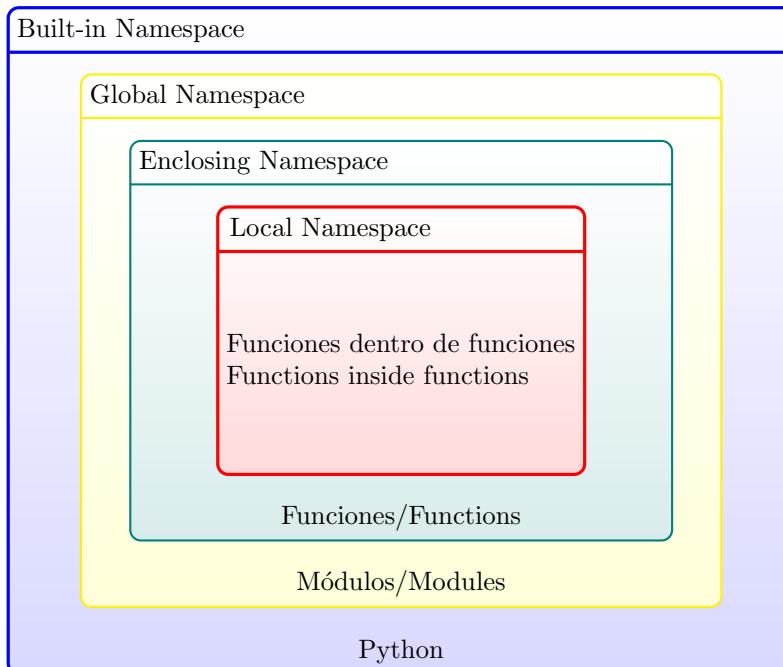


Figura 2.9: Relación de inclusión entre los Namespaces de Python  
 Figure 2.9: Inclusion relationship among Python Namespaces

El contenido de las variables `e,f,g`, que pertenecen al namespace global del módulo `Example_import.py` en qué se llama a la función, se copian en las variables `a,b,c` que pertenecen al namespace local de la función. Cuando se ejecuta el comando `return` el valor de las variables `xp,xm` que pertenecen al namespace local de la función, se copian en las variables

`soplus, solminus`

que pertenecen al namespace global del módulo `Example_import.py`.

**Enclosing namespace.** En ocasiones, llamamos a una función desde dentro de otra función. Por ejemplo nuestra función `soleq2`, llama a la función `print`, que pertenece al namespace Built-in the Python. Cuando se ejecuta la función `soleq2`, ésta crea su propio namespace, y cuando dentro de ésta se ejecuta la función `print` esta última también crea su propio namespace local, dentro del namespace de la función `soleq2`. Se dice entonces que el namespace local de `soleq2` *encierra* al

When the function is called, variables `e,f,g`, which belong to the global namespace of the modules `Example_import.py` are copied in variables `a,b,c`, which belong to the function local namespace. When the function at the end executes the command `return`, the values of variables `xp,xm`, which belong to the local namespace of the function, are copied in variables

`soplus, solminus`

, which belong to the global namespace of module `Example_import.py`.

**Enclosing namespace.** Sometimes, we call a function from inside another function. So, our function `soleq2` calls to función `print`, which belongs to the Python Built-in namespace. When function `soleq2` is executed, it creates its own namespace and when inside it function, `print` is executed this last, also creates its own local namespace inside the namespace of `soleq2`. The local namespace of `soleq2` *encloses* the namespace of `print`. In general, when a fun-

namespace de `print`. En general, cuando una función llama a otra, El namespace de la primera se considera el Enclosing namespace de la segunda. El porqué de esta definición se entiende mejor explicando el concepto de ámbito de una variable.

**Ámbito.** Acabamos de describir el concepto de *namespace* como una región de memoria, asociada a un módulo o a una función. La existencia de distintos namespace, hace que no todas las variables y/o funciones puedan ser manipuladas desde cualquier lado. El ámbito de una variable o función define las funciones, módulos, etc desde lo que es posible acceder y manipular dicha variable o función. Los ámbitos, están fuertemente relacionados con los namespaces y en python se cumple una regla general conocida como la regla "LEGB"(Local-;Enclosing-;Global-;Built-in). Nos da, el orden de precedencia de los namespaces. Así, una función tiene acceso a todas sus variables y funciones locales (Local namespace) a todas las de la función que la ha llamado, si es el caso (Enclosing namespace), a todas las del módulo que ha llamado a la función (Global namespace) a todas las del Built-in namespace. También se cumple lo contrario: desde la línea de comando solo se tiene acceso a las variables que se han creado en la propia línea de comando. Desde un módulo se tiene acceso a las variables y funciones built-in y pero no a las de las funciones contenidas en el módulo. Desde una función se tiene acceso al contenido Built-in y al del módulo y, en su caso, la función, que contiene la función, pero no al de las funciones, contenidas o llamadas en dicha función. La figura 2.9, muestra un esquema con el anidamiento de los namespaces en Python.

**Más sobre el comando `import`.** El comando `import`, es quizás de los más empleados en Python. Más arriba hemos descrito su uso para importar un módulo entero en un script. Sin embargo, es posible emplearlo de otras muchas maneras:

```

1 import eq_sec_grado
2 import eq_sec_grado as sec
3 from eq_sec_grado import *
```

ction calls another, the namespace of the first function is considered the Enclosing namespace of the second function. The reason for this name may be better understood by explaining the concept of variable scope.

**Scope** We have described the concept of *namespace* as a memory area linked to a Python module or function. The namespaces make sure that not all variables or functions can be handled from everywhere. The scope of a variable or a function defines which functions, modules, etc., can manipulate such variable or function. The scopes are tightly related to namespaces. There is the rule known as the "LEGBrule (Local-;Enclosing-;Global-;Built-in)" that establishes the namespace precedence order. So, a function has access to all its local variables and function (Local Namespace), to all ones belonging to the function that calls it, if any (Enclosing namespace), to all ones belonging to the module that calls the function (global namespace) to all builtin namespace variables and functions. It is also true that if we follow the opposite direction from the command line, we can only access built-in variables and variables created from the command line itself. From a module, we have only access to variables and functions built-in or created from the module but not to variables belonging to functions contained in the module. From a function, access is granted to built-in variables, those belonging to the module that contains the function, those belonging to the function that includes the function, if any, and those belonging to the function itself. However, a function has no direct access to variables belonging to functions contained in or called from the function. Figure 2.9 shows a schematic view of Python namespace nesting.

**More on `import` command.** The `import` command is widely used in Python programming. We have already described how to use it to import a whole module. Nevertheless, it is possible to use it in very many different ways,

```
4 from eq_sec_grado import soleq2
5 from eq_sec_grado import soleq2 as cuqui
```

En todos los casos, el módulo desde el que se realiza la importación es el mismo, pero el modo de hacerlo es lo que cambia. Así en la línea 1 del ejemplo importamos el módulo entero, la forma de acceder a las variables y funciones del módulo, es a través de su namespace global: `eq_sec_grado.soleq2`. En la línea 2, también estamos importando el módulo completo, pero ahora, usamos un 'alias', `sec`, para el nombre del namespace asociado. Esto quiere decir que para acceder a las variables y funciones del módulo debemos ahora emplear el alias en lugar del nombre del módulo: `sec.soleq2`. En la línea 3 no importamos propiamente el módulo. Lo que hacemos es importar todas las funciones y variables contenidas el el módulo. De este modo no tenemos ya que anteponerles el nombre del módulo para usarlas. Es muy importante ser muy cuidadoso cuando se hace una importación así. Si en el script que ha importado el módulo hay funciones y/o variables que cuyo nombre coincida con alguno de los importados, tendremos un conflicto de nombres y solo prevalecerá el último creado. En el número 4 solo hemos importado la función `soleq2`. Podemos emplearla en el módulo de destino sin necesidad de hacer referencia a su módulo de origen. Por último, en número 5 del ejemplo hemos importado tan solo la función `soleq2`, pero además le hemos dado un alias. Esto quiere decir que en el modulo de destino deberemos emplearla usando el alias en lugar del nombre original de la función. Es decir, en lugar de utilizar `x1,x2 = soleq2(1,-3,2)` deberemos usar `x1,x2=cuqui(1,-3,2)`.

## 2.6. Depuración

Siempre que escribimos un programa, es preciso comprobar su funcionamiento y, en muchos casos, corregir los errores cometido. El proceso de corrección de código desde su versión original hasta la versión definitiva se conoce con el nombre de depuración de código. Podemos distinguir dos tipos de errores:

The module from which the import takes place is the same in every case, but not how the import is carried out. So, in the line number 1 of the example, we import the module as a whole. The method to access the variables and function of the module is by using its global namespace `eq_sec_grado.soleq2`. In line 2, we are also importing the whole module, but now, we use an 'alias', `sec` instead of the module name. This means that to access variables or functions belonging to the module, we use the alias: `sec.soleq2`. In line 3, we are not importing the module as it is. We are just importing all variables and functions included in the module. In this case, we no longer need to put before the module name to access them. But we should be careful using this importing method. If the script which has imported the module has variables or functions whose names coincide with those of the variables or functions imported, the names will clash. Only the last created ones will be accessible. In name 4, we only imported function `soleq2` from the module. We can use it in the destination script without putting it before the origin module name. lastly, in line 5 we have imported only function `soleq2` but this time we have assigned it an alias. This means we have to use the alias instead of the proper function name when we use it in the destination module. That is: we have to write `x1,x2=cuqui(1,-3,2)`, instead of `x1,x2 = soleq2(1,-3,2)`.

## 2.6. Debugging

Whenever we write a program, we must check its performance; often, we must fix the errors. The process that takes us from the original program version to the final one is known as code debugging. We could differentiate two main groups of errors:

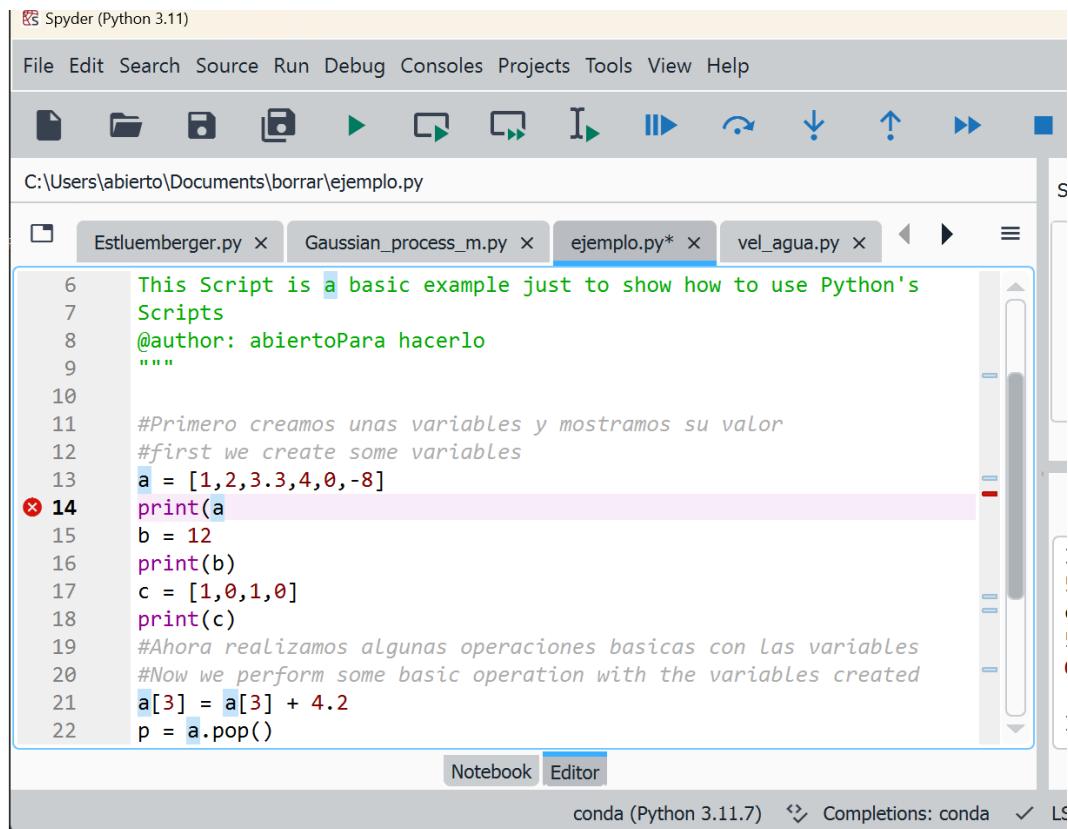


Figura 2.10: Un ejemplo de error de sintaxis (falta cerrar un paréntesis)  
 Figure 2.10: A syntax error example (a closing bracket has been omitted)

**Errores de sintaxis.** Normalmente son errores de escritura. Hemos escrito mal el nombre de una función o un comando o bien no hemos escrito correctamente el código siguiendo las reglas del lenguaje. El editor de Spyder advierte directamente de estos errores. Además, si se intenta ejecutar el código, en la consola de Ipython aparecerá un mensaje de error.

Como ejemplo veamos los errores del script contenido en el editor de textos de la figura 2.10. Si nos fijamos, el editor a dibujado delante de la línea 14 un círculo rojo con un aspa en el centro. Es la manera que tiene de indicarnos que en esas líneas hay errores de sintaxis. Si pasamos el ratón por encima del círculo rojo, obtendremos un mensaje con información sobre el error cometido.

Hay un segundo tipo de errores de sintaxis, que están relacionados con las propiedades de

**Syntax errors.** Broadly speaking, these are *spelling* mistakes. We have written something wrong, like a variable or a function name. Also, we make a syntax error if we do not follow the language rules. The Spyder editor warns us in either case. If we ignore the editor signals and run the program, we will get an error message in the Ipython console.

For example, figure 2.10 shows a script with several syntax errors. The editor has drawn a bold red circle with a cross in the centre before the lines containing an error. If we put the mouse atop the red circles, we get a message with some information on the error made.

There is a second type of syntax error that the editor does not detect. It is related to the properties of variables and functions. An error message will be cast on the Ipython console if we try to run the program. For instance, if we

las funciones y variables que empleamos. Estos errores no son detectados directamente por el editor, pero arrojan tambien un mensaje de error en la consola de Ipython cuando los cometemos. por ejemplo, si escribimos las dos siguientes líneas de código,

```
In [88]: a = 3
In [89]: a.pop()
Traceback (most recent call last):
Cell In[89], line 1
    a.pop()
AttributeError:
'int' object has no attribute 'pop'
In [90]:
```

Obtenemos un error en que se nos dice que no podemos extraer elementos de una variable entera, como es el caso de la variable `a=3`, creada. Efectivamente, solo podemos extraer elementos de una lista.

**Errores de codificación.** Este segundo tipo de errores son mucho más difíciles de detectar. El código se ejecuta sin problemas, pero los resultados no son los esperados. Ante esta situación, no queda más remedio que ir revisando el código, paso a paso para detectar donde está el error. Normalmente, este proceso se lleva a cabo con la ayuda de programas específicos llamados depuradores.

El editor de texto de Spyder nos permite ejecutar un programa paso a paso, ver los valores que van tomando las variables etc, mediante el depurador que lleva incorporado. Para ello, se definen en el editor de Spyder *breakpoints*, esto es: líneas en las cuales Python detendrá la ejecución de un programa, entrará en modo de depuración y esperará instrucciones del usuario. La figura 2.11 muestra el código del ejemplo de la ecuación de segundo grado, en el que se ha definido un *breakpoint*, pulsando con el ratón sobre el espacio situado a continuación del número de la línea en que se desea introducir el *breakpoint*. Spyder indica que el *breakpoint* está activo, dibujado un círculo rojo. Alternativamente, también es posible establecer o remover *breakpoints* empleando el botón *Set/Clear Breakpoint* de la pestaña *Debug*, situada encima del editor de

write the two following code lines,

We get an error message telling us that we cannot extract an element from an integer variable, as is the case of the variable `a=3`. For sure, you can only 'pop' an element from a list.

**Codding errors.** Errors of this kind are usually more challenging to find out than Syntax errors. The code runs correctly, but we do not get the expected results. The only way to find the bug is to review the code line by line. Fortunately, there are specific programs that can carry out this debugging task. They are called, well, yes, debuggers.

Thanks to its built-in debugger, the Spyder text editor allows us to run a program line by line, watching the values the variables are taking, etc. To use the debugger, we need to define in the Spyder editor *breakpoints*, i.e., lines in which Python will stop the program, get into debugging mode, and wait for the user's instructions. Figure 2.11 shows the code of the quadratic equation example, in which we have defined a *breakpoint*, pushing the mouse left button on top of the space after the number of the line at which we want to define a *breakpoint*. Spyder marks the active *breakpoint*, drawing a bold red circle in the space after the line number. It is also possible to set or remove *breakpoints*, using the *Set/Clear Breakpoint* button located in the tab *Debug* on top of the text editor.

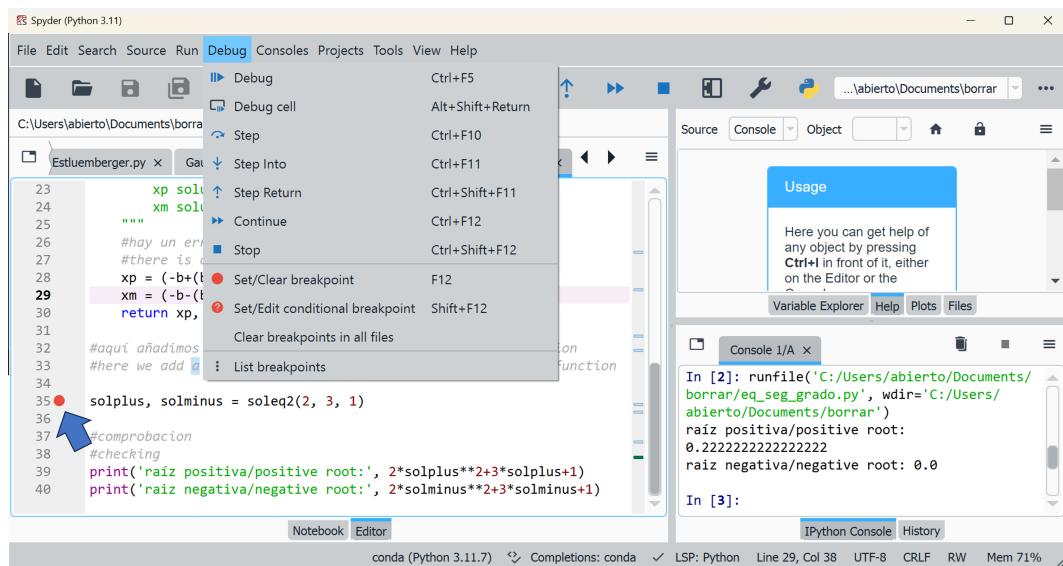


Figura 2.11: Breakpoint activo señalado con una flecha azul, y menú desplegable *debug*  
Figure 2.11: Active Breakpoint pointed by a blue arrow and *debug* drop-down menu

textos.

Si una vez señalado el *breakpoint*, entramos en el modo de depuración, Pulsando el primero de los botones azules en la parte superior del editor de textos (ver figura 2.12), Spyder nos indica que ha detenido la ejecución del programa en la línea marcada por el *breakpoint* (línea 35 en el ejemplo). Además, en el terminal de Ipython nos indica también que ha entrado en el modo de depuración,

Suppose after activating the *breakpoint*, we start the debugging mode by pressing the first of the blue buttons located on top of the text editor (see figure 2.12). In that case, Spyder shows us that it has stopped the program run at the line marked by the *breakpoint* red dot (line 35 in the example). Besides, the Ipython console also shows that it has come into debugging mode.

```
IPdb [1]: !continue
> c:\users\abierto\documents\borrar\eq_seg_grado.py(35)<module>()
 33 #here, we add a 'how to use' example that's not part of the function
 34
2--> 35 solplus, solminus = soleq2(2, 3, 1)
 36
 37 #comprobacion
```

IPdb [2]:

A partir de aquí Spyder pone a nuestra disposición las herramientas de depuración, la figura 2.12 muestra la línea en que se ha parado la ejecución del programa, (señalada con una flecha verde), y algunas de estas herramientas. Se trata del mismo módulo de ejem-

Now, we can access Spyder debugger's tools. Figure 2.12 shows the line where the program execution has been stopped (marked with a green arrow) and some of these tools. It is the same modules used in previous examples, but we have added a small error in line 28 this

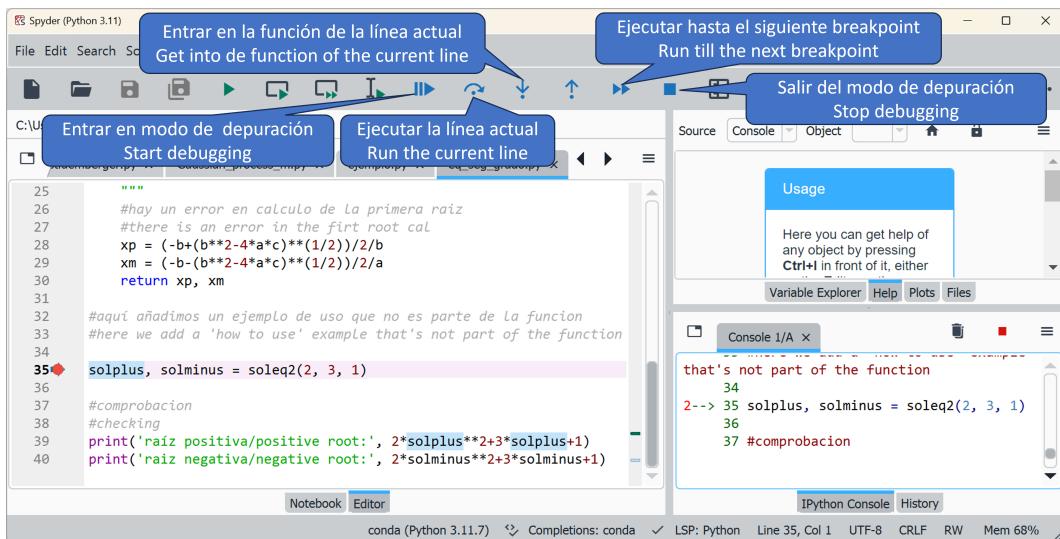


Figura 2.12: Parada de programa en un breakpoint y herramientas de depuración  
Figure 2.12: program stop on a breakpoint and debugging tools

los anteriores, en el que hemos introducido un pequeño error en la línea 28. Básicamente, Spyder nos da la posibilidad de ejecutar el código línea a línea, de entrar e ir línea a línea en las funciones que llama nuestro programa, o de continuar la ejecución hasta el final del programa o hasta el siguiente *breakpoint*.

Si pulsamos el botón “ejecutar línea actual” Spyder ejecutará la línea de programa señalada con la flecha verde y se parará en la línea siguiente. En cada paso, podemos ver el valor que toman las variables, pidiendo su valor directamente en el terminal de Ipython. En el ejemplo de la figura, el breakpoint está situado en la línea 35 que corresponde a la llamada a la función `soleq`. Si volvemos a pulsar el “ejecutar la línea actual”, el depurador ejecutará la función y saltará directamente a la linea 39. Si queremos que recorra paso a paso el código correspondiente a la función, empleamos el el botón “entrar en la función de la línea actual”. El uso del resto de las opciones del depurador el bastante evidente (ver figura 2.12). Si entramos en la función, ejecutamos su código línea a línea y vamos comprobando el resultado que deberían dar, comprobaremos que la línea 28 no da un resultado correcto y estaremos en condiciones de repasar su código

time. Spyder allows us to run the program line by line, get into the function our program calls to and cover it line by line, or continue the program execution till the next breakpoint or to the program end.

If we select the button “Run the current line,” the program will execute the code line marked with the green arrow and stop in the following line. In each step, we can check the values the variables so far defined take by asking for them at the Ipython terminal. In the example shown in the figure, the breakpoint is located in line 35, where the program calls to `soleq` function. If we push the button “Run the current line” again, the debugger will execute the function and jump to line 39. If we want to cover line by line the function `soleq`, then we may push the button “Get into the function of the current line”. The performance of the remaining debugger tools is easy to understand, looking at figure 2.12. If we get into the function `soleq` and execute its code line by line, checking their results, we will realise that there is a bug in line 28 because the result cast by this line is wrong. So we would be ready to review its code and discover the error. Can you find it?

y descubrir el error. ¿Cuál es?.

## 2.7. Control de Flujo

En la sección anterior, se introdujo el modo de escribir programas en Python mediante el uso de *scripts* y funciones. En todos los casos vistos, la ejecución del programa empezaba por la primera línea del programa, y continuaba, por orden, línea tras línea hasta alcanzar el final del programa. Se trata de programas en los que el *flujo* es lineal, porque los resultados de cada línea de programa se van obteniendo regularmente uno detrás de otro.

Hay ocasiones en las que, por diferentes razones que expondremos a continuación, puede interesarnos alterar el orden en que se ejecutan las sentencias de un programa, bien repitiendo una parte de los cálculos un determinado número de veces o bien ejecutando unas partes de código u otras en función de que se satisfagan unas determinadas condiciones.

El control del orden en que se ejecutan las sentencias de un programa es lo que se conoce con el nombre de *control de flujo*. Veremos dos tipos principales de control de flujo: El flujo condicional y los bucles.

### 2.7.1. Flujo condicional.

Empezaremos con un ejemplo sencillo de cómo y para qué condicionar el flujo de un programa. Supongamos que queremos construir un programa que reciba como variable de entrada un número cualquiera y nos muestre un mensaje por pantalla si el número es par.

Para ello, podríamos hacer uso del operador, *resto de la división entera*, %. Si el resto de la división entre dos del número suministrado a la función es cero, se trata de un número par; si no, es un número impar. Podríamos hacer uso de operadores relacionales, en particular de == para comprobar si el resto de la división entre dos es cero. Por último necesitaríamos algún mecanismo que permitiera al programa escribir un mensaje solo cuando el número introducido sea par.

## 2.7. Flow control

In the previous section, we introduced how to write Python programs using scripts and functions. In all the examples shown, the programs ran from the first line and went on, in an orderly manner, line after line, until the program ended. These programs follow a linear *flow* because the results of each line are obtained sequentially, one after another.

Many times, in situations we will describe later, we would like to modify the order in which the program sentences are executed. Sometimes, we need to repeat a block of code several times, while other times, we want a chunk of code to be executed only when a predefined condition is fulfilled.

The control of the order a program follows when executing its sentences is known as *flow control*. We will describe two main kinds of flow control: Conditional flow and loops.

### 2.7.1. Conditional flow.

We begin with a simple example of how and why conditioning the flow. Suppose we want to make a program that gets a number as input and shows us a message on the screen if the number is even.

We can use the operator *modulus*, %. If we divide the number by two and the remainder is zero, the number is even; otherwise, it is odd. We may use relational operators, specifically the == operator, to check if the division by two remainder is zero. Eventually, we need some method which allows the program to write a message only if the input number is even.

**if-elif-else.** The Python structure **if** supplies the method we need. But, let's see first the code of the example we are talking about,

**if - elif - else.** El mecanismo que necesitamos nos lo suministra la estructura `if` de Python. Veamos en primer lugar el código del ejemplo del que venimos hablando,

---

numero\_par.py

---

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3 """
4 Created on Fri Apr 19 15:31:19 2024
5 Script para crear una función que comprueba si un número es par
6 Script to create a function tha checks if a number is even
7 @author: juan
8 """
9
10 def par(n):
11     """
12     Esta función toma como entrada un número n. Si n es par devuelve un mensaje
13     indicando que el número es par.
14     This funtion takes an integer number n as input. If n is even then
15     it shows a message indicating the number is even.
16     """
17     if n%2 == 0:
18         print(str(n), ' es un numero par')
19         print(str(n), 'is an even number')
20     return()

```

---

El script genera una función `par(a)`. En la línea 17 aparece definida la entrada a una bloque `if`. Empieza con la palabra clave `if` que va siempre seguida de una expresión que da un resultado lógico: verdadero (1) o falso (0). Esta expresión puede ser cualquier combinación válida de expresiones relacionales o lógicas.

En nuestro caso comparamos el resto de la división del número introducido entre dos con cero. La entrada del bloque constituye una condición, y termina siempre con el símbolo “dos puntos” (:). El cuerpo del bloque mininlinepythonif lo constituyan las líneas posteriores a la línea de entrada, que deben estar sangradas. Estas líneas solo se ejecutan si se cumple la condición de la entrada del bloque `if`. En nuestro caso, el cuerpo lo constituyen las líneas 18 y 19. Por último, la línea 20 corresponde al comando `return()` que está alineado con las líneas de código de la función, no con las de bloque mininlinepythonif. Por tanto, en nuestro ejemplo, si el número introducido no es par, la función se saltará todas

The script generates a function `par(a)`. In line 17, we find the definition of an `if` opening. It starts with the keyword `if` followed by an expression that casts a logic result: true (1) or false (0). This expression could be any valid combination of relational or logical expressions.

In our example, we are comparing the remainder of the division of the input number (`n`) by two with zero. The `if` block opening line always makes up a condition and always ends with a colon (:) symbol. The body of the `if` block comprises the code lines that follow the opening line, which must be indented. The program executes these lines only if the condition defined in the block opening line is met. In our case, the `if` block body comprised lines 18 and 19. Eventually, in line 20, the command `return()` closes the function. Notice how this line is not indented like the lines of the `if` body. It is instead aligned with the lines of the function body because it is not part of the `if` block. Thus, if we introduce an odd number

las líneas del cuerpo del bloque `if` e irá directamente a la línea 20.

En las líneas de código siguientes se muestra un ejemplo del funcionamiento de la función `par`. Para probarla hemos importado directamente la función en el terminal de Ipython, y luego la hemos llamado dándole como variable de entrada primero el número 3 y luego el número 8. En el primer caso no se cumple la condición impuesta, ya que el número no es par, por tanto la función se salta todo el bloque condicional, y ejecutamente directamente el comando `return()`. Como no hemos pedido que devuelva nada, la función no devuelve nada. En el segundo caso, el número es par y por tanto si se cumple la condición necesaria para ejecutar el bloque `if`. Como resultado, la función imprime en el terminal de Ipython el mensaje indicado. Una vez que acaba la ejecución del bloque condicional, ejecuta el comando `return()` igual que en el caso anterior.

```
In [2]: from numero_par import par
In [3]: par(3)
Out[3]: ()
In [4]: par(8)
8  es un numero par
8  is an even number
Out[4]: ()
```

Acabamos de ver la estructura condicional `if` más sencilla posible. Podríamos complicarla un poco pidiendo que también nos saque un mensaje por pantalla cuando el número sea impar. Esto supone incluir en nuestro programa una disyuntiva; si es par el programa debe hacer una cosa y si no, debe hacer otra. Para incluir este tipo de disyuntivas en una estructura `if`, se emplea la palabra clave `else`. Veamos nuestro ejemplo modificado,

into the function, it will skip the lines of the `if` block body and will go straight to line 20.

Below are some code lines that demonstrate the function `par` performance. To test it, we imported the function to the IPython terminal and called it twice - first with the input variable of 3 and then with the input variable of 8. When we passed 3, the condition was not met because the number was odd. Therefore, the function skipped the conditional block and directly executed the `return` command. Since we didn't ask the function to return anything, it returned an empty bracket. On the other hand, when we passed 8, the number was even, and it fulfilled the imposed condition to execute the `if` block. As a result, the function printed the stated message in the IPython terminal. Once the `if` block has been completed, the program executes the command `return`, as in the even number case.

We have seen the simplest conditional `if` structure. We could make it a bit more complex, also asking for a message when the input number is odd. This means we want to include a disjunctive in our program; if the number is even, the program should do one thing; if the number is odd, it should do another one. To include this kind of disjunctive in an `if` structure, we use the keyword `else`. We can see below our example after being modified,

---

numero\_par\_mod.py

---

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Created on Fri Apr 19 15:31:19 2024
5  Script para crear una función que comprueba si un número es par
6  Script to create a function tha checks if a number is even
7  @author: juan
8  """
9
```

```

10 def par(n):
11     """
12     Esta función toma como entrada un número n. Si n es par devuelve un mensaje
13     indicando que el número es par.
14     This function takes an integer number n as input. If n is even then
15     it shows a message indicating the number is even.
16     """
17     if n%2 == 0:
18         print(str(n), ' es un numero par')
19         print(str(n), 'is an even number')
20     return()
21
22 def par2(n):
23     """
24     Esta función toma como entrada un número n y muestra un mensaje
25     indicando si el número es par o impar
26     This function takes an integer number n as input and shows a message
27     stating whether the number is even or odd
28     """
29     if n%2 == 0:
30         print(str(n), ' es un numero par')
31         print(str(n), 'is an even number')
32     else:
33         print(str(n), ' es un numero impar')
34         print(str(n), 'is an odd number')
35     return()

```

---

Hemos añadido a nuestro script una segunda función a la que hemos llamado `par2`. El código es idéntico al de la función `par`, hasta que llegamos a la palabra clave `else`, que marca la disyuntiva; si el número es par, el programa ejecuta las líneas de código entre el `if` y el `else`, si el número no es par ejecutará las líneas situadas debajo del `else` y que aparecen indentadas. En cualquiera de los dos casos, ejecutará la línea 35 que no pertenece al bloque condicional.

Las siguientes líneas de código muestran los resultados de aplicar la función `par2` a los números 3 y 8.

We have added a second function called `par2` to our script. The code is identical to the `par` function until we reach the `else` keyword. At that point, the disjunctive is set. If the input number is even, the program will execute the code lines between the `if` and `else`. Otherwise, it will execute the indented lines beneath the `else`. In both cases, the program will execute line 35, which does not belong to the conditional block.

The following code lines show the result of applying the function `par2` to the numbers 3 and 8.

In [11]: `from numero_par_mod import par2`

```

In [12]: par2(3)
3  es un numero impar
3  is an odd number
Out[12]: ()

```

```
In [13]: par2(8)
8 es un numero par
8 is an even number
Out[13]: ()
```

La estructura `if` admite todavía ampliar el número de posibilidades de elección mediante la palabra clave `elif`. Al igual que con `if`, `elif` va seguido de una expresión lógica que establece una condición, si se cumple se ejecutará el código de las líneas siguientes, si no se cumple, el programa saltará a la siguiente línea que contenga una palabra clave: otro `elif`, un `else` o directamente a la siguiente línea no indentada, saliendo así del bloque condicional. Para ver cómo funciona, vamos a modificar nuestro ejemplo anterior, para que, si el número introducido no es divisible por dos, compruebe si es divisible por tres,

The `if` structure allows to expand the number of possible elections even more, using the keyword `elif`. As in the case of the keyword `if`, `elif` is followed by a logical expression which establishes a condition. The following indented code lines will be executed if the condition is fulfilled. Otherwise, the program will jump to the next line with a keyword: another `elif`, an `else` or straightforwardly to the next non-indented line, exiting the conditional block. We are going to see how it works, modifying our last example. Now we want the program to check when the input number is odd, whether it is divisible by three,

---

numero\_div2\_div3.py

---

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Created on Fri Apr 19 15:31:19 2024
5  Script para crear una función que comprueba si un número es par y si no lo es
6  comprueba si es divisible por tres
7  Script to create a function tha checks if a number is even
8  @author: juan
9  """
10
11
12 def partthree(n):
13     """
14     Esta función toma como entrada un número n y muestra un mensaje
15     indicando si el número es par si no comprueba si es divisible por 3
16     This function takes an integer number n as input and checks if the
17     number is even. If not it checks if the number is divisible by 3
18     """
19     if n%2 == 0:
20         print(str(n), ' es un numero par')
21         print(str(n), 'is an even number')
22     elif n%3 ==0:
23         print(str(n), ' es un numero divisible por tres')
24         print(str(n), 'is a number divisible by three')
25     else:
26         print(str(n), ' es un numero impar pero no divisible por tres')
27         print(str(n), 'is an odd number but it not divisible by three')
28
29 return()
```

---

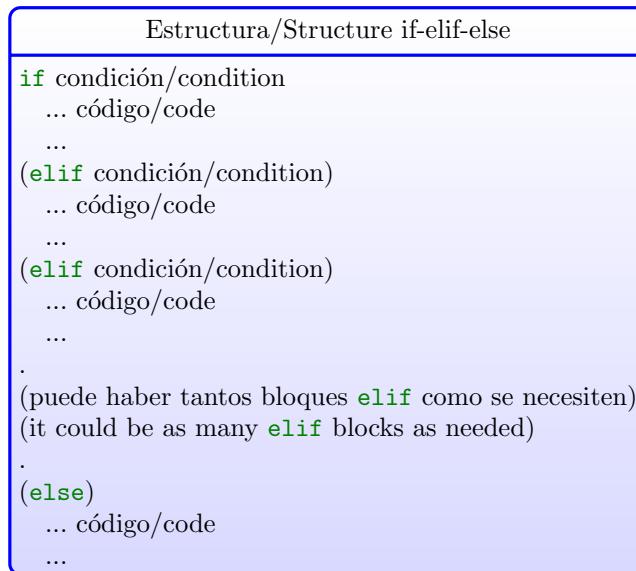


Figura 2.13: Esquema general de la estructura de flujo condicional `if` los términos escritos entre paréntesis son opcionales.

Figure 2.13: General outline of an `if` conditional flow structure. Terms enclosed in parenthesis are optional

Si llamamos ahora a la función dando como valor de entrada un número par, Ejecutará el código situado debajo del `if` y antes del `elif` y se saltará todo lo demás hasta llegar al final del bloque condicional. Si el número introducido no es par, pero es divisible por tres, se saltará el código situado por debajo del `elif`, ejecutará el código contenido debajo del `elif` hasta el `else` y saltará el resto del código hasta llegar al final del bloque condicional. Por último, si el número no es par ni divisible por tres, solo ejecutará el código situado por debajo del `else`.

Un aspecto que debemos resaltar, es que el programa ejecutará el código correspondiente a la primera condición que se cumpla, y se saltará el resto hasta llegar al final del bloque condicional. Así por ejemplo, si en nuestro ejemplo introducimos el número 6, el programa nos mostrará el mensaje “el número es par”, puesto que ésta es la primera condición que se cumple, pero nunca nos mostrará el mensaje “el número es divisible por 3”. Porque una vez comprobada y cumplida la primera condición (ser par) el programa sal-

If we introduce now an even number in the function `parthree`, it executes the code beneath the `if` and before the `elif`. It skips all the remaining code till the end of the conditional block. If the introduced number is odd and divisible by three, the function skips the code beneath the `if` and executes the code between the `elif` and the `else`. It skips the remaining code till the end of the conditional block. Lastly, if the number is odd and is not divisible by three, it only executes the code beneath the `else`.

Note that the program will execute the code belonging to the first condition met and skip the remaining code till the end of the conditional block. For instance, if we introduce the number 6 in our example function, the program will show the message “6 is an even number” because this is the first condition it meets. But it never will show us the message ‘6 is a number divisible by three because, once the program checks that the first condition is fulfilled (be even), the program jumps to the end of the conditional structure without further checking. Figure 2.13 shows a comple-

ta directamente al final de la estructura, sin comprobar nada más. La figura 2.13 muestra el esquema completo de una estructura `if`. Los términos entre paréntesis pueden estar o no presentes en una implementación concreta. Las siguientes líneas de comandos muestran la importación y el uso de la función `partthree` empleando distintos número como entrada,

```
In [18]: from numero_div2_div3 import partthree
In [19]: partthree(4)
4   es un numero par
4   is an even number
Out[19]: ()
In [20]: partthree(6)
6   es un numero par
6   is an even number
Out[20]: ()
In [21]: partthree(9)
9   es un numero divisible por tres
9   is a number divisible by three
Out[21]: ()
In [22]: partthree(7)
7   es un numero impar pero no divisible por tres
7   is an odd number but it not divisible by three
Out[22]: ()
```

**Estructuras if anidadas.** En el ejemplo anterior, hemos visto cómo, si el número introducido en la función era par y además divisible por tres, el programa nunca nos informaría de esta segunda propiedad, debido al carácter excluyente de la estructura `if`. Una manera de resolver este problema, es mediante el uso de estructuras `if` anidadas. La idea es muy sencilla, se construye una estructura `if` para comprobar una determinada condición, si esta se cumple, dentro de su código se construye otra estructura `if` para comprobar una segunda condición, y así sucesivamente, todas las veces que sea necesario. Si modificamos nuestro ejemplo anterior, incluyendo un `if` anidado,

te outline of the `if` structure. Terms enclosed in parentheses are optional and could or may not be included in a specific program implementation. The following commands show the import and use of function `partthree`, using different numbers as inputs.

**Nested if structures** In the previous example, we saw that the program could not detect if the input number was both even and divisible by three due to the exclusive nature of the `if` statement. We can solve this problem using nested `if` structures. It is a straightforward idea. We build an `if` statement to check a specific condition. If the condition is fulfilled, inside the `if` statement, we build another `if` statement to check a second condition, and so on as many times as needed. Notice that in a nested `if` structure, the inner condition is checked only when the outer one is met. We can modify our previous example, using now a nested `if`,

---

numero.parytres\_nested.py

---

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Created on Fri Apr 19 15:31:19 2024
5  Script para crear una función que comprueba si un número es par
```

```

6  Si lo es, tiene un if anidado para probar si además es divisible entre tres
7  si no lo es comprueba si es divisible por 3.
8
9  Script to create a function tha checks if a number is even and if it is so,
10 checks if it is divisible by three to. else, the program checks if the number
11 can be divided by 3
12 ©author: juan
13 """
14
15 def partthreenested(n):
16     """
17     Esta función toma como entrada un número n. Si n es par devuelve un mensaje
18     indicando que el número es par.
19     Si no, comp
20     This funtion takes an integer number n as input. If n is even then
21     it shows a message indicating the number is even.
22     """
23     if n%2 == 0:
24
25         if n%3 == 0:
26             print(str(n),' es un numero par y divisible entre 3')
27             print(str(n),'is an even & divisible by 3 number')
28         else:
29             print(str(n),' es un numero par')
30             print(str(n),'is an even number')
31     elif n%3 == 0:
32         print(str(n),' es un numero divisible entre 3')
33         print(str(n),'is an number divisible by 3')
34     else:
35         print(str(n),' es un numero no es divisible entre 2 ni entre 3')
36         print(str(n),'is not divisible by 2 neither by 3 ')
37
38     return()
39

```

---

la modificación realizada está en las líneas 23-25. Se trata de una nueva estructura `if` definida dentro de la que comienza en la línea 20 comprobando si el número es par. El programa solo llegará a comprobar este nuevo `if`, si efectivamente se a cumplido la condición de que el número sea par. Si nos fijamos en la estructura del código, vemos que tiene tres niveles de indentación: El primero corresponde al código perteneciente a la función `partthreenested`, el segundo corresponde al código la estructura `if` más externa y el tercero al código del `if` anidado. Si añadieramos un nuevo `if` dentro de éste último, deberíamos indentar su código un nivel más.

We can see that we have added code lines 23 – 24. This added code defines a new `if` statement defined inside the `if` structure that starts at line 20. The program will only check this new condition if the number is even and it has met the condition imposed in line 20. If we focus on the code structure, we can see that it has three indentation levels: The first one belongs to the code of function `partthreenested`, the second one to the outer `if` structure and the last one to the nested `if`. If we would add a new `if` inside the last one, we should indent the code one level more.

```
In [41]: from numero_parytres_nested import partreenested as piubello

In [42]: piubello(12)
12 es un numero par y divisible entre 3
12 is an even & divisible by 3 number
Out[42]: ()

In [43]: piubello(5)
5 es un numero no es divisible entre 2 ni entre 3
5 is not divisible by 2 nor by 3
Out[43]: ()

In [44]: piubello(4)
4 es un numero par
4 is an even number
Out[44]: ()

In [45]: piubello(9)
9 es un numero divisible entre 3
9 is an number divisible by 3
Out[45]: ()
```

### 2.7.2. Bucles

En ocasiones, es preciso repetir una operación un número determinado de veces o hasta que se cumple una cierta condición. Los lenguajes de alto nivel poseen estructuras específicas, para repetir la ejecución de un trozo de programa las veces que sea necesario. Cada repetición recibe el nombre de iteración. Estas estructuras reciben el nombre genérico de bucles. Vamos a ver dos tipos: los bucles `for` y los bucles `while`.

**Bucles for.** Un bucle `for` repite las sentencias contenidas en el bucle un determinado número de veces, es decir realiza un número fijo de iteraciones. La estructura general de un bucle `for` se muestra en la figura 2.14.

El bucle empieza con la palabra clave `for`, seguida de una variable a la que hemos dado el nombre genérico de índice. Después viene la palabra clave `in`, una lista de valores y el símbolo `:`. La variable índice irá tomando sucesivamente los valores de los elementos contenidos en la [lista de valores]. El código contenido en el bucle, desde la línea siguiente al `for`, debe estar indentado, igual que vimos

### 2.7.2. Loops

Sometimes, it is necessary to repeat an operation a number of times or till a particular condition is fulfilled. High-level program languages have specific structures to repeat a chunk of code as many times as needed. Each repetition is named an iteration. These structures receive the generic name of loops. We will see two types of loops: `for` loops and `while` loops.

**For loops** A `for` loop repeats the sentences inside the loop a predefined number of times. That is, it performs a fixed number of iterations. A general outline of the `for` loop structure is depicted in figure 2.14.

The loop starts with the keyword `for` followed by a variable we have called index. After, we write the keyword `in`, a list of values and a colon `:`. The index (variable) will take all values contained in the [list of values] one after another. The code contained in the `for` loop, from the line just beneath the `for`, should be written using indented lines, in the same way that we seen for the conditional structures. Inside the loop, this code will run as many ti-

Estructura de un bucle for / For loop structure

```
for índice/index in [Lista de valores/Values list]
    ... código /...code
    (condición: / condition: break)
        ... código / ...code
    (condición: / condition: continue)
        ... código / ... code
```

Figura 2.14: Esquema general de la estructura de un bucle `for` los términos escritos entre paréntesis son opcionales.

Figure 2.14: `for` structure General outline. Terms enclosed in parentheses are optional.

en el caso de las estructuras condicionales `if`. Se ejecutará tantas veces como valores tenga la lista de valores. Antes de hablar de las sentencias `break` y `continue`, veamos alguno ejemplos.

mes as items the list of values has. Let's have a look at some examples before describing the `break` and `continue` sentences.

---

— numero\_par.py —

```
1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3 """
4 Created on Mon Apr 29 15:08:34 2024
5 Este modulo contiene algunos ejemplos del uso de los bucles for
6 This module contains some examples of for loops use
7 @author: juan
8 """
9
10 def sumalist(A):
11     """
12     Input A should be a list of numbers no mater how long
13     the function shows the values on the screen, sum them and save the result
14     in the returned variable y
15
16     La entrada A debe ser una lista de números the cualquier longitud.
17     La función muestra los valore de la lista, los suma y devuelve el resultado
18     en la variable y
19     """
20     y = 0 #definimos la variable en la que guardaremos el resultado de la suma
21         #define the variable to save the result of the sum
22     for i in A:
23         print(i)
24         y = y + i
25     return y
26
27 def sumaelementos(A,B):
28     """
29     Inputs A and B shoudd be two list of numbers with the same length
30     The funtion calculates the sum of the elements of A and B that share the
```

```

31     same location inside their respective list i.e. The result is save
32     in list S in the same location: S[i] = A[i]+B[i]
33 """
34
35     if len(A) != len(B):
36         print('Las listas que me das no son igual de largas')
37         print('Sling your hook! The lists lengths are different. ')
38         return([])
39     else:
40         S = []
41         index = range(len(A)) #index to cover the elements of A&B
42         for j in index:
43             S.append(A[j] + B[j]) #adding the elements in the same location
44
45     return(S)
46
47 def firsttime(A,b):
48 """
49     esta función busca la primera vez que un número b aparece en una lista
50     A y devuelve su posición dentro de la lista.
51     Si no aparece muestra un mensaje por pantalla
52     This program find the first time a number b appears in a list A
53 """
54     p = 0
55     for i in A:
56         if i==b:
57             break
58         p = p + 1 #p += 1
59     if p == len(A):
60         print('el número', b, 'no está en la lista')
61         print('number', b, 'is not included in the list')
62     else:
63         print('el número', b, 'ocupa en la lista la posición', p )
64         print('number', b, 'is located at position', p, 'in the list' )
65     return(p)
66
67 def buscapar(A):
68 """
69     Esta función busca los numeros pares de una lista de numeros y
70     crea una nueva lista solo con los pares contenidos en la lista inicial
71 """
72     B = [] #and empty list to save the even numbers
73     for i in A:
74         if i%2 !=0:
75             continue
76         B.append(i)
77     return B
78
79
80 def listadelistas(A,b):
81 """
82     Esta función busca las veces que aparece un número en una lista

```

```

83     bidimensional (una lista de listas de números)
84     Devuleve las veces que aparece el número y los indices de las posisiciones
85     en que aparece.
86     """
87     ind = [] #an empty list to save the list indexes where the number is.
88     counter = 0 #number of times the number is repeated in the list
89     for i in range(len(A)):
90         for j in range(len(A[i])):
91             if A[i][j] == b:
92                 ind.append((i,j))
93                 counter = counter+1
94
95     return(ind,counter)
96
97 def pairs(A,B):
98     """
99     A and B are two list of numbers of the same lenght. The function build a list
100    of pairs, taken one element from and one element from b with the condition that
101    they should be different. It doesn't the mater the order i.e (1,2) and (2,1)
102    are different a valid'
103
104     """
105    pares = [] #an empty list
106    for i in A:
107        for j in B:
108            if i != j:
109                pares.append((i,j))
110
111    return(pares)

```

---

La función `sumalist`, toma como variable de entrada una lista de números de cualquier longitud y devuelve como salida la suma de todos los números contenidos en la lista. Además, imprime en el terminal de Ipython cada uno de los números. El bucle `for` comienza en la línea 22. La primera vez que el código llega a dicha línea asigna a la variable `i` el primer elemento de la lista `A`. En la línea 23 muestra por pantalla el valor de dicho elemento. En la línea 24 suma su contenido a la variable `y` y vuelve de nuevo a la línea 22 y asigna a la variable `i` el valor del segundo elemento de `A`. El proceso se repite hasta que se han recorrido todos los elementos de la lista. Es entonces cuando el programa termina el bucle y ejecuta la linea 25, devolviendo el valor de la variable `y`, en la que se ha acumulado la suma de los elementos de la lista.

La función `sumaelementos`, toma como variables de entrada dos listas de igual longitud

The function `sumalist` takes a whatever length list of numbers as an input and returns the sum of the number in the list as an output. Besides, it writes each number of the list in the Ipython console. The `for` starts in line 22. The first time the program arrives at this line, it assigns to variable `i` the first element of the list `A`. In line 23, the program shows the value of this element on the screen. In line 24, it adds this value to variable `y`, and the program goes back to line 22 and assigns to variable `i` the second value of the list `A`. This process repeats until the elements of the list `A` have been exhausted. Then, the program finishes the `for` loop and runs line 25 returning the value of variable `y`, which will have accumulated the sum of list `A` elements.

The function `sumaelementos` takes two lists of equal length as input variables and returns a list with the sums of the elements located at the same position on the input lists. A con-

y devuelve como variable de salida una lista cuyos elementos contienen las sumas de los elementos de las listas de entrada que ocupan la misma posición. El código está protegido por una estructura condicional, de modo que si las lista de entrada no tienen la misma longitud el programa muestra un mensaje por pantalla y devuelve una lista vacía. Si las listas tienen la misma longitud, en la línea 41 se crea la variable `index` empleando el comando `range`. Por tanto, `index` sera una lista que contendrá los valores enteros desde 0 hasta la *longitud de las listas de entrada* – 1 . En la línea 42 se inicia el bucle `for` sobre dicha lista `i in index`. En la línea 43 se suman los elementos de las listas de entrada que ocupan la posición indicada por el valor de `j` y se añade el resultado a la lista `S`. Cuando el bucle ha recorrido y sumado todos los elementos de las listas de entrada, el bucle termina y se devuelve el valor de `S`. Esta manera de crear una lista de índices para recorrer los elementos de una estructura de datos se usa con mucha frecuencia en programación.

Volvamos, antes de seguir con los ejemplos, a la estructura de la figura (2.14). En ella aparecen dos palabras clave opcionales que permiten interrumpir o alterar la ejecución de un bucle `for`. Ambas, deben siempre incluirse dentro de una estructura condicional. La palabra clave `break`, interrumpe la ejecución del bucle `for` que la contiene y sigue adelante con la ejecución del resto de las líneas de código que haya por debajo del bucle. La palabra clave `continue` salta todas las líneas de código que tenga por debajo, dentro del bucle y empieza una nueva iteración. Para ver su funcionamiento volvamos al programa de ejemplo anterior.

La función `firsttime` busca la primera vez que un número `b` aparece en una lista `A`, devuelve la posición que ocupa en la lista y termina la ejecución. El bucle `for` comienza en la línea 55. La estructura `if`, contenida en el bucle `for`, comprueba si el elemento correspondiente de la lista coincide con el valor buscado. Si es así, `p`, ejecuta la instrucción `break` y termina la ejecución del bucle. Si no coincide con el número, incrementa el valor de la variable `p` en 1, y vuelve al principio del bu-

ditalional structure protects the code so that if both input lists are not the same length, the program casts a message on the screen and returns an empty list. If both lists have the same size, in line 41, the program creates the variable `index` using the command `range`. Therefore, `index` will be a list of numbers from 0 to *length of the input lists* – 1. In line 42, the `for` loop stars, using this list `i in index`. In line 43, the elements of the input lists, located at the position indicated by the value of `j`, are summed, and the result is appended to the list `S`. When the loop has covered all the elements of the input lists, the loop is over, and the program returns the value of the list `S`. It is interesting to note the method for covering the elements of a data structure using the `range` command because, in programming, it is a very commonly used method.

Before proceeding with the code examples, we will return to figure 2.14. There are two keywords included in the general `for` structure layout that allows to interrupt or alter the execution of a `for` block. Both should always be included in a conditional statement. The keyword `break` interrupts the execution of the `for` loop that encloses it, and the program jumps to the following line after the `for` block. The keyword `continue` skips every code line beneath it, belonging to the `for` block and starts a new iteration. We will see how they work, resuming the study of the example code included above.

The function `firsttime` searches for the first time that a number `b` appears in a list `A`, returns the position the number is located at in the list and finishes the program execution. The `for` loop starts at line 55. The `if` structure, enclosed in the `for` loop, checks if the element `i` of the list meets the wanted number. if so, the program executes the `break` command, and the program returns the position `p` of the number in the list. If the element `i` of the list `A` does not match the wanted number, the program increments the value of variable `p` in 1 and goes back to the beginning of the `for` loop and will try the next element of the list. If the wanted number is in the list, the final value of `p` will be less than the list length, and the Ipython console will display a mes-

cle. Si el número está en la lista, el valor de `p` será menor que la longitud de la lista y por, tanto se mostrará en el terminar de Ipython un mensaje indicando que el número está en la lista y qué posición ocupa. Si el número no está en la lista, el bucle `for` no se interrumpe y el valor de `p` coincidirá con la longitud de la lista, mostrando entonces en el terminal de Ipython un mensaje indicando que el número no está en la lista<sup>4</sup>.

La función `buscapar` busca los números pares de una lista y crea una nueva lista con los números pares contenidos en la lista original. En la línea 71 se comprueba si el número es impar y, caso de serlo, la instrucción `continue` hace que el programa no ejecute la línea 73, evitando que los números impares se incluyan en la lista `B`, que solo contendrá números pares al final de la ejecución del programa.

**bucles for anidados.** Los bucles `for` pueden anidarse unos dentro de otros de modo análogo a como se hace con las estructuras `if`. Las funciones `listadelistas` y `pairs` muestran ejemplos de su uso.

`listadelistas` toma una lista de dos dimensiones —una lista `A` formada por listas de números— y busca cuantas veces aparece un número en ella. Para ello emplea dos bucles `for` anidados. En la línea 86, empieza el primero de los bucles, creando un índice `i` que recorrerá la lista. En la línea 87 comienza un segundo bucle creando un segundo índice `j`. Es segundo índice recorrerá cada una de las lista de números incluidas en la lista `A`. Este bucle `for` interior contiene una estructura condicional que comprueba si el elemento correspondiente, al elemento que ocupa la posición `j` dentro de la lista de números `i`, coincide con el valor deseado. Si coincide, los índices de número se guardan como una dupla en la lista `ind` y se incrementa en una unidad el contador `counter`, que lleva la cuenta de las veces que ha aparecido el número. Veamos un ejemplo de cómo funciona,

---

<sup>4</sup>Hay maneras mejores de hacer esto mismo en Python. El ejemplo solo pretende mostrar el uso del comando `break`

sage indicating that the number is in the list and the position the number has in the list. If the program can not find the number in the list, the `for` loop is not interrupted, and the Ipython console will display a message indicating that the number is not in the list<sup>4</sup>.

The function `buscapar` searches for the even numbers included in a list and builds a new list with the even numbers contained in the original list. The program checks in line 71 if the number is odd. If this is the case, the `continue` command makes the program skip line 73, avoiding including the odd numbers in list `B`, which will only have even numbers when the program finishes.

**Nested for loops.** The `for` loops can be nested one into another, likewise to how we nested `if` structures. Functions `listadelistas` and `pairs` show a couple of examples of how to do it.

The function `listadelistas` takes a bidimensional list – a list `A` in which each element is, in turn, a list of numbers– as input and searches how many times a number also supplied to the function as an input, is enclosed in the list. It performs this task using two nested `for` loops. The first (outer) loop starts at line 86, defining an index `i`, which will cover the list. A second (inner) `for` loop starts at line 87, creating a second index `j`. This second index will cover each list of numbers included in list `A`. The inner `for` loop has a conditional structure inside that checks if the number that takes position `j` inside the number list `i` meets the wanted value. If so, the index `(i,j)` are saved as a tuple in list `ind` and the variable `counter`, which counts the times the wanted number has been found in the list, is increased in one unit. The following code presents an example of the function performance,

---

<sup>4</sup>there are better ways to do this. This example is intended only to show the performance of the `break` statement.

```
In [12]: LoL =[[1],[1,2,5],[2,-1,5,3,5],[4,3,2,0,5],[2,1]]  
In [13]: from ejemplo_for import listadelistas  
In [14]: indice, contador = listadelistas(LoL,5)  
In [15]: indice  
Out[15]: [(1, 2), (2, 2), (2, 4), (3, 4)]  
In [16]: contador  
Out[16]: 4
```

La función nos muestra las veces que aparece el número 4 en la lista LoL. Dejamos al sufrido lector que trate de averiguar como funciona la función `pairs`.

**List comprehension.** Python cuenta con excelentes recursos para realizar operaciones de modo iterativo sobre diversos tipos de estructuras de datos. Entre ellos, vamos a ver una conocida con el nombre de *List comprehension*. La idea es crear una lista, en la que los elementos se obtienen como el resultado de iteraciones en otra u otras listas. Su estructura general toma la forma:

The function shows us the times that number 4 is included in the list LoL. We leave you to try to discover how the function `pairs` works. (if you have followed the reading till now, you should find it out with your hands tied behind your backs).

**List comprehension.** Python has excellent resources for iterating over different data structures. We will not present you many of them, but just one very useful and widely used by Python programmers: *List comprehension*. The idea is to build a list whose elements result from iterations on several other lists. Its general structure is,

---

```
newlist = [expression for item in iterable if condition == True]
```

La nueva lista se forma a partir de una operación (expresión) que se realiza sobre cada elemento (item) de otra estructura de datos (iterable). Además se puede imponer una condición sobre cada elemento para que se lleva a cabo la operación. El siguiente script de Python, contiene un par de ejemplos,

The new list is built from an operation (expression) carried out on any element (item) of another (iterable) data structure. Moreover, it is possible to add a condition for the operation to be carried out. The following Python script contains a couple of examples.

---

#### List\_comprehension\_example.py

---

```
1 #!/usr/bin/env python3  
2 # -*- coding: utf-8 -*-  
3 """  
4 Created on Mon May 13 08:59:21 2024  
5 Un par de funciones usando list comprehension  
6 A pair of function using list comprehension  
7 @author: juan  
8 """  
9  
10  
11 def lc_demo(A):  
12     """  
13     Esta función busca los numeros pares en una lista A y los guarda en
```

```

14     una nueva lista B
15     This function finds the odd number in list A, a created a new list
16     B with them
17     """
18     B = [i for i in A if i%2 != 0]
19     return B
20
21 def pair_cmh(A,B):
22     """
23     A and B are two list of numbers of the same lenght. The function build a list
24     of pairs, taken one element from and one element from b with the condition that
25     they should be different. It doesn't the mater the order i.e (1,2) and (2,1)
26     are different a valid'. But, but we will use a list comprehension
27
28     """
29     pares = [(i,j) for i in A for j in B if i != j]
30     return(pares)

```

---

La función `lc_demo`, Crea una lista con los números impares contendidos en otra lista. Es fácil ver la estructura de la *list comprehension*, creada en la línea 18: la lista B, está formada por los elementos `i` incluidos en la lista A, que cumplan `i%2 != 0`, es decir, que no sean divisibles por dos y, por tanto, impares.

La segunda función, `pair_cmh`, es un poco más compleja, pero deja ver las posibilidades del método. La función crea una lista de tuplas, para ello emplea dos estructuras `for`, tal y como están escritas, es como si fueran dos `for` anidados. Así, la función compara cada elemento `i` de la primera lista con todos los elementos `j` de la segunda y, si son distintos `i != j`, guarda los dos números como una tupla `(i,j)` en la lista `pares`.

**Bucle while.** Este bucle tiene la misma finalidad que un bucle `for`, repetir un trozo de código un determinado número de veces. Lo que cambia, es el mecanismo que determina cuantas iteraciones realizará el bucle. En el caso de un bucle `while` las iteraciones se repiten un número indefinido de veces mientras se cumpla una determinada condición impuesta al principio del bucle. La figura 2.15 muestra la estructura general de un bucle `while`. Como sucede en todas las estructuras de programación que hemos visto en Python, es preciso que el código correspondiente al bucle `while` esté indentado. El siguiente script muestra algunos

The function `lc_demo` makes a list with the odd numbers contained in another list. It is easy to understand the structure of the *comprehension list* created in line 18: List B is built up by the elements `i`, included in list A which fulfil `i%2 != 0`, that is, those elements that are not divisible by two and, thus, odd numbers.

The second function, `pair_cmh`, is slightly more complex, but it allows us to appreciate the possibilities of the method. The function creates a list of tuples. To do so, it uses a pair of `for` structures, written in a way that is equivalent to two nested `for` statements. Hence, the function compares each element, `i`, from the first list with any element, `j`, from the second one. If the elements are different `i != j`, it saves the elements as a tuple `(a,b)` in the list `pares`.

**While loop** This kind of loop has the same purpose as the `for` loop, to repeat a chunk of code several times. The main difference is the mechanism determining the number of iterations the loop will take. For a `while` loop, the iterations are repeated an undefined number of times while a condition imposed at the beginning of the loop is fulfilled. Figure 2.15 shows an outline of the `while` loop structure. As in every Python programming structure we have been so far, the code of the `while` loop should be indented. The following script

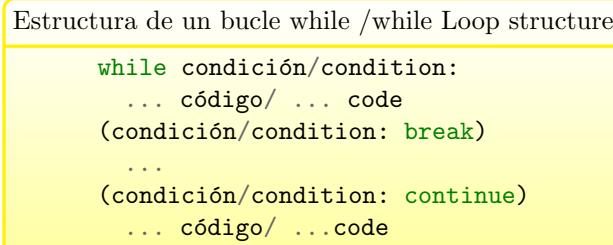


Figura 2.15: Esquema general de la estructura de un bucle while los términos escritos entre paréntesis son opcionales.

Figure 2.15: General structure for a while loop. Terms enclosed in parentheses are optional.

ejemplos de uso,

shows some examples of how to use it.

---

examples\_while.py

---

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Created on Mon May 13 10:46:35 2024
5  Un par de ejemplos de uso de bucles while
6  A pair of loop while examples
7  @author: juan
8  """
9
10 def pmax(a,M):
11     """
12     This funtion optain the minimum exponent n such that a**n >= M
13     esta funci-on optiene el minimo esponte n tal que a**n >= M
14     """
15     n = 0
16     while a**n < M:
17         n = n+1
18     return n
19
20 def L_suma(A,B):
21     """
22     This function take tho list of list of numbers A and B and adds the numbers
23     located at the same position in both list. (both lists should have the
24     same dimensions)
25     Esta funciόn toma dos listas A y B y crea una nueva lista con la suma de los
26     números que ocupan la misma posiciόn (ambas listas deben tener las mismas
27     dimensiones)
28     """
29
30     i = 0 #indice de para recorrer las listas de números
31             #numbers to cover the list of number list
32     C = []
33     n_lista = len(A)
34     while i < n_lista:
35         n_num = len(A[i])

```

```

36     j = 0 #indice para recorrer los numeros de cada lista de numeros
37         #index to cover the number of each number list
38     C.append([])
39     while j < n_num:
40         C[i].append(A[i][j]+B[i][j])
41         j = j+1
42         i = i+1
43     return C

```

---

La función `pmax`, toma como entradas dos números y calcula a qué exponente es necesario elevar el primero de los números para que el resultado sea mayor que el segundo. El código del bucle `while` comienza en la línea 16 comprobando si el número elevado al valor de `n` es menor que el valor de la segunda variable de entrada `M`. Si efectivamente se cumple esta condición, se ejecuta la línea 17 que lo único que hace es incrementar en una unidad el valor de `n`. A continuación el programa vuelve de nuevo a la línea 16 y comprueba si se cumple la condición para el nuevo valor de `n`. Este proceso continúa hasta que la condición deja de cumplirse. En ese momento, el programa sale del bucle `while` y ejecuta la instrucción `return`, devolviendo el último valor calculado de `n`.

Un aspecto muy importante del bucle `while` es que al programarlo hay que asegurarse de que dentro del bucle existe la posibilidad de cambiar la condición de entrada. Si no, el programa no podrá terminar nunca el bucle<sup>5</sup>. Las sentencias `break` y `continue` son idénticas a las descritas en el caso de los bucles `for`, por lo que no insistiremos más sobre el asunto.

**Bucles while anidados.** Del mismo modo que se anidan los bucles `for`, es posible anidar bucles `while`. En el código de ejemplo anterior se ha definido una función, `L_suma`, Que emplea dos bucles `while` anidados para sumar los valores de dos listas de listas de números que ocupan la misma posición. Las listas deben tener las mismas dimensiones, mismo número de listas de números y mismo número de valores en cada lista de número. Lo primero

<sup>5</sup>Cuando se produce esta situación por un error en el diseño del programa, el bucle se puede parar pulsando a la vez las teclas `ctrl+C`

The function `pmax` takes two numbers as input and calculates the exponent you have to raise the first number to obtain a result greater than the second one. The `while` loop code starts at line 16, checking whether the first input variable raises to `n` is less than the second input variable. When this condition is fulfilled, the program executes line 17 that increases in one unit the value of `n`. Then, the program returns to line 16 and checks if the condition is met for a new value of `n`. This process goes on until the condition is no longer met. At this moment, the program exits the `while` loop and executes the `return` instruction, returning the last value of `n` the program calculated.

A critical point to consider when programming a `while` is ensuring that the input to the loop condition will change inside the loop. Otherwise, the program will get into an infinite loop and never stop<sup>5</sup>. The commands `break` and `continue` are identical to those described for the `for` loops. For this reason, we will not insist on them any more.

**Nested while loops.** In a similar way as with the `for` loops, it is possible to nest `while` loops. We have defined the function `L_suma`, in the above code example, which uses two nested `while` loops to add the values of two lists located at the same position in each list. The lists should have the same dimension, the same number of lists of numbers and the same number of numbers in each list of numbers. We first define an index to cover the lists of numbers contained in the input lists `A` and `B`. Then, a new empty list `C` is created. An outer

<sup>5</sup>If you get into this situation for an error in the program design, please don't panic: press the board keys `Control + C` simultaneously and breathe normally.

que se hace en definir un índice, para ir recorriendo cada una de las listas de números contenidas en las listas de entrada A y B. Despues se crea una nueva lista C, vacía. En la línea 34 se empieza el bucle `while` exterior. Este bucle se ejecutará mientras le índice `i` sea menor que el número de listas de números contenidas en las listas de entrada. Una vez que se entra en dicho bucle, lo primero que se hace es comprobar cuantos numeros hay en la lista `i` y se añade una nueva lista (de números) vacía a la lista `C`. El programa entra en el `while` interior, que va recorriendo los valores de las listas de números `i` contenidas en las listas de entrada, los suma y los añade a la lista `C`. Cada vez que termina con una de las listas de números, `j = n_num`, El programa sale del bucle interior, incrementa en una unidad el valor de `i`, cuenta cuantos números hay en la nueva lista y añade una nueva lista vacía a `C` y vuelve a entrar en el bucle interior. El proceso continúa hasta que se agotan las listas de números, `i = n_lista`; cuando se cumple esta condición, el programa sale del bucle `while` exterior y devuelve el valor de `C`.

### 2.7.3. Funciones recursivas.

Una función recursiva es una función que se llama a sí misma. Hemos esperado hasta aquí para hablar de ellas porque, de alguna manera, se comportan como un bucle y necesitan una condición de salida para dejar de llamarse a sí mismas y terminar su ejecución. En general, son delicadas de manejar y tienden a consumir mucha memoria ya que cada vez que la función se llama a sí misma necesita crear un nuevo espacio de memoria independiente. Veamos un ejemplo de función recurrente que permite obtener el término enésimo de la sucesión de Fibonacci.

$$f_0 = 0, f_1 = 1, f_2 = 1, f_3 = 2, f_4 = 3, f_5 = 5, f_6 = 8, \dots f_i = f_{i-1} + f_{i-2} \dots$$

La sucesión empieza con los términos 0 y 1 y a partir de ahí cada término es la suma de los dos anteriores. Podemos convertir directamente esta definición en código,

`while` loop starts at line 34. This loop will run while the index `i` is less than the number of list of numbers contained in the input lists. Once the program gets into this outer loop, the program checks first of all how many numbers the list `i` contains and appends a new empty list (of numbers) to list `C`. The program then gets into the inner `while` loop, which covers the values of the lists of numbers `i` contained in the input lists, adds these values between then and appends the result to the list `C`. Any time a list of numbers is completed, `j = n_num`, the program jumps off the inner loop, increases in one unit the value of `i`, checks how many numbers are in the new list of numbers, adds a new empty list to `C` and get again into the inner loop. The process goes on till the lists of numbers are exhausted, `i = n_lista`; when this condition is met, the program gets out the outer `while` loop and returns the value of `C`.

### 2.7.3. recursive functions.

A recursive function is a function that calls itself. We have waited till now to speak of them because, in some ways, they behave like a loop and need an output condition to stop their execution. Speaking in general, they are difficult to deal with and tend to be very memory-consuming because any time a function calls itself creates a new memory independent memory area. We are going to see an example of a recursive function which allows for obtaining the Fibonacci series  $n$  term.

The series starts with the terms 0 and 1, and later on, each term is the sum of the previous two. We can straightforwardly cast this definition into code,

---

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Created on Mon May 13 12:38:14 2024
5
6  @author: juan
7  """
8
9  def fibonacci(n):
10    """
11    Parameters
12    -----
13    n : TYPE integer
14        DESCRIPTION. n is the fibonacci series term whose value we want to
15        calculate
16        n: es el término de la serie de fibonacci cuyo valor queremos calcular
17
18    Returns
19    -----
20    s : TYPE integer
21        DESCRIPTION the value of fibonacci series term n
22
23    """
24    if n < 2:
25        #the value of the term is n itself
26        s = n
27    else:
28        s = fibonacci(n-1)+fibonacci(n-2)
29    return(s)
30
31
32 def fibo_series(n):
33    """
34    Parameters
35    -----
36    n : TYPE integer
37        DESCRIPTION. Last number of the series whe want to build
38
39    Returns
40    -----
41    Fs: List with the n first fibonacci series terms.
42
43    """
44    Fs = [fibonacci(i) for i in range(n)]
45    return(Fs)

```

---

Si  $n$  es menor que dos, la función `fibonacci` da como valor el término correspondiente (1 o 0). Si  $n$  es mayor que dos, vuelve a llamarse a sí misma con entrada  $n-1$  y  $n-2$ , para calcular

if  $n$  is less than or equal to two, the function `fibonacci` returns the corresponding term (1 or 0). If  $n$  is greater than two, the function calls itself with inputs  $n-1$  and  $n-2$ , to cal-

el valor enésimo de la sucesión a partir de la suma de los dos anteriores, la función se irá llamando a sí misma hasta llegar a  $n < 2$ . A partir de ahí irá devolviendo los valores obtenidos en cada llamada hasta obtener el enésimo término.

La segunda función, `fibo_series`, emplea una *list comprehension* para crear una lista con los  $n$  primeros términos de la serie de Fibonacci<sup>6</sup>.

#### 2.7.4. Algoritmos y diagramas de flujo.

Un algoritmo es, en la definición de la Real Academia:

Un conjunto ordenado y finito de operaciones que permite hallar la solución de un problema.

La palabra algoritmo ha llegado hasta nosotros transcrita del nombre del matemático árabe *Al-Juarismi* (circa 780-850 dc). En programación los algoritmos son importantes, porque suponen un paso previo a la creación de un programa.

Habitualmente, partimos de un problema para el que tenemos un enunciado concreto. Por ejemplo: obtener los  $n$  primeros números primos.

El siguiente paso, sería pensar y definir un algoritmo que permita resolver nuestro problema, es importante caer en la cuenta de que un mismo problema puede resolverse, en muchos casos, por distintos caminos. Por tanto es posible diseñar distintos algoritmos para resolver un mismo problema. Un posible algoritmo para el problema de los números primos sería el siguiente,

- Considerar 2 como el primer números primo. (un número primo es aquel que solo es divisible por si mismo o por uno.)

---

<sup>6</sup>si se emplea un valor de  $n$  muy grande, es posible que se agote el número máximo de llamadas recursivas a una función admitido por Python, en cuyo caso el programa dará un error. Además se puede modificar directamente la función `fibonacci` para obtener los  $n$  primeros términos de la serie con menos esfuerzo

culeate the value of the nth term of the series by adding the values of the previous two. The function will be calling itself till it arrives at  $n \leq 2$ . From then on, it will be returning the values of the terms obtained in each call till it gets the nth one.

the second function `fibo_series`, uses a *list comprehension* to create a list with the  $n$  first terms of the Fibonacci series<sup>6</sup>.

#### 2.7.4. Algorithms and flow charts.

According to the Oxford Dictionary, an algorithm is defined as:

##### 2. Mathematics and Computing.

A procedure or set of rules used in calculation and problem-solving; (in later use spec.) a precisely defined set of mathematical or logical operations for the performance of a particular task.

The word algorithm is a transliteration of the Arab mathematician *Al-Juarismi* name (circa 780-850 ac). Algorithms are fundamental in programming due to they are the previous step to program creation.

Usually, we depart from a problem we have a precise statement for. For instance: Obtain the first  $n$  prime numbers.

The next step would be to define an algorithm that allows us to solve the problem. It is important to realise that the same problem can, in many cases, be solved by different methods. Therefore, it is possible to design different algorithms to solve the same problem. An algorithm for the odd numbers problem may be as follows,

- Take number 2 as the first prime number. (A prime number is such that is only divisible by itself and by one)
- Cover all odd numbers from 3 till the number  $n$  of prime numbers required is completed.

---

<sup>6</sup>If the number  $n$  is too large, it is possible that the maximum number of recursive function calls allowed by Python be reached. In this case, the program casts an error. Besides, we could modify the function `fibonacci` to obtain the list with the first  $n$  terms of the series, with less effort.

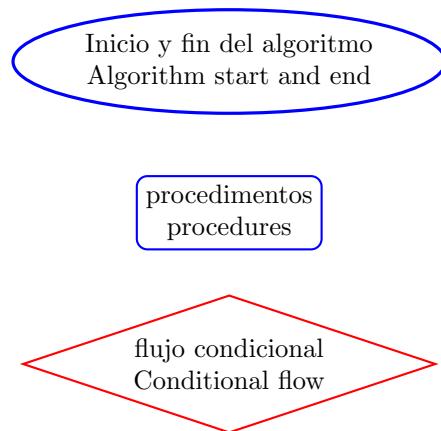


Figura 2.16: Símbolos empleados en diagramas de flujo  
Figure 2.16: Symbols used in flowcharts

- Recorrer todos los números impares desde 3 hasta que se complete el número  $n$  de números primos solicitados.
- Para cada número, probar a dividirlo por todos los primos obtenidos hasta ese momento. Si no es divisible por ninguno, el número es primo y se guarda, si es divisible por alguno de ellos, se interrumpe el proceso y se prueba con el siguiente.

En ocasiones, facilita la comprensión de un algoritmo representarlo gráficamente mediante un diagrama de flujo. Los diagramas de flujo emplean símbolos bien definidos para representar los distintos pasos de un algoritmo y flechas para indicar la relación entre ellos; la relación en la que la información *fluye* de un paso del algoritmo a otro.

No hay una norma rígida para realizar un diagrama de flujo, el grado de detalle con que se describe el algoritmo va en función de las necesidades del programador, o de los destinatarios a quienes va dirigido el diagrama. La idea fundamental es que facilite la comprensión del algoritmo. Se utilizan diversos símbolos para indicar, procedimientos, condiciones, almacenamiento de resultados, etc. La figura 2.16 muestra los tres símbolos más empleados.

Para indicar el inicio y el fin de un algoritmo se emplea como símbolo una elipse. Para indicar un procedimiento concreto, como por

- For each odd number, try to divide it between all prime numbers discovered so far. If the number is not divisible by any of them, then the number is prime. save it and try the next odd number. Otherwise, i.e. if it is divisible for one of them, break the process and try the next odd number.

Sometimes understanding an algorithm is easier if we represent it graphically using a flow chart. Flowcharts use well-defined symbols to represent the steps of an algorithm and arrows to indicate the relationship among the steps; how the information *flows* from one step to another.

There is no rigid form to designing a flow chart; the degree of detail description used in the algorithm depends on the programmer's needs or the needs of the people the flow chart addresses. The basic idea is to use the chart to ease the algorithm's understanding. There are several symbols to indicate procedures, conditions, data storage, etc. Figure 2.16 shows the three most common chart symbols.

We use an ellipse to indicate the beginning and end of an algorithm. To indicate a procedure, for instance, to perform a computation,

ejemplo realizar un cálculo, asignar un valor a una variable, etc, se emplea como símbolo un rectángulo. Por último se emplea un rombo como símbolo, para representar una condición.

Los símbolos se relacionan mediante flechas que indican el sentido en que se ejecuta el algoritmo. los rombos suelen tener dos flechas de salida marcadas con las palabras "si" y "no", para indicar por donde sigue el flujo de información dependiendo de si la condición representada se cumple o no.

Por último un bucle se representa habitualmente mediante una flecha que devuelve el flujo a un símbolo ya recorrido anteriormente.

La figura 2.17 muestra un posible diagrama de flujo para el problema de los números primos. Como puede observarse, contiene más información que la versión que hemos dado del algoritmo descrito con palabras.

Las líneas que marcan los flujos de información nos indican que será necesario implementar un bucle exterior hasta que se complete el número  $n$  de primos solicitados y un bucle interior que deberá comprobar si cada nuevo número impar que probamos, es divisible por los números primos encontrados hasta ese momento.

Hay una tercera condición que debe interrumpir la comprobación para el primer número primo que resulte ser divisor del número que se está comprobando.

Es fácil extraer del diagrama de flujo las estructuras de programación que necesitaremos para elaborar un código que nos permita resolver el problema planteado. Por ejemplo, parece lógico implementar el bucle exterior empleando un bucle **while**, implementar el bucle interior con un **for**, que de tantas iteraciones como primos se han encontrado hasta ese momento, Emplear un **break** para interrumpir la comprobación, etc.

Por supuesto es posible realizar un diagrama de flujo más detallado, en el que incluso se incluya explícitamente parte del código que se va a utilizar. Por ejemplo se podría indicar que se empleará el comando `%` para comprobar si un número es divisible entre otro. Sin embargo, hay que tener cuidado para evitar que un exceso de detalle dificulte entender la lógica del algoritmo contenida en el diagrama.

to assign a value to a variable, etc., we use a rectangle. finally, we use a rhombus as a symbol to indicate a condition.

The symbols are related by arrows to establish the direction of the program flow. Rhombus usually have two exit arrows marked with the words 'yes' and 'not' to indicate where the program flows depending on whether the condition is met or not.

Finally, a loop is usually represented by an arrow which comes back to a previously covered symbol.

Figure 2.17 shows a possible flow chart for the prime numbers problems. As you can see, it has more information than the text version of the algorithm we gave above.

The arrows describing the information flow show that we must implement an outer loop until the required number  $n$  of prime numbers is fulfilled. We will also need an inner loop if any odd number we test is divisible by any of the prime numbers found so far.

There is a third condition which should break the checking when we find the first odd number being a divisor for the odd number the program is checking.

From the chart flow, it is easy to obtain the programming structures needed to build a code that allows us to solve the prime numbers problem. So, it seems reasonable to implement the outer loop using a **while** loop, a **for** loop –that computes as many iterations as prime numbers have been found so far– for the inner loop, a **break** to interrupt the checking process, etc.

Of course, it is possible to build a more detailed flow chart, even including explicitly part of the code to be included in the program. For instance, we may indicate that we will use the command `%` to check if a number is divisible by another. Nevertheless, it is essential to note that you should be careful to avoid an excess of details that make it difficult to understand the logic of the algorithm described by the flow chart.

Eventually, we have to cast the algorithm into code, generating a computer program that allows us to solve the problem. We need to

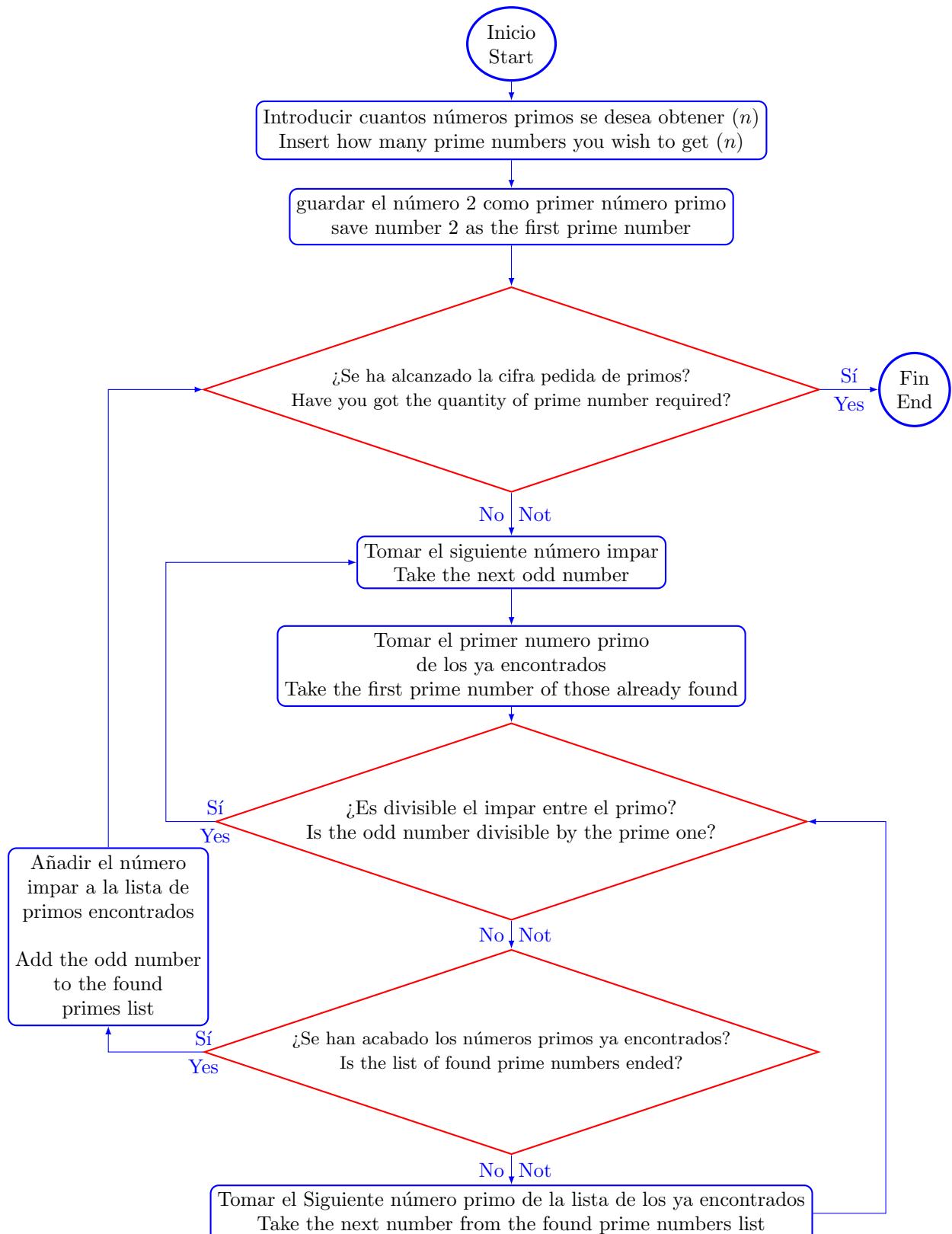


Figura 2.17: Diagrama de flujo para el problema de los números primos

Figure 2.17: Flow chart for the prime numbers problem

Por último el algoritmo se codifica dando lugar a un programa de ordenador que permite resolver el problema. Para ello, hay que identificar las instrucciones del algoritmo con estructuras de programación validas: bucles, condicionales, etc.

El siguiente código muestra dos funciones para calcular los  $n$  primeros números primos. La primera emplea una variable de aviso para detectar cuando un número impar no es divisible por todos los primos anteriores a él. Trata de comprender como funciona

La segunda emplea una estructura `for` especial definida en Python, de la que no hemos hablado anteriormente. Se trata de la estructura `for - else`. Si nos fijamos en las líneas de código 37 a 41, vemos una estructura `for` en la que se comprueba si el numero impar correspondiente es divisible por los primos ya obtenidos. En la linea 41 aparece una instrucción `else`; dicha instrucción esta asociada al bucle `for` y solo se ejecuta cuando el bucle `for` llega al final. En nuestro caso, solo se ejecutará si el bucle `for` no es interrumpido por la instrucción `break` contenida en él. Es decir solo se ejecuta si el numero impar que estamos probando no es divisible por ninguno de los primos anteriores. Si se da esta situación, el bucle `for` llega has el final y además quiere decir que el número impar en cuestión es el además el siguiente número primo. El programa entra entoces en el bloque `else` y ejecuta la única sentencia que contiene, guardando el número impar en la lista de primos. Ni que decir tiene que, como siempre, el código correspondiente al bloque `else`, debe estar indentado.

identify the algorithm instruction with valid programming structures: loops, conditionals, etc.

The following code shows two functions for calculating the  $n$  first prime numbers.

The first function uses a flag variable to detect when an even number is not divisible for any prime number less than it. Try to understand how it works.

The second one uses a Python special `for` structure, which we have not spoken of so far. it is the `for - else` structure. If we focus on code lines 37 to 41, we shall see a `for` structure in which we check if the odd number we are checking is divisible by any of the previously found prime numbers. In line 41, there is an `else` instruction. It is linked to the `for` loop and the program will execute this instruction only if the for loop reaches its end. In our case, the program reaches this `else` command only if the `break` instruction inside does not interrupt the `for` loop. In other words, this happens only if the odd number we are cheking is not divisible by any of the previously found primes and, therefore, it is the next prime number. Then the program gets into the `else` block and executes the single line it contains, saving the odd number in the list of prime numbers. It is needless to say that the code within the "else" statement should be properly indented.

---

primos.py

---

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Created on Tue May 14 11:22:29 2024
5  función para obtener los primeros n numeros primos
6  En la primera empleamos un flag.
7  En la segunda empleamos la estructura for else
8  function to get the first n prime numbers
9  @author: juan
10 """
11
12 def numpr(n):

```

```

13     primos = [2] #introducimos 2 como el primer primo
14             #we take 2 as the first prime number
15     i = 0
16     impar = 3 #tomamos 3 como el rpier impar para probar
17             #we take 3 as the first odd number to test
18     while i<n:
19         pr = 1 #flag activation
20         for i in primos:
21             if impar%i == 0:
22                 pr = 0 #flag deactivation
23                 break
24         if pr ==1:
25             primos.append(impar)
26
27         impar = impar + 2
28     return(primos)
29
30 def numprels(n):
31     primos = [2] #introducimos 2 como el primer primo
32             #we take 2 as the first prime number
33     i = 0
34     impar = 3 #tomamos 3 como el primer impar para probar
35             #we take 3 as the first odd number to test
36     while i<n:
37         for i in primos:
38             if impar%i == 0:
39                 break
40         else:
41             primos.append(impar)
42
43         impar = impar + 2
44     return(primos)
45
46
47
48

```

---

## 2.8. Exportar e importar datos en Python

Almacenar datos es importante para trabajar en varias sesiones y compartir resultados. Dado que cuando Python se cierra, todas las variables de la memoria se pierden, los datos deben almacenarse de alguna otra forma. A veces también es necesario leer datos que están almacenados en un archivo.

### 2.7.5. Reading and writing data in Python

Storing data is important for working in multiple sessions and sharing results. Since when Python shuts down, all variables in memory are lost, data must be stored in some other way. Sometimes it is also necessary to read data that is stored in a file.

**Text files** A text file, many times saved with the *.txt* extension, is a file that only contains plain text. To work with text files we need to

**Archivos de texto** Un archivo de texto, que muchas veces se guarda con extensión `.txt`, es un archivo que sólo contiene texto sin formato. Para trabajar con ficheros de texto, necesitamos utilizar la función `open` que devuelve un objeto de tipo fichero. Se suele utilizar con dos argumentos:

```
f=open(filename, mode)
```

`f` es el objeto tipo archivo devuelto por la función, `filename` es un string con el nombre y ruta del archivo de texto que se quiere abrir y `mode` es otro string que indica de qué manera va a usarse el archivo. Algunos de los modos más comunes son:

- `'r'` es el modo por defecto. Indica que el archivo se abre sólo para leerse.
- `'w'` abre el archivo para escribir en él. Si el archivo no existe lo crea.
- `'a'` es el modo añadir. Añade texto al final del archivo. Si el archivo no existe lo crea al igual que el modo `'w'`.
- `'r+'` abre un archivo (pero no lo crea) para leer y escribir en él.
- `'w+'` abre un archivo para leer y escribir, lo crea si es que no existe. Borra lo que tuviera escrito anteriormente en él.
- `'a+'` abre un archivo para leer y escribir, lo crea si es que no existe. Añade los datos al final del archivo.

Los objetos de tipo archivo tienen métodos que nos permiten acceder y modificar su contenido. Algunos de esos métodos son:

- `write` que permite escribir datos en el archivo
- `read` lee todos los datos que contiene el archivo y los devuelve
- `readline` lee una línea (hasta el carácter fin de línea) del archivo.
- `readlines` lee el archivo completo pero devuelve una lista donde cada línea es uno de los elementos

use the `open` function, that returns a file object. It is usually used with two arguments:

`f` is the file object returned by the function, `filename` is a string with the name and path of the text file to be opened and `mode` is another string indicating how the file is to be used. Some of the most common modes are:

- `'r'` is the default mode. It indicates that the file is opened for reading only.
- `'w'` opens the file for writing to. If the file does not exist, it creates it.
- `'a'` is the append mode. Adds text to the end of the file. If the file does not exist, it creates it as in `'w'` mode.
- `'r+'` opens a file (but does not create it) for reading and writing to it.
- `'w+'` opens a file for reading and writing, creates it if it does not exist. Discards what was previously written to it.
- `'a+'` opens a file for reading and writing, creates it if it does not exist. Adds data to the end of the file.

File type objects have methods that allow us to access and modify their content. Some of these methods are:

- `write` which allows writing data to the file
- `read` reads all the data contained in the file and returns it
- `readline` reads a line (up to the end-of-line character) from the file
- `readlines` reads the whole file but returns a list where each line is one of the elements

En el siguiente ejemplo se puede ver el código para usar algunos de los métodos anteriores. Se abre un archivo y se escriben 5 líneas en él. A continuación se lee dicho archivo. Se vuelve a abrir el archivo pero en modo añadir `a`, escribiendo una línea al final del archivo. Se lee pero usando el método `readlines`, de manera que lo que se obtiene es una lista donde cada elemento es una de las líneas. Finalmente se vuelve a abrir el archivo y se lee, en la variable `line` cada línea, se para el bucle cuando el método `readline` no devuelve nada.

```
f=open('textfile.txt','w')
for i in range(5):
    f.write(f"Line {i}\n")
f.close()

f=open('textfile.txt','r')
print(f.read())
f.close()

f=open('textfile.txt','a')
f.write("Another line")
f.close()

f=open('textfile.txt','r')
print(f.readlines())
f.close()

f=open('textfile.txt','r')
while True:
    line=f.readline()
    if not line:
        break
    print(line)

f.close()

Line 0
Line 1
Line 2
Line 3
Line 4

['Line 0\n', 'Line 1\n', 'Line 2\n', 'Line 3\n', 'Line 4\n', 'Another line']

Line 0

Line 1
```

In the following example you can see the code to use some of the above methods. You open a file and write 5 lines to it. Then read the file. The file is reopened but in append mode, writing a line to the end of the file. It is read but using the `readlines` method, so that what you get is a list where each element is one of the lines. Finally the file is reopened and read, in the variable `line` each line, and the loop is stopped when the `readline` method does not return anything.

Line 2

Line 3

Line 4

Another line

**Trabajar con números y arrays** En muchos de los casos vamos a necesitar guardar y leer números o matrices. Podríamos utilizar los métodos anteriores para guardar los números o matrices en un archivo y leerlos de nuevo. Sin embargo, una manera mejor de hacerlo es usar el paquete numpy para guardar y leer directamente un array.

Dentro de numpy tenemos los métodos **savetxt** y **loadtxt** para guardar arrays en archivos de texto o cargarlos desde archivos de texto.

Para usar **savetxt** tenemos que proporcionar los siguientes argumentos: un string con el nombre del archivo y la variable que contiene el array que vamos a guardar. Además, de manera opcional se pueden proporcionar más. Por ejemplo el formato, bajo la clave de **fmt** (en el ejemplo el formato es un número (%), con tres cifras decimales (.3) de tipo float (f). El tipo de delimitador entre elementos, una cabecera etc. Se puede consultar la documentación para verlos todos.

El método **loadtxt** nos permite leer un array de un archivo y guardarlo directamente en una variable de tipo array de numpy. Para ello simplemente proporcionaremos el archivo donde está almacenado el array y el método nos devolverá el array. En el ejemplo puede verse cómo se guarda una matriz  $2 \times 3$  en un archivo de texto, con una cabecera, cómo se lee usando las funciones **open** y **read** y cómo se lee la matriz y se almacena en un array de numpy usando el método **savetxt**.

```
import numpy as np

arr=np.array([[0.3,4.3,3.5],[-1.23,0.032,-3.23]])
np.savetxt('matrix01.txt', arr,fmt='%.3f', delimiter=' ', header='Col1 Col2 Col3')
f=open('matrix01.txt','r')
```

**Working with numbers and arrays** In many cases we will need to save and read numbers or arrays. We could use the above methods to save the numbers or arrays to a file and read them back. However, a better way to do this is to use the numpy package to directly save and read an array.

Within numpy we have the methods **savetxt** and **loadtxt** to save arrays to text files or load them from text files.

To use **savetxt** we have to provide the following arguments: a string with the name of the file and the variable containing the array to be saved. In addition, you can optionally provide more. For example the format, under the key of **fmt** (in the example the format is a number (%), with three decimal digits (.3) of type float (f)). The type of delimiter between elements, a header etc. You can consult the documentation to see them all.

The **loadtxt** method allows us to read an array from a file and store it directly into a numpy array variable. To do this we simply provide the file where the array is stored and the method will return the array. In the example you can see how a  $2 \times 3$  array is stored in a text file, with a header, how it is read using the **open** and **read** functions and how the array is read and stored in a numpy array using the **savetxt** method.

```

print(f.read())
f.close()
#Now we read it with loadtxt

m1= np.loadtxt('matrix01.txt')
print(m1)

# Col1 Col2 Col3
0.300 4.300 3.500
-1.230 0.032 -3.230

[[ 0.3    4.3    3.5   ]
 [-1.23   0.032 -3.23 ]]

```

**Datos en formato CSV** Hay muchos datos científicos que se almacenan en el formato de archivo de valores separados por comas (CSV). Se trata de un archivo de texto que utiliza una coma para separar los valores. Es un formato muy útil que permite almacenar grandes tablas de datos (números y texto) en texto plano. Cada línea (fila) de los datos es un registro de datos, y cada registro consta de uno o varios campos, separados por comas. También se puede abrir con Microsoft Excel, y visualizar las filas y columnas.

Python tiene su propio módulo csv que puede manejar la lectura y escritura del archivo csv. No vamos a introducir este módulo csv aquí. En su lugar, usaremos el paquete numpy para manejar el archivo csv ya que muchas veces leeremos el archivo csv directamente a un array numpy.

Usando el mismo método `savetxt` de numpy del epígrafe anterior podemos indicar que el carácter delimitador que queremos poner para separar los datos es una coma. A la hora de leer los datos con `loadtxt` tendremos también que indicar que se ha usado la coma para separar unas columnas de otras, como se ve en el siguiente ejemplo.

```

import numpy as np

# Create a random 20x4 matrix
m2=np.random.random([20,4])

```

**Data in CSV format** There is a lot of scientific data that is stored in the Comma Separated Values (CSV) file format. This is a text file that uses a comma to separate values. It is a very useful format that allows large tables of data (numbers and text) to be stored in plain text. Each line (row) of data is a data record, and each record consists of one or more fields, separated by commas. It can also be opened with Microsoft Excel, and rows and columns can be displayed.

Python has its own csv module that can handle reading and writing the csv file. We will not introduce this csv module here. Instead, we will use the numpy package to handle the csv file since we will often read the csv file directly to a numpy array.

Using the same `savetxt` method of numpy from the previous section, we can indicate that the delimiter character we want to use to separate the data is a comma. When reading the data with `loadtxt` we will also have to indicate that the comma has been used to separate some columns from others, as shown in the following example.

```

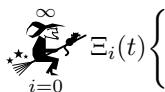
#Save it as a csv file
np.savetxt('randomMat.csv', m2, fmt='%.2f', delimiter=',')

#Read it back with loadtxt
m3=np.loadtxt('randomMat.csv',delimiter=',')

print(m3)

[[0.08 0.91 0.36 0.19]
 [0.61 0.22 0.7 0.66]
 [0.77 0.57 0.77 0.4 ]
 [0.58 0.32 0.89 0.78]
 [0.63 0.09 0.97 0.78]
 [0.35 0.7 0.5 0.96]
 [0.65 0.78 0.04 0.04]
 [0.96 0.31 0.45 0.26]
 [0.75 0.51 0.09 0.87]
 [0.93 0.13 0.8 0.61]
 [0.67 0.99 0.83 0.76]
 [0.21 0.72 0.56 0.46]
 [0.81 0.82 0.67 0.88]
 [0.12 0.01 0.02 0.55]
 [0.88 0.08 0.95 0.7 ]
 [0.74 0.84 0.56 0.3 ]
 [0.77 0.73 0.67 0.89]
 [0.44 0.23 0.67 0.18]
 [0.61 0.4 0.06 0.1 ]
 [0.55 0.35 0.89 0.11]]

```



**Archivos JSON** JSON son las siglas de JavaScript Object Notation y es un formato muy utilizado de archivo de datos, independiente del lenguaje de programación. Un archivo JSON normalmente termina con la extensión ".json". Exploraremos brevemente cómo manejar archivos JSON en Python.

El texto en JSON se guarda mediante cadenas entrecomilladas que contienen valores en pares clave-valor dentro de llaves {}. En realidad es muy similar al diccionario que vimos en Python. Por ejemplo:

{

**JSON files** JSON stands for JavaScript Object Notation and is a widely used data file format, independent of programming language. A JSON file usually ends with the extension ".json". We will briefly explore how to handle JSON files in Python.

Text in JSON is stored as quoted strings containing values in key-value pairs inside {} braces. It's actually very similar to the dictionary we saw in Python. For example:

```

"id": "Frodo",
"tipo" : "Hobbit",
"estado":{
    "posicion": [2.345,-5.985],
    "velocidad lineal": 0.15,
    "velocidad angular": 0,
    "baterias": 7.32
}
}

```

Aunque podríamos trabajar directamente con el archivo de texto y todas las funciones anteriores, hay librerías de python que nos ofrecen métodos para guardar, leer y manipular datos en formato JSON de manera sencilla. Una de ellas es la librería `json`. Importando esta librería tenemos el método `dump` para guardar un diccionario de python (uni en el ejemplo) en un archivo json. A este proceso se le llama serializar. A `dump` hay que proporcionarle el objeto python de tipo archivo (por eso abrimos un archivo con `open` en modo escritura 'w').

Para leer un archivo de datos de tipo JSON y guardar el contenido en un diccionario de python podemos usar el método `load` de la librería `json`. Al igual que a `dump` tendremos que proporcionarle el objeto archivo (ahora lo abriremos en modo lectura 'r') y nos devolverá un objeto de tipo diccionario.

Although we could work directly with the text file and all the above functions, there are Python libraries that provide methods to store, read and manipulate data in JSON format in a simple way. One of them is the `json` library. Importing this library we have the `dump` method to save a Python dictionary (uni in the example) in a json file. This process is called serialize. The `dump` must be provided with the python object of type file (that's why we open a file with `open` in write mode 'w').

To read a JSON data file and save the content in a Python dictionary, we can use the `load` method of the `json` library. Like `dump` we will have to provide it with the file object (now we will open it in read mode 'r') and it will return a dictionary object.

```

import json as js

robot= {
    "id": "Frodo",
    "tipo" : "Hobbit",
    "estado":{
        "posicion": [2.345,-5.985],
        "velocidad lineal": 0.15,
        "velocidad angular": 0,
        "baterias": 7.32
    }
}

js.dump(robot,open('jsondat.json','w'))

mi_robot=js.load(open('jsondat.json','r'))

```

```
print(mi_robot)  
{'id': 'Frodo', 'tipo': 'Hobbit', 'estado': {'posicion': [2.345, -5.985], 'velocidad lineal': 0,
```

Python es un lenguaje de programación muy usado para trabajar con datos. Cuenta con librerías muy potentes y apropiadas para el manejo de grandes cantidades de datos como por ejemplo pandas.

Python is a widely used programming language for working with data. It has very powerful libraries suitable for handling large amounts of data such as pandas.



## 2.9. Ejercicios

1. Tipos de variables en Matlab; matrices y vectores. Escribe, por orden, las siguientes expresiones en la *línea de comandos* de Matlab e interpreta los resultados que obtienes. En los casos en que Matlab devuelva un mensaje de error, trata de averiguar la razón. **Nota:** Es importante hacerlos por orden ya que algunos operaciones se apoyan en los resultados de operaciones anteriores.

```

1 a=[[1,2,3],[4,5,6]]
2 a=[1,2,3],4,5
3 a=[[1 2 3] 1 2 3]
4 a=[[1,2,3],[5,6,8],[3,2,6]]
5 b=[1,2,3]
6 c=[a b]
7 c=[a,b]
8 d=[b,a]
* function/function type()
9 type(a)
10 type(a[0])
11 type(a[0][1])
12 a[2][2] = 'b'
13 type(a[2][2])
* cambiando listas copiadas/changing
  copied lists
14 c[1] = 'ojo'
15 print('c=',c)
16 print('a=',a)
17 print('b=',b)
18 print('d=',d)
19 e = a.copy()
20 a[2][2] = 'cambia tb e'
21 print('c=',c)
22 print('a=',a)
23 print('b=',b)
24 print('d=',d)
25 print('e=',e)
26 e[2][2]= 'pero no al revés'
27 print('c=',c)
28 print('a=',a)

29 print('b=',b)
30 print('d=',d)
31 print('e=',e)
32 %who
33 %whos
34 del(d)
35 %reset
* Operadores/operators
36 a = -6
37 b = 5
38 c = 3.5
39 e = 0.9
40 a*e
41 g=a*e
42 g=e*a
43 a*b
44 r=a-b
45 f=(a+b)*c
46 f= a+b*c
47 p=a**2
48 r=a**2-a*a
* Encadenando operaciones/linking
  operations
49 w=a*b**-1
50 w=a/b
51 w=a//b
52 w=a*b**-1-a/b
53 w=a**-1*b
54 w=(a*b)**-1
55 w=a**-1*b-1/a*b
56 w=a**-1*b-a
* Relacionales y lógicos / relational
  and logic
57 y0 = p<2
58 y1 = p<=2

```

<pre> 59 y2 = p&gt;2 60 y3 = p&gt;=2 61 y4 = p==2 62 y5 = p!=36 63 r1 = a&lt;5 and b==5 </pre>	<pre> 64 r2 = a&lt;b or b&gt;=5 65 r3 = a&lt;b or b&gt;5 66 r4 = (a&lt;b or b&gt;=5) and not(a&lt;b and b&gt;=5) </pre>
--	---

2. Obtén empleando la *línea de comandos* de Ipython el resultado de las siguientes expresiones:

$$\frac{\frac{5 + 3x^2}{6y} - \frac{1 + 4\sqrt{2x}}{6}y}{\frac{-x + \sqrt{x^2 - 4xy}}{2x}}$$

Emplea para  $x$  e  $y$  los valores que prefieras.

3. Abre un fichero nuevo en el editor de textos de Spyder. Escribe las siguiente líneas:

```

A = [35,23]
B = [12,-12.34]
x = [A[0]*B[0],A[1]*B[1]]
y = [A[0]+B[0],A[1]+B[1]]

```

Guarda el fichero con el *primero.py*. Emplea el comando `%reset` en el terminal de IPython para borrar cualquier variable anterior. Ejecuta el fichero y comprueba que existen las variables `A`, `B`, `x`, `y`.

Vuelve ejecutar el comando `%reset`. Emplea el comando `import primero`. ¿Se han creado las variables? ¿qué ha pasado ahora?

Vuelve a borrar las variables del namespace de Python e importa de nuevo tu programa con `import primero as kiki`. ¿Dónde estan ahora las variables `A`, `B`, `x`, `y`?

Crea una variable `x = 'nueva'` Comprueba que las variables `x` y `kiki.x` son distintas.

Importa la variable `x` del modulo `primero` directamente usando la expresion:

`from primero import x`

¿Cuánto vale ahora la variable `x`?

4. Crea en el terminal de IPython la siguiente función, ponle el nombre que quieras

---

```

1 def mifun(A):
2     E = A[0] + A[1]**2
3     D = A[0] - A[1]**-1
4     return(E,D)

```

---

Crea variables adecuadas, ejecuta la función que has creado, guardando el resultado en una sola variable `resultado`.

Ejecuta las siguientes líneas de comando y trata de explicar lo que hacen

```

uno = 2
dos = 4
resultado = mifun([uno,dos])
print(resultado)
me, da = mifun(resultado)
la,risa = mifun(mifun([uno,dos]))
print(me,da,la,risa)

```

5. Escribe el siguiente código en un fichero. Ejecútalo y analiza los resultados,

---

```

1      from random import random
2      dato = random()
3      if dato == 0.5:
4          dato = 0
5      elif dato < 0.5:
6          dato = -dato*10
7      else:
8          dato = dato*10
9      print(dato)

```

---

Modifica el programa,

- a) Para que muestre los resultados de los cálculos por pantalla
  - b) para que, genere una variable (distinta de 'dato') con valor  $-1$  si se cumple que 'dato' es menor o igual que  $0.3$ , con valor  $-0.5$  si 'dato' es mayor que  $0.3$  y menor que  $0.5$ , con valor  $0$  si 'dato' es mayor o igual que  $0.5$  y menor o igual que  $0.7$  y con valor  $1$  en cualquier otro caso.
  - c) Convierte el *script*, en una función, que tome 'dato' como variable de entrada, y devuelva el valor de la variable obtenida, según la condición que se cumpla. (Lógicamente se debe eliminar o comentar la línea `dato=random()`).
6. Crea un scrip de python con el siguiente código y trata de adivinar qué hace antes de ejecutarlo. Utiliza el *help* de Spyder o el interrogante `abs?` en el terminal, para averiguar lo que hace la función `abs`.

---

```

1      n = 5
2      x = []
3      for i in range(n):
4          x.append([])
5          for j in range(n):
6              if i == j:
7                  x[i].append(1)
8              elif abs(i-j) == 1:
9                  x[i].append(-1)
10             else:
11                 x[i].append(0)

```

---

Introduce un punto de ruptura (*breakpoint*) en el código, por ejemplo en la línea 12. Ejecuta paso a paso el programa para ver como va generando el resultado.

7. Crea un script con el siguiente código,

---

```

1  n = 5
2  x = []
3  i = 0
4
5      while i<n:
6          j=0
7          x.append([])
8          while j<n:
9              if i==j:
10                  x[i].append(1)
11              elif abs(i-j)==1:
12                  x[i].append(-1)
13              else:
14                  x[i].append(0)
15              j = j + 1
16          i = i+1

```

---

Repite el estudio del programa anterior

8. Modifica el *script* anterior de modo que quede el siguiente código,

---

```

1  n = 5
2  x = []
3  i = 0
4      while 1:
5          j=0
6          x.append([])
7          while 1:
8              if i==j:
9                  x[i].append(1)
10             elif abs(i-j)==1:
11                 x[i].append(-1)
12             else:
13                 x[i].append(0)
14             j = j + 1
15             if j==n:
16                 break
17         i = i+1
18         if i==n:
19             break

```

---

Repite el estudio del programa anterior. ¿Qué sucede si comentamos la(s) línea(s) que contiene la sentencia `break`? Reconvierte los *scripts* anteriores en funciones que admitan como variable de entrada el tamaño de la lista que se desea generar y devuelvan como variable de salida la lista creada.

9. Escribe —empleando bucles y sin hacer uso del operador (`**`) — una función `potencia` que reciba como argumentos de entrada un número real  $x \in \mathbb{R}$  y un entero  $n \in \mathbb{Z}$  y devuelva el

valor de la  $n$ -ésima potencia de  $x$  ( $x^n$ ). El segundo argumento de entrada puede ser opcional. Si se omite debe tomarse como  $n = 2$  (devolviendo el cuadrado de  $x$ ).

- Crea una función que determine la suma de los inversos de los números naturales impares menores de 1000. Emplea bucles para el cálculo.

$$S_{imp} = \sum_{n=1}^{500} \frac{1}{2n-1} = \frac{1}{1} + \frac{1}{3} + \frac{1}{5} + \cdots + \frac{1}{999}$$

Modifica tu programa, de modo que admita como variable de entrada un número  $m \in \mathbb{N}$  y calcule el valor de la suma de los inversos de los impares menores o iguales que  $m$ .

- Los números de Fibonacci,  $F(n)$  son una serie de números naturales. Los dos primeros valores de la serie son:  $F(1) = 1$  y  $F(2) = 2$ . Los demás términos se calculan como la suma de los dos anteriores:  $F(n) = F(n - 1) + F(n - 2)$ . Escribe un script que genere los primeros  $m$  números de la serie de Fibonacci, almacenándolos en una lista  $F$ . Emplea para el cálculo la estructura `while`.
- Escribe una función que admita como variable de entrada una lista y devuelva como variable de salida otra lista de igual tamaño con los elementos colocados en orden inverso a los de la lista original.  
¿Sabrías hacerlo con una única línea de código? Pista: Usa *List comprehension*
- Modifica el programa del ejercicio 11 para que guarde la lista con los números de Fibonacci creada en un archivo de texto.
  - Modifícalo ahora para que los guarde en un archivo *CSV*.

# Capítulo/Chapter 3

## Introducción a Numpy Introduction to Numpy

When the going gets tough, the  
tough get going

---

Popular Witticism (US)

### 3.1. Numpy: un paquete de Python para cálculo numérico

En el capítulo 2, vimos cómo importar módulos de python en un script o directamente en el terminal de Ipython, de modo que podamos reutilizar el código contenido en ellos. Numpy es un librería de python que contiene funciones y variables específicamente diseñadas para el cálculo numérico. Está estructurada en forma de módulos y submódulos de modo que para utilizarla en nuestros programas basta importarla en nuestros script, importar sus submodulos o importar sus funciones. La base de Numpy la constituyen objetos y operaciones algebráicas. En esta sección vamos a repasar algunos conceptos fundamentales de álgebra lineal y cómo pueden manejarse empleando Python. No daremos definiciones precisas ni tampoco demostraciones, ya que tanto unas como otras se verán en detalle en la asignatura de álgebra.

### 3.1. Numpy: a Python library for scientific computing

In chapter 2, we have seen how to import Python modules to a script or the Ipython console. In this way, we can reuse the code available in the modules. Numpy is a Python library with functions and variables specifically intended to perform scientific computing. Numpy is structured in modules and submodules so that we can import the whole library, its submodules, or just any of its functions into our scripts. The background of Numpy is built of algebraic objects and operations. In this section, we will review some fundamental notions of linear algebra and how to deal with them using Numpy. We will not provide formal definitions or demonstrations because both will be studied in detail in algebra during the next term.

**Matrices** From a functional perspective, we will define a matrix as a table of numbers ordered by rows and columns,

**Matrices.** Desde un punto de vista funcional definiremos una matriz como una tabla bidimensional de números ordenados en filas y columnas,

$$A = \begin{pmatrix} 1 & \sqrt{2} & 3.5 & 0 \\ -2 & \pi & -4.6 & 4 \\ 7 & -19 & 2.8 & 0.6 \end{pmatrix}$$

Cada línea horizontal de números constituye una *fila* de la matriz y cada línea vertical una *columna* de la misma.

A una matriz con  $m$  filas y  $n$  columnas se la denomina matriz de orden  $m \times n$ .  $m$  y  $n$  son la dimensiones de la matriz y se dan siempre en el mismo orden: primero el número de filas y después el de columnas. Así, la matriz  $A$  del ejemplo anterior es una matriz  $3 \times 4$ , y como esta formada por números reales se dice que  $A \in \mathbb{R}^{3 \times 4}$ . El orden de una matriz expresa el tamaño de la matriz.

Dos matrices son iguales si tienen el mismo orden, y los elementos que ocupan en ambas matrices los mismo lugares son iguales.

Una matriz es cuadrada, si tiene el mismo número de filas que de columnas. Es decir es de orden  $n \times n$ .

Mientras no se diga expresamente lo contrario, emplearemos letras mayúsculas  $A, B, \dots$  para representar matrices. La expresión  $A_{m \times n}$  indica que la matriz  $A$  tiene dimensiones  $m \times n$ . Para denotar los elementos de una matriz, emplearemos la misma letra en minúsculas empleada para nombrar la matriz, indicando mediante subíndices, y siempre por este orden, la fila y la columna a la que pertenece el elemento. Así por ejemplo  $a_{ij}$  representa al elemento de la matriz  $A$ , que ocupa la fila  $i$  y la columna  $j$ .

$$A = \begin{pmatrix} 1 & \sqrt{2} & 3.5 & 0 \\ -2 & \pi & -4.6 & 4 \\ 7 & -19 & 2.8 & 0.6 \end{pmatrix} \rightarrow a_{23} = -4.6$$

**vectores** A una matriz compuesta por una sola fila, la denominaremos vector fila. A una matriz compuesta por una sola columna la denominaremos vector columna. Siempre que hablemos de un vector, sin especificar más, entenderemos que se trata de un vector colum-

Each horizontal line of numbers forms a matrix row, and each vertical line a matrix column. A matrix with  $m$  rows and  $n$  columns is denoted as a matrix of order  $m \times n$ .  $m$  and  $n$  are the matrix dimensions, and they are always defined in the same order: first, the number of rows and then the number of columns. So matrix  $A$  in the example above is a  $3 \times 4$  matrix, and as it is built of real numbers, we say that  $A \in \mathbb{R}^{3 \times 4}$ . The matrix order defines the size of the matrix.

Two matrices are equal if they have the same order and the entries located in both matrices in the same place are equal. A matrix is square whenever it has the same number of rows and columns. That is, it is an  $n \times n$  matrix.

From now on, we will use capital letters  $A, B, \dots$  to name matrices. The expression  $A_{m \times n}$  means that matrix  $A$  has dimensions  $m \times n$ . We will refer to the entries of a matrix using the same letter used to name the matrix but in lowercase and using subindexes to indicate the row and column the entry belongs to. The first subindex always represents the row, and the second is the column. For instance,  $a_{ij}$  Represents the matrix  $A$  entry that belongs to row  $i$  and to column  $j$ .

We call a matrix formed by a single row a row vector. We call a matrix formed by a single column a column vector. Whenever we speak of a vector without further specification,

na.<sup>1</sup> Para representar vectores, emplearemos letras minúsculas. Para representar sus elementos añadiremos a la letra que representa al vector un subíndice indicando la fila a la que pertenece el elemento.

<sup>1</sup>Esta identificación de los vectores como vectores columna no es general. La introducimos porque simplifica las explicaciones posteriores.

we consider it a column vector.<sup>1</sup> To name vectors, we will use lowercase letters. To name the entries of a vector, we will add a subindex to the letter representing the vector to show the row the entry belongs to.

<sup>1</sup>This identification of a vector with a column vector is by no means general. We introduce it because it simplifies the exposition.

$$a = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_i \\ \vdots \\ a_n \end{pmatrix}$$

Podemos asociar los puntos del plano con los vectores de dimensión dos. Para ello, usamos una representación cartesiana, en la que los elementos del vector son los valores de las coordenadas  $(x, y)$  del punto del plano que representan. Cada vector se representa gráficamente mediante una flecha que parte del origen de coordenadas y termina en el punto  $(x, y)$  representado por el vector. La figura 3.1 representa los vectores,

We can establish a relationship between the plane points and two-dimensional vectors. To do so, we use a Cartesian representation, in which the entries of a vector are the  $(x, y)$  coordinates of the point of the plane they represent. Each vector is drawn using an arrow that starts at the origin of the coordinates and ends at the point  $(x, y)$  represented by the vector. Figure 3.1 represents vectors,

$$a = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, b = \begin{pmatrix} 2 \\ -3 \end{pmatrix}, c = \begin{pmatrix} 0 \\ -2 \end{pmatrix}$$

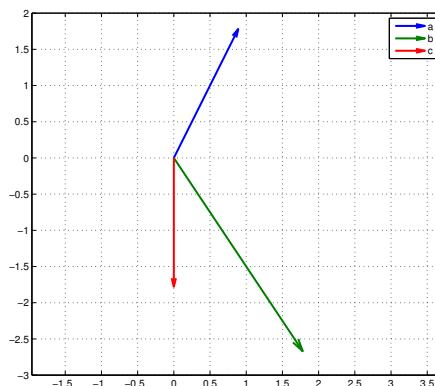


Figura 3.1: Representación gráfica de vectores en el plano  
Figure 3.1: Graphic of vectors on the plane

De modo análogo, podemos asociar vectores de dimensión tres con puntos en el espacio tridimensional. En este caso, los valores de los elementos del vector corresponden con la coordenadas  $(x, y, z)$  de los puntos en el espacio. La figura 3.2 muestra la representación gráfica en espacio tridimensional de los vectores,

$$a = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}, b = \begin{pmatrix} 2 \\ -3 \\ -1 \end{pmatrix}, c = \begin{pmatrix} 0 \\ -2 \\ 1 \end{pmatrix}$$

Likewise, we can associate vectors of dimension three with points in the 3D space. In this case, the vector entries represent the coordinates  $(x, y, z)$  of the points in the space. Figure 3.2 shows a graphic representation of vectors,

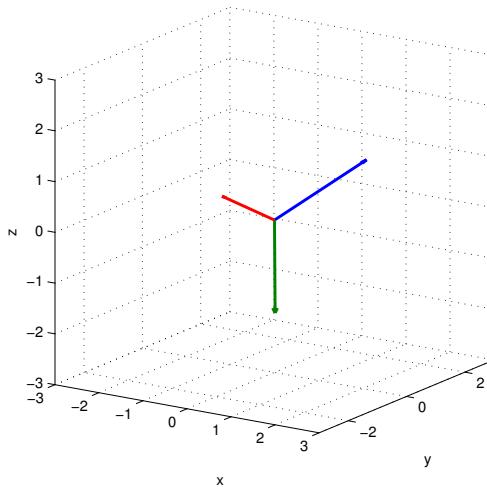


Figura 3.2: Representación gráfica de vectores en el espacio 3D

Figure 3.2: Graphic of vector in the 3D space

Evidentemente para vectores de mayor dimensión, no es posible obtener una representación gráfica. Si embargo muchas de las propiedades geométricas, observables en los vectores bi y tridimensionales, pueden extrapolarse a vectores de cualquier dimensión.

### 3.1.1. Vectores y matrices en Numpy.

**Matrices.** Una de las características más interesantes de Numpy, es la posibilidad de crear fácilmente matrices. Se pueden crear de diferentes maneras, la más elemental de todas ellas, emplea la función de Numpy `array` aplicada a una lista de filas de la matriz que se quiere construir. Cada fila debe ser a su vez

Obviously, it is impossible to get a graphic representation for vectors of larger dimensions. Nevertheless, many geometrical properties owned by 2D and 3D vectors can be applied to vectors of whatever dimension.

#### 3.1.1. Vectors and matrices in Numpy.

**Matrices.** One interesting feature of numpy is that it allows us to create matrices easily. There are several methods to create them, but using the Numpy function `array` with a Python list with the rows of the matrix we want to build is probably the easiest method. Each row should be, in turn, a list of numbers, i.e., matrix entries. Of course, to build a matrix, all

una lista de números. Evidentemente, para que se pueda construir la matriz, todas las filas deben tener el mismo número de elementos. El siguiente ejemplo muestra como construir una matriz de dos filas y tres columnas,

```
In [3]: import numpy as np

In [4]: A = np.array([[1,2,3],[4,5,6]])

In [5]: print(A)
[[1 2 3]
 [4 5 6]]

In [6]: L = [[1,2,3],[4,5,6]]

In [7]: B = np.array(L)

In [9]: print(L)
[[1, 2, 3], [4, 5, 6]]

In [10]: print(B)
[[1 2 3]
 [4 5 6]]
```

Lo primero que hacemos es importar Numpy, lo importamos usando como alias la abreviatura `np`, porque es más cómodo a la hora de llamar a funciones específicas de Numpy. Hemos construido dos matrices iguales `A` y `B`. En el primer caso hemos creado directamente dentro de la llamada a la función `array`, la lista a partir de la cual construimos la matriz. En el segundo caso, hemos construido primero una lista `L`, y luego hemos empleado dicha lista como variable de entrada de la función `array`. Si nos fijamos en las líneas [9] y [10], vemos como Python distingue la lista —todos sus elementos aparecen representados en la misma línea—, de la matriz en la que cada fila ocupa una línea distinta. Podemos construir matrices a partir de listas y variables ya definidas, siempre que seamos coherentes con el criterio de que cada fila se cree a partir de una lista y que todas las filas tengan los mismos elementos,

rows should have the same number of entries. The following example shows how to build a matrix of two rows and three columns.

First, we import Numpy; we do it using `np` as an alias. The reason is that `np` is shorter than `numpy`, and it eases the calls to specific Numpy functions. We have built two identical matrices `a` and `B`. In the first case, we have straightforwardly created the list needed to make the matrix inside the call to the function `array`. In the second case, we first created a list `L`, and then we used it as an input variable to the function `array`. Focusing on lines [9] and [10], we see how Python can tell a list and a matrix apart. For a list, Python represents all entries in the same line. For a matrix, each row is written in a different line. We can build matrices from lists and variables already defined as long as we follow the criteria that each row be created from a list and all rows have the same number of entries,

```
In [23]: a = 1; b= 2; c =3

In [24]: d =[4,5,6]
```

```
In [25]: C = np.array([[a,b,c],d])
In [26]: print(C)
[[1 2 3]
 [4 5 6]]

In [27]: C = np.array([[a,b,c],d,[7,8,9]])

In [28]: print(C)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

**Indexación.** Al igual que se hace en álgebra, Numpy es capaz de referirse a un elemento cualquiera de una matriz empleando índices para determinar su posición (fila y columna) dentro de la matriz.

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

Sin embargo, el criterio para referirse a un elemento concreto de una matriz, en Numpy está heredado de las listas: se indica el nombre de la variable que contiene la matriz y a continuación, entre corchetes y separados por una coma, el índice de su fila y después él de su columna **pero empezando a contar desde 0**. Es decir, la primera fila de una matriz de dimensión  $m \times n$  es la fila 0 y la última es la fila  $m - 1$ . De modo análogo su primera columna es la 0 y su última columna es la  $n - 1$ ,

**Indexing** As in algebra, in Numpy is also possible to refer any entry of a matrix using indexes to indicate its position (row and column) in the matrix.

However, the Numpy criteria to refer to a specific entry inside a matrix has been borrowed from Python Lists: we write the name of the matrix followed by the row and column indexes of the entry we are interested in, enclosed in a square bracket and separated by a comma. The point is that we **count the rows and columns starting at 0**. Thus, the first row of a matrix with dimensions  $m \times n$  is the row 0, and the last is the row  $m - 1$ . Likewise, its first column is column 0, and the last one is column  $n - 1$ ,

```
In [31]: print(C)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

```
In [32]: C[1,2]
Out[32]: 6
```

```
In [33]: C[0,0]
Out[33]: 1
```

Numpy puede seleccionar dentro de una matriz no solo elementos aislados, sino también submatrices completas. Para ello, emplea un símbolo reservado, el símbolo *dos puntos*

Inside a matrix, Numpy can select single entries and whole submatrices. To do so, it uses the colon : symbol as a special symbol. Using a fixed step, we use this symbol to co-

`:`. Este símbolo se emplea para recorrer valores desde un valor inicial hasta un valor final, con un incremento o paso fijo. La sintaxis es: `inicio:fin:paso`. Es importante tener en cuenta que Numpy detendrá la cuenta en el valor `stop-step`. Además, si no indicamos el tamaño del paso, Numpy tomará por defecto un paso igual a uno. En este caso basta emplear `start:stop`

ver a set of values from a start (initial) value to a stop (final) one. The syntax is simple: `start:stop:step`. Beware! Numpy will stop the count in the step `stop-step`. Besides, if we leave apart the step size, Numpy will take a step equal to one. In this case, the expression is just `start:stop`

```
In [87]: D = numpy.array([[1.,2.,3.],[4.,5.,6.],[-2,-3,0],[3,2,1]])

In [88]: print(D)
[[ 1.  2.  3.]
 [ 4.  5.  6.]
 [-2. -3.  0.]
 [ 3.  2.  1.]]

In [90]: D[0:1,0:3]
Out[90]: array([[1., 2., 3.]])

In [91]: D[0:2,0:2]
Out[91]:
array([[1., 2.],
       [4., 5.]])

In [92]: D[2:3,1:2]
Out[92]: array([-3.])

In [93]: D[3:4,0:3]
Out[93]: array([[-3., 2., 1.]])

In [94]: D[0:4,0:3]
Out[94]:
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.],
       [-2., -3.,  0.],
       [ 3.,  2.,  1.]])
```

```
In [95]: D[0:4,2:3]
Out[95]:
array([[3.],
       [6.],
       [0.],
       [1.]])
```

La línea [90], extrae un vector fila con los elementos de la primera fila de la matriz D. La fila [91] extrae una matriz de dimensión  $2 \times 2$  con los cuatro elementos de la esquina superior

Line [90] extracts a row vector from the first row of matrix D. Line [91] extracts a  $2 \times 2$  matrix using the four entries located on the left-up corner of the original, D, matrix. Line

izquierda de la matriz original. La fila [92] extrae un único elemento pero sigue siendo una matriz de dimensión  $1 \times 1$ , por tanto es distinto que si empleamos la indexación directa del elemento: `D[2, 1]`. La línea [93] nos devuelve un vector fila con la última fila de la matriz. La línea [94], nos devuelve de nuevo la matriz entera. Por último la línea [95] nos devuelve un vector columna con la primera columna de la matriz.

Hemos dicho que la línea [95] nos devuelve un vector columna. Bueno, sí y no; vamos a verlo más despacio.

**Vectores.** Cuando introducimos los vectores, distinguimos entre vectores filas y columna, definiéndolos como matrices de una sola fila o una sola columna. Sin embargo, en Numpy, se sigue un criterio distinto que permite generalizar el concepto de matriz asociándolo con el de tensor. Sin entrar en detalles<sup>2</sup>, podemos decir que un tensor es un objeto algebráico caracterizado por dos parámetros; el orden y la dimensión. Así un escalar  $a \in \mathbb{R}$  es un tensor de orden cero. Un vector  $b \in \mathbb{R}^n$  es un tensor de orden 1 y dimensión  $n$ . Una matriz  $A \in \mathbb{R}^{m \times n}$  es un tensor de orden dos y dimensiones  $m, n$ . Un tensor de orden 3,  $T \in \mathbb{R}^{n \times m \times l}$ , etc. Podemos asociar el orden al número mínimo de índices que necesitamos para definir de forma unívoca los elementos de un Tensor: para un escalar, no nos hace faltar ningún índice, solo tenemos un elemento que es el propio escalar, por tanto le asociamos orden cero. Para un vector es suficiente emplear un índice para definir sus elementos,  $b = (b_i)$ ,  $i = 1, \dots, n$ ,  $b \in \mathbb{R}^n$ . Para una matriz necesito dos índices,  $A = (a_{ij})$ ,  $i = 1, \dots, n$ ,  $j = 1, \dots, m$ ,  $A \in \mathbb{R}^{n \times m}$ . Para un tensor de orden tres necesitaría tres índices,  $T = (T_{ijk})$ ,  $i = 1, \dots, n$ ,  $j = 1, \dots, m$ ,  $k = 1, \dots, l$ ,  $T \in \mathbb{R}^{n \times m \times l}$  y así sucesivamente. Numpy permite definir estructuras de cualquier orden y dimensión que queramos. Pero nosotros nos vamos a limitar a vectores (orden 1) y matrices (orden 2). ¿Tiene sentido entonces distinguir entre vectores fila y columna? So-

---

<sup>2</sup>Una definición formal del concepto de tensor, queda fuera del alcance de estos apuntes.

[92] extracts a single matrix entry, but notice that it is a  $1 \times 1$  matrix. Thus, this result is different than the result achieved when we directly use the indexes of the entry: `D[2, 1]`. Line [93] returns a row vector with the matrix's last row entries. Lastly, line [95] returns a column vector with the matrix's last column entries.

We have said that line [95] returns a column vector. Well, yes and no; let's examine it in more detail.

**Vectors.** When we introduced vectors in the previous section, we distinguished between row and column vectors, defining them as matrices with a single row or column. Nevertheless, Numpy follows another criterion that allows us to generalise the idea of matrix, linking it with the concept of tensor. We do not get into details<sup>2</sup> and simply say that a tensor is an algebraic object defined by two parameters, its order, and its dimension. So, a scalar  $a \in \mathbb{R}$  is a tensor of order zero. A vector  $b \in \mathbb{R}^n$  is a tensor of order one and dimension  $n$ . A matrix  $A \in \mathbb{R}^{m \times n}$  is a tensor of order two and dimensions  $m, n$ . A third order tensor,  $T \in \mathbb{R}^{n \times m \times l}$ , etc. We can relate the order with the minimum number of indexes we need to univocally define the tensor entry: for a scalar, we don't need an index at all; we have only a single entry, the scalar itself. For this reason, we associate scalars with a zero-order tensor. For a vector, it is enough to use a single index to define its entries,  $b = (b_i)$ ,  $i = 1, \dots, n$ ,  $b \in \mathbb{R}^n$ . For a matrix, we need two indexes,  $A = (a_{ij})$ ,  $i = 1, \dots, n$ ,  $j = 1, \dots, m$ ,  $A \in \mathbb{R}^{n \times m}$ . For third-order tensors, we need three indexes,  $T = (T_{ijk})$ ,  $i = 1, \dots, n$ ,  $j = 1, \dots, m$ ,  $k = 1, \dots, l$ ,  $T \in \mathbb{R}^{n \times m \times l}$  and so on. In Numpy, we can define structures of whatever order and dimensions we want, but we will only use vectors (order 1) and matrices (order 2). Has, then, any sense to distinguish between row and column vector? Only if we always consider vectors as matrices (order 2) and dimensions  $1 \times n$  (row vector) or  $n \times 1$  (column vector). For Numpy, however, vectors and matrices are

---

<sup>2</sup>A formal definition of tensors is far beyond the scope of these notes.

lo si consideramos siempre los vectores como matrices (orden 2) y dimesiones  $1 \times n$  (vector fila) ó  $n \times 1$  (vector columna). Para Numpy, sin embargo, vectores y matrices son estructuras de distinto orden. Veamos con algunos ejemplo cómo se diferencian. Para verlo mejor podemos emplear la propiedad `shape` de los arrays en numpy. Dicha propiedad nos devuelve una tupla con las dimensiones del array, el número de elementos que contiene la tupla nos da el orden,

```
In [23]: D
Out[23]:
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.],
       [-2., -3.,  0.],
       [ 3.,  2.,  1.]])
```

```
In [24]: D.shape
Out[24]: (4, 3)
```

```
In [25]: D[1,1]
Out[25]: 5.0
```

```
In [26]: D[1,1].shape
Out[26]: ()
```

```
In [27]: D[1,1:2]
Out[27]: array([5.])
```

```
In [28]: D[1,1:2].shape
Out[28]: (1,)
```

```
In [29]: D[1:2,1:2]
Out[29]: array([[5.]])
```

```
In [30]: D[1:2,1:2].shape
Out[30]: (1, 1)
```

Empezamos con la matriz `D` de ejemplos anteriores. Para obtener sus dimensiones empleamos `D.shape`. El resultado es una tupla compuesta de dos elementos, puesto que es una matriz y, por tanto su orden es dos. El primer elemento no da la dimensión de sus columnas, es decir, el número de filas. El segundo elemento nos da la dimensión de sus filas, es decir el número de columnas.

En la línea [25] extraemos el elemento que ocupa la posición `[1, 1]`. En la [26] cuando tra-

structures of different order. Let's see some examples of their differences. To appreciate it better, we can use the Numpy arrays attribute `shape`, which gives us a tuple containing the dimensions of the array. The number of entries of the tuple tells us the order of the array,

We begin with the same matrix `D` we use in previous examples. To get its dimensions, we use `D.shape`. The result is a tuple of two entries because it is a matrix, and thus, its order is two. The first entry gives us its column dimension, that is, its number of rows. The second entry is its row dimension, that is, its number of columns.

In line [25] we extract the entry located at position `[1, 1]`. In line [26] when we try to get its dimension, Numpy returns an empty tu-

tamos de obtener sus dimensiones, nos da una tupla vacía, porque es un escalar y su orden es cero. En la línea [27] le hemos pedido a Numpy que nos de los elementos de la fila 1 de la matriz D que ocupan las columnas desde la 1 hasta la 1. Es decir, hacemos referencia al mismo elemento de la matriz, sin embargo el resultado no es exactamente el mismo. Nos ha devuelto un array con el elemento seleccionado. Cuando en la línea [28] pedimos sus dimensiones, obtenemos una tupla con un único elemento (1,). Es decir, el orden del array es 1, se trata de un vector, y tiene de dimensión 1, el vector solo tiene un elemento.

Por último, en la línea [29] volvemos a pedir a Python que nos de los elementos de la matriz D, que ocupan las filas desde la 1 hasta la 1 y las columnas desde la 1 hasta la 1, el resultado es ahora una matriz, podemos ver en la línea `out` [29] que el numero 5 aparece ahora encerrado entre dos pares de corchetes. Cuando en la línea [30], preguntamos por su `shape`, nos devuelve una tupla con dos elementos –el orden del array es 2, puesto que se trata de una matriz– y sus dimensiones son una fila y una columna, puesto que la matriz solo tiene un elemento.

Vamos a completar nuestro estudio de los arrays en Python, extrayendo ahora partes más grandes de la misma matriz D,

ple because it is a scalar and its order is zero. In line [27], we ask Numpy that retrieve the entries of row 1 of the matrix D that fill the columns 1 to 1. Thus, we are making reference to the same entry of the matrix as in the previous case. However, the result is not exactly the same. Numpy retrieves an array with the selected entry. When we ask for its dimensions in line [28] we get a tuple with a single entry (1,). Then, the order of the array is one. It is a vector with a single entry.

Finally, in line [29], we ask Numpy to retrieve the entries of matrix D with fill the rows from 1 to 1 and the columns from 1 to 1. Thus, the result is now a matrix, We can see in line `out` [29] that the number 5 is enclosed in two pair of square brackets. When in line [30], we ask for its `shape`, we get a tuple with two entries –now the order of the array is two, because it is a matrix– and their dimension are one row and one column as far as the matrix has a single entry.

We are going to complete our study on Numpy arrays, extracting larger parts of the same matrix,

```
In [9]: D
Out[9]:
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.],
       [-2., -3.,  0.],
       [ 3.,  2.,  1.]])
```

```
In [10]: D[1,:]
Out[10]: array([4., 5., 6.])
```

```
In [11]: D[:,1]
Out[11]: array([ 2.,  5., -3.,  2.])
```

```
In [12]: D[1:2,:]
Out[12]: array([[4., 5., 6.]])
```

```
In [13]: D[:,1:2]
Out[13]:
array([[ 2.],
       [ 5.],
      [-3.],
       [ 2.]])
```

En el primer caso, línea [10], hemos extraido toda la segunda fila de la matriz D, el resultado es un vector, por tanto tiene orden 1 y dimensión 3. En la línea [11] hemos extraído la primera columna y el resultado es de nuevo un vector, por tanto tiene orden 1 y la dimensión esta vez 4. En la línea [12] extraemos todas las columnas de las filas que van desde la segunda hasta la segunda. El resultado es una matriz, ya que el orden del array extraído es 2, y la dimensiones será 1 para las filas y 3 para las columnas. Es lo más parecido a un vector 'fila' que podemos obtener con Numpy. Por último, en la línea [13], extraemos todas las filas de las columnas que van desde la segunda hasta la segunda. El resultado es de nuevo una matriz, porque el array extraído tiene orden dos, pero las dimensiones son ahora 4 para las filas y 1 para las columnas, lo podemos identificar con un vector columna de los descritos antes.

Evidentemente, podemos tambien extraer de una matriz bloque (submatrices), indicando las filas y columnas que queremos extraer de la matriz original. por ejemplo,

```
In [20]: D
Out[20]:
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.],
      [-2., -3.,  0.],
       [ 3.,  2.,  1.]])
```

```
In [21]: D[1:4,1:3]
Out[21]:
array([[ 5.,  6.],
      [-3.,  0.],
       [ 2.,  1.]])
```

```
In [22]: D[1:3,0:2]
Out[22]:
array([[ 4.,  5.],
      [-2., -3.]])
```

In the first case, line [10] we get the whole second row of matrix D. The result is a vector and has order 1 and dimension 3. In line [11], we extract the first column of the matrix, and the result is again a vector. This time the order is 1, and the dimension is 4. In line [12] we extract all columns belonging to rows second to second. The result is a matrix because the order of the extracted array is two. The dimensions are 1 for the rows and 3 for the columns. It is the most similar to a 'row' vector we can get using Numpy. Lastly, in line [13], we extract all the rows belonging to columns second to second. the result is again a matrix because the array obtained has order two, but the dimensions are now 4 for the rows and 1 for the columns, we could identify it as a column vector of those described above.

Indeed, we can also extract a block matrix (submatrices), using indexes to define the rows and columns we want to obtain from the original matrix,

## 3.2. Operaciones matriciales

A continuación definiremos las operaciones matemáticas más comunes, definidas sobre matrices. Vamos a empezar por aquellas que se realizan elemento a elemento, entre aquellos elementos que ocupan la misma posición en las matrices que se operan,

**Suma.** La suma de dos matrices, se define como la matriz resultante de sumar los elementos que ocupan en ambas la misma posición. Solo está definida para matrices del mismo orden,

$$C = A + B$$

$$c_{ij} = a_{ij} + b_{ij}$$

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} 1 & 3 & 5 \\ 3 & 5 & 7 \\ 5 & 7 & 9 \end{pmatrix} + \begin{pmatrix} 0 & -1 & -2 \\ 1 & 0 & -1 \\ 2 & 1 & 0 \end{pmatrix}$$

La suma de matrices cumple las siguientes propiedades,

1. Asociativa:  $(A + B) + C = A + (B + C)$
2. Comutativa:  $A + B = B + A$
3. Elemento neutro:  $O_{n \times m} + A_{n \times m} = A_{n \times m}$   
El elemento neutro  $O_{n \times m}$  de la suma de matrices de orden  $n \times m$  es la matriz nula de dicho orden, —compuesta exclusivamente por ceros— .
4. Elemento opuesto: La opuesta a una matriz se obtiene cambiando de signo todos sus elementos,  $A_{op} = -A$

En numpy el signo  $+$  se también utiliza para representar la suma de matrices, por lo que la suma de dos matrices puede obtenerse directamente como,

## 3.2. Matrix Operations

In this section we will define the most common mathematical operation for matrices. We are going to begin for those operations that are carried out, entry by entry, between those entries which occupy the same place in the operating matrices.

**Addition.** Addition of two matrix, the result is a new matrix obtained adding the entries which occupy the same position in both matrices. It is defined only for matrix of the same dimensions,

Matrix addition fulfills the following properties,

1. Asociative:  $(A + B) + C = A + (B + C)$
2. Comutative:  $A + B = B + A$
3. Identity element:  $O_{n \times m} + A_{n \times m} = A_{n \times m}$   
The identity element  $O_{n \times m}$  of the addition of  $n \times m$  matrices is the null matrix of this dimensions, —an only zeros matrix— .
4. Inverse: we get the addition inverse of a matrix changing the signs of all its entries  $A_{inv} = -A$

In numpy, the symbol  $+$  also represents the matrix addition. Thus, you can use it to add two matrices in the same way you use it to add two numbers,

```
In [0]: import numpy as np
In [1]: A = np.array([[1,3,5],[2,4,6]])
In [2]: A
Out[2]:
array([[1, 3, 5],
       [2, 4, 6]])

In [3]: B = np.array([[3,-2,0],[1,-4,3]])

In [4]: A+B
Out[4]:
array([[4, 1, 5],
       [3, 0, 9]])
```

En numpy, podemos crear una matriz de cualquier orden, compuesta exclusivamente por ceros mediante el comando `numpy.zeros((m,n))`, donde  $m$  es el número de filas y  $n$  el de columnas de la matriz de ceros resultante. Si damos un único valor, `numpy.zeros(n)`, obtendremos un vector formado por  $n$  ceros.

We can use the Numpy command `numpy.zeros((m,n))` to create a matrix, of whatever dimension, with all its entries equal to zero.  $m$  stands for the number of rows and  $n$  for the number of columns of the zero matrix we want to create. If we use the command `numpy.zeros(n)` with a single value instead a tuple, then we will obtain a vector built up of  $n$  zeros..

```
In [375]: np.zeros(3)
Out[376]: array([0., 0., 0.])

In [377]: np.zeros((3,1))
Out[378]:
array([[0.],
       [0.],
       [0.]))

In [379]: np.zeros((1,3))
Out[380]: array([[0., 0., 0.]))

In [381]: np.zeros((3,3))
Out[382]:
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]))

In [383]: Z = np.zeros((2,3))

In [384]: A+Z
Out[385]:
array([[1., 3., 5.],
       [2., 4., 6.]])
```

```
In [386]: Aop = -A
```

```
In [387]: A+Aop
Out[388]:
array([[0, 0, 0],
       [0, 0, 0]])
```

**Multiplicación elemento a elemento.** No es propiamente una operación matricial. Dadas dos matrices  $A$  y  $B$  de las mismas dimensiones, si las multiplicamos elemento a elemento, obtendremos una nueva matriz  $C$  tal que,  $c_{i,j} = a_{i,j}b_{i,j}$ . Al igual que la suma, el producto elemento a elemento es asociativo y commutativo. En Numpy el simbolo para la multiplicación elemento a elemento es el asterisco  $*$ ,

```
In [49]: A
Out[49]:
array([[1, 2],
       [2, 0],
       [3, 5]])

In [50]: B
Out[50]:
array([[-3, 1],
       [0, 2],
       [1, -4]])

In [51]: A*B
Out[51]:
array([[ -3,    2],
       [  0,    0],
       [  3, -20]])
```

**División elemento a elemento.** Es análoga a la multiplicación que acabamos de ver. Si dividimos elemento a elemento una matrix  $A$  entre otra matriz  $B$ , ambas de las mismas dimensiones, obtenemos una matriz  $C$  cuyos elementos cumplen  $c_{ij} = a_{ij}/b_{ij}$ . El símbolo que se emplea es el mismo de la división ordinaria entre números.

En el ejemplo siguiente, dividimos entre sí las dos matrices del ejemplo anterior, es interesante observar como Python nos advierte de la división entre cero, y asigna al elemento en que se produce el valor `inf`.

**entry-wise multiplication** This is not a proper matrix (algebraic) operation. You can get the entry-wise product of two matrices,  $A$  and  $B$ , with the same dimensions to obtain a new matrix  $C$  which entries are  $c_{i,j} = a_{i,j}b_{i,j}$ . This product, like the matrix addition, is commutative and associative. In numpy the entry-wise multiplication symbol is the asterisk  $*$ ,

**entry-wise division.** It is verymuch alike the elemet-wise product. If we divide entry-wise a matrix  $A$  by another matrix  $B$ , both of the same dimensions, we get a new matrix  $C$  which entries are  $c_{ij} = a_{ij}/b_{ij}$ . For matrix entry-wise division, we use the same symbol as in the ordinamry number division.

In the following example, we divide the two matrix of the previous example, it is interesting to see how Python warnings us of a division by zero, and assigns to the resulting entry the value `inf`.

```
In [52]: A/B
/tmp/ipykernel_10098/713994841.py:1: RuntimeWarning: divide by zero encountered
in divide A/B
Out[52]:
array([[-0.33333333,  2.        ],
       [         inf,   0.        ],
       [ 3.        , -1.25     ]])
```

**Transposición.** Dada una matriz  $A$ , su transpuesta  $A^T$  se define como la matriz que se obtiene intercambiando sus filas con sus columnas.

**Transposition.** The transpose matrix  $A^T$  of a matrix  $A$  is the matrix we obtain interchanging its rows and columns.

$$\begin{aligned} A &\rightarrow A^T \\ a_{ij} &\rightarrow a_{ji} \\ A = \begin{pmatrix} 1 & -3 & 2 \\ 2 & 7 & -1 \end{pmatrix} &\rightarrow A^T = \begin{pmatrix} 1 & 2 \\ -3 & 7 \\ 2 & -1 \end{pmatrix} \end{aligned}$$

En numpy, la operación de transposición se indica mediante un punto y la letra T, `A.T`. Solo tiene sentido aplicarla para matrices; si transponemos un vector volvemos a obtener el mismo vector.

In Numpy the transpose matrix is obtained adding a point an the letter T to the matrix we wish to transpose, `A.T`. It only has sense for matrices. If we try to transpose a vector we will obtain the same vector again.

```
In [375]: A
Out[375]:
array([[1, 3, 5],
       [2, 4, 6]])

In [376]: A.T
Out[376]:
array([[1, 2],
       [3, 4],
       [5, 6]])

In [377]: B = np.array([1,2,3])

In [378]: B.T
Out[378]: array([1, 2, 3])

In [379]: C = np.array([[1,2,3]])

In [380]: C
Out[380]: array([[1, 2, 3]])
```

```
In [381]: C.T
Out[381]:
array([[1],
       [2],
       [3]])
```

Una matriz cuadrada se dice que es simétrica si coincide con su transpuesta,

A square matrix is antisimetric if it is equal to its transpose matrix,

$$\begin{aligned} A &= A^T \\ a_{ij} &= a_{ji} \\ A = A^T &= \begin{pmatrix} 1 & 3 & -3 \\ 3 & 0 & -2 \\ -3 & -2 & 4 \end{pmatrix} \end{aligned}$$

Una matriz cuadrada es antisimétrica cuando cumple que  $A = -A^T$ . Cualquier matriz cuadrada se puede descomponer en la suma de una matriz simétrica más otra antisimétrica.

La parte simétrica puede definirse como,

A square matrix is antisimetric when it meets that  $A = -A^T$ . Any square matrix can be split in the sum of two matrix, one simetric and another antisimetric

The simetric part can be defined as,

$$A_S = \frac{1}{2} (A + A^T)$$

y la parte antisimétrica como,

An the antisimetric part as,

$$A_A = \frac{1}{2} (A - A^T)$$

Así, por ejemplo,

For instance,

$$A = A_S + A_A \rightarrow \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} 1 & 3 & 5 \\ 3 & 5 & 7 \\ 5 & 7 & 9 \end{pmatrix} + \begin{pmatrix} 0 & -1 & -2 \\ 1 & 0 & -1 \\ 2 & 1 & 0 \end{pmatrix}$$

Por último, la transpuesta de la suma de matrices cumple,

The transpose of the addition of matrix meets,

$$(A + B)^T = A^T + B^T$$

**Producto de una matriz por un escalar.** El producto de una matriz  $A$  por un número  $b$  es una matriz del mismo orden que  $A$ , cuyos elementos se obtienen multiplicando los elementos de  $A$  por el número  $b$ ,

**Product of a matrix and a scalar.** The product of a matrix  $A$  for a number  $b$  is a matrix with the same dimensions as  $A$ . We obtain the entry of this product multiplying the elemetns of  $A$  by the number  $b$ ,

$$\begin{aligned} C &= b \cdot A \rightarrow c_{ij} = b \cdot a_{ij} \\ 3 \cdot \begin{pmatrix} 1 & -2 & 0 \\ 2 & 3 & -1 \end{pmatrix} &= \begin{pmatrix} 3 & -6 & 0 \\ 6 & 9 & -3 \end{pmatrix} \end{aligned}$$

En Python, el símbolo `*` se emplea también para representar el producto de una matriz por un número

```
In [391]: A
Out[391]:
array([[ 3, -5, -2,  1],
       [ 2,  3,  4,  5]])

In [394]: A*5
Out[394]:
array([[15, -25, -10,  5],
       [10,  15,  20, 25]])
```

**Producto escalar de dos vectores.** Dados vectores de la misma dimensión  $m$  se define su producto escalar como,

$$a \cdot b = \sum_{i=1}^n a_i b_i$$

$$\begin{pmatrix} 1 \\ 3 \\ 4 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ -2 \\ 0 \end{pmatrix} = 1 \cdot 1 + 3 \cdot (-2) + 4 \cdot 0 = -5$$

El resultado de producto escalar de dos vectores, es siempre un número; se multiplican los entre sí los elementos de los vectores que ocupan idénticas posiciones y se suman los productos resultantes.

**Producto matricial** El producto de una matriz de orden  $n \times m$  por una matriz  $m \times l$ , es una nueva matriz de orden  $n \times l$ , cuyos elementos se obtiene de acuerdo con la siguiente expresión,

$$P = A \cdot B \rightarrow a_{ij} = \sum_{t=1}^m a_{it} b_{tj}$$

Por tanto, el elemento de la matriz producto que ocupa la fila  $i$  y la columna  $j$ , se obtiene multiplicando por orden los elementos de la fila  $i$  de la matriz  $A$  con los elementos correspondientes de la columna  $j$  de la matriz  $B$ , y sumando los productos resultantes

Para que dos matrices puedan multiplicarse es imprescindible que el número de columnas de la primera matriz coincida con el númer-

In Python we also use the symbol `*` to represent the product of a matrix by a number.

**Dot product of two vectors** For two vectors of the same dimension, we define the dot product as,

The dot product result is always a number; we multiply the entry which take the same place in both vectors and then, we sum the resulting products.

**Matrix product** The matrix product of a  $n \times m$  matrix by a  $m \times l$  matrix is a new matrix which dimensions are  $n \times l$ . We obtain the entries of the resulting matrix according with the following expression,

So, we get the entry of the product matrix, which occupies the row  $i$  and the column  $j$ , by multiplying in turn the entries of the row  $i$  of matrix  $A$  with the entries of the column  $j$  of matrix  $B$  and adding up the resulting products.

Two matrix can be multiplied only if the number of columns in the first matrix dimension is equal to the second matrix row dimension.

ro de filas de la segunda.

Podemos entender la mecánica del producto de matrices de una manera más fácil si consideramos la primera matriz como un grupo de vectores fila,

$$\begin{aligned} A_1 &= (a_{11} \quad a_{12} \quad \cdots a_{1n}) \\ A_2 &= (a_{21} \quad a_{22} \quad \cdots a_{2n}) \\ &\vdots \\ A_m &= (a_{m1} \quad a_{m2} \quad \cdots a_{mn}) \end{aligned} \rightarrow A = \begin{pmatrix} a_{11} & a_{12} & \cdots a_{1n} \\ a_{21} & a_{22} & \cdots a_{2n} \\ \vdots & \vdots & \cdots \\ a_{m1} & a_{m2} & \cdots a_{mn} \end{pmatrix}$$

y la segunda matriz como un grupo de vectores columna,

$$B_1 = \begin{pmatrix} b_{11} \\ b_{21} \\ \vdots \\ b_{n1} \end{pmatrix} B_2 = \begin{pmatrix} b_{12} \\ b_{22} \\ \vdots \\ b_{n2} \end{pmatrix} \cdots B_3 = \begin{pmatrix} b_{1m} \\ b_{2m} \\ \vdots \\ b_{nm} \end{pmatrix} \rightarrow B = \begin{pmatrix} b_{11} & b_{12} & \cdots b_{1n} \\ b_{21} & b_{22} & \cdots b_{2n} \\ \vdots & \vdots & \cdots \\ b_{m1} & b_{m2} & \cdots b_{mn} \end{pmatrix}$$

Podemos ahora considerar cada elemento  $p_{ij}$  de la matriz producto  $P = A \cdot B$  como el producto escalar del vector fila  $A_i$  por el vector columna  $B_j$ ,  $p_{ij} = A_i \cdot B_j$ . Es ahora relativamente fácil, deducir algunas de las propiedades del producto matricial,

1. Para que dos matrices puedan multiplicarse, es preciso que el número de columnas de la primera coincida con el número de filas de la segunda. Además la matriz producto tiene tantas filas como la primera matriz y tantas columnas como la segunda.
2. El producto matricial no es conmutativo. En general  $A \cdot B \neq B \cdot A$
3.  $(A \cdot B)^T = B^T \cdot A^T$

En Numpy se emplea el símbolo  $\circledast$  para representar el producto escalar, el producto de un vector por una matriz y el producto matricial. En todos los casos, es preciso que las dimensiones internas de los objetos que se multiplican coincidan. A continuación se muestran algunos ejemplos,

sion.

We may understand better the matrix product mechanism if we consider the first matrix as a group of row vectors,

An the second matrix as a group of column vectors.

we may consider the consider each entry  $p_{ij}$  of the product matrix  $P = A \cdot B$  as a escalar product of the row vector  $A_i$  and the column vector  $B_j$ ,  $p_{ij} = A_i \cdot B_j$ . Now we can easily find out some interesting properties of the matrix product.

1. Two matrices can be multiplied only if the number of columns of the first matrix meet the number of columns of the second one. Besides, the product matrix has so meny row as the first matrix and so many columns as the second one.
2. Matrix produc it is not commutative. In general,  $A \cdot B \neq B \cdot A$
3.  $(A \cdot B)^T = B^T \cdot A^T$

Numpy uses the symbol  $\circledast$  to represent the scalar product, the product of a matrix and a vector and the matrix product. In any case, it is necessary that the inner dimensions of factor objects meet. Next we show some matrix multiplication examples,

```
In [382]: a = np.array([1,2,3,4])

In [383]: b = np.array([-1,2,0,-3])

In [384]: a@b
Out[384]: -9

In [386]: A = np.array([[3,-5,-2,1],[2,3,4,5]])

In [387]: A
Out[387]:
array([[ 3, -5, -2,  1],
       [ 2,   3,   4,  5]])

In [388]: A@b
Out[388]: array([-16, -11])

In [389]: b@A.T
Out[389]: array([-16, -11])

In [390]: b@A
Traceback (most recent call last):

Cell In[390], line 1
      b@A

ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0,
with gufunc signature (n?,k),(k,m?)->(n?,m?) (size 2 is different from 4)

In [398]: B
Out[398]:
array([[ 0, -1],
       [-1,  0],
       [ 2,  3],
       [ 3,  4]])

In [399]: A@B
Out[399]:
array([[ 4, -5],
       [20, 30]])

In [400]: B@A
Out[400]:
array([[ -2, -3, -4, -5],
       [ -3,   5,   2, -1],
       [12, -1,   8, 17],
       [17, -3, 10, 23]])
```

En la Línea In [384] se ha calculado el producto escalar de los vectores  $\mathbf{a}$  y  $\mathbf{b}$ . En este caso, la única condición requerida es que tengan la misma dimensión. En la línea In [388] se calcula el producto de la matriz  $\mathbf{A}$  por el vector  $\mathbf{b}$ . El requisito ahora es que la dimensión de la matriz ( $2 \times 4$ ), que corresponde al número de columnas, coincida con la única dimensión del vector (4). En la línea In [389] calculamos el producto del vector  $\mathbf{b}$  por la transpuesta de la matriz  $\mathbf{A}$ . La operación es posible porque la única dimensión del vector y coincide con la dimensión de la matriz transpuesta (4), que corresponde con a número de filas. Sin embargo, no es posible calcular el producto del vector  $\mathbf{b}$  por la matriz  $\mathbf{A}$ , ya que la dimensión del vector no coincide con la primera dimensión de la matriz ( $2 \times 4$ ). En la línea In [399] hemos multiplicado las matrices  $\mathbf{A}$  ( $2 \times 4$ ) por la matriz  $\mathbf{B}$  ( $4 \times 2$ ). Como coinciden las dimensiones “internas”, –número de columnas de la primera matriz con número de filas de la segunda– La operación puede llevarse a cabo. Si invertimos el orden de las matrices, (línea In [400]) el producto también es posible, ya que en también coincidirán las dimensiones “internas”. Sin embargo, es fácil ver que los resultados son complementariamente distintos.

**Matriz identidad** La matriz identidad de orden  $n \times n$  se define como:

$$I_n = \begin{cases} i_{ll} = 1 \\ i_{kj} = 0, k \neq j \end{cases}$$

Es decir, una matriz en la que todos los elementos que no pertenecen a la diagonal principal son 0 y los elementos de la diagonal principal son 1. Por ejemplo,

$$I_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

La matriz identidad  $I_n$  es el elemento neutro del producto de matrices cuadradas de orden  $n \times n$ ,

In line In[384] we calculated the scalar product of vectors  $\mathbf{a}$  and  $\mathbf{b}$ . In this case, the only requirement is that both vectors have the same dimension. In line In [388] we calculated the product of matrix  $\mathbf{A}$  and the vector  $\mathbf{b}$ . Now the requirement is the dimension of the matrix ( $2 \times 4$ ), corresponding to the number of columns, meets the the single dimension of the vector (4). In line [389], we multiplied vector  $\mathbf{b}$  with the transpose of matrix  $\mathbf{A}$ . This operation can be carried out because the single dimension of vector  $\mathbf{b}$  is the same as the dimension of the transpose of the matrix ( $4 \times 2$ ), which corresponds with the number of rows of the matrix. However, we cannot multiply vector  $\mathbf{b}$  by matrix  $\mathbf{A}$  because the dimension of the vector does not meet the first dimension of the matrix ( $2 \times 4$ ). In line In [399] we multiplied matrix  $\mathbf{A}$  ( $2 \times 4$ ) by matix  $\mathbf{B}$  ( $4 \times 2$ ). We can carry out the operation because the “inner” dimensions of the matrices –number of columns of the first matrix and number of rows of the second one–, meet. If we invert the order of the matrices (line In [400]), we can still multiply the matrix because, in this particular case, the new “inner” dimensions also meet. However, the result is entirely different

**The identity matrix** The identity matrix of dimension  $n \times n$  is defined as:

So, an identity matrix has any entry off the main diagonal equal to 0 and all entries on the main diagonal equal to 1, for example,

The identity matrix  $I_n$  is the identity element of the square  $n \times n$  matrices product.

$$A_{n \times n} \cdot I_n = I_n \cdot A_{n \times n}$$

Además,

Besides,

$$\begin{aligned} A_{n \times m} \cdot I_m &= A_{n \times m} \\ I_n \cdot A_{n \times m} &= A_{n \times m} \end{aligned}$$

En Numpy se emplea el comando `eye(n)` para construir la matriz identidad de dimensiones  $n \times n$ ,

```
In [381]: np.eye(4)
Out[381]:
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

Una matriz cuadrada se dice que es ortogonal si cumple,

$$A^T \cdot A = I$$

**Norma de un vector.** La longitud euclídea, módulo, norma 2 o simplemente norma de un vector se define como,

$$\|x\|_2 = \|x\| = \sqrt{x \cdot x} = \sqrt{x^T x} = \sqrt{x_1^2 + x_2^2 + \dots + x_n^2} = \left( \sum_{i=1}^n x_i^2 \right)^{\frac{1}{2}}$$

Constituye la manera usual de medir la longitud de un vector. Tiene una interpretación geométrica inmediata a través del teorema de Pitágoras: nos da la longitud del segmento que representa al vector. La figura 3.3 muestra dicha interpretación, para un vector bidimensional.

La norma de un vector en Numpy se obtiene empleando el comando `norm` que pertenece al submódulo `linalg`,

```
In [422]: a
Out[422]: array([1, 2, 3, 4])
```

```
In [425]: np.linalg.norm(a)
Out[425]: 5.477225575051661
```

A partir de la norma de un vector es posible obtener una expresión alternativa para el producto escalar de dos vectores,

In Numpy, we can use the command `eye(n)` to build the identity matrix of dimension  $n \times n$ .

An orthogonal matrix is a square matrix that fulfills,

**Vector norm** The Euclidean length, module, norm two or just norm of a vector is defined as follows,

The vector norm represents the usual method to measure its length. It has a direct geometrical interpretation using Pithagoras' theorem: the norm is the length of the segment that represents the vector. Figure 3.3 shows such an interpretation for a bi-dimensional vector.

$$a \cdot b = \|a\| \|b\| \cos \theta$$

We can derive an alternative expression of the scalar product of two vectors, using the vector norm

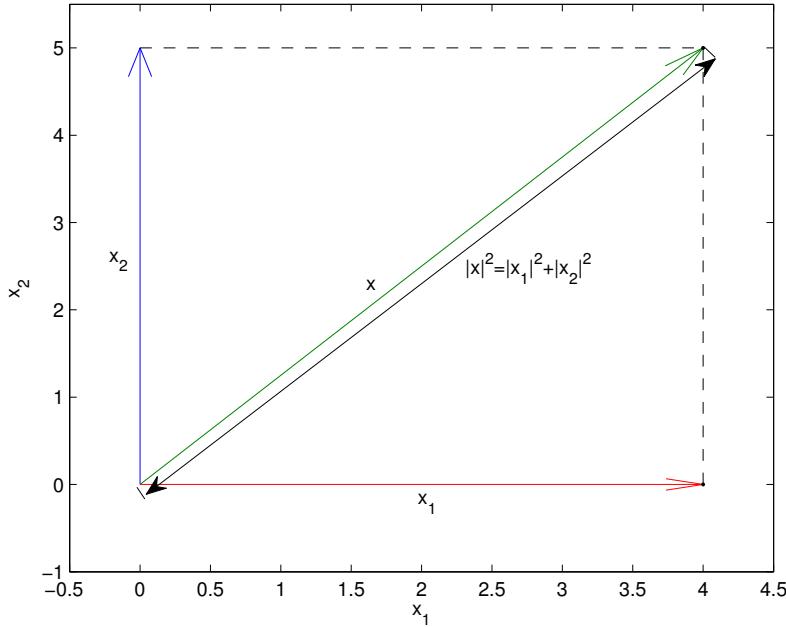


Figura 3.3: interpretación geométrica de la norma de un vector  
Figure 3.3: Geometrical interpretation of the norm of a vector.

Donde  $\theta$  representa el ángulo formado por los dos vectores.

where  $\theta$  is the angle between the two vectors.



Aunque se trate de la manera más común de definir la norma de un vector, la norma 2 no es la única definición posible,

*Norma 1:* Se define como la suma de los valores absolutos de los elementos de un vector,

$$\|x\|_1 = |x_1| + |x_2| + \dots + |x_n|$$

*Norma p:* Es una generalización de la norma 2,

$$\|x\|_p = \sqrt[p]{|x_1^p| + |x_2^p| + \dots + |x_n^p|} = \left( \sum_{i=1}^n |x_i^p| \right)^{\frac{1}{p}}$$

*Norma  $\infty$ :* se define como el mayor ele-

Being the norm 2 the most common way to define a vector norm is by no means the only possible definition,

*Norm 1:* defined as the sum of the absolute values of vector entries.

*P-Norm:* It is a generalization of the norm 2

*$\infty$  Norm:* The entry with the largest ab-

mento del vector valor absoluto,

solute value

$$\|x\|_{\infty} = \max \{|x_i|\}$$

*Norma  $-\infty$ :* el menor elemento del vector en valor absoluto,

$-\infty$  Norm: The entry with the smallest absolute value.

$$\|x\|_{-\infty} = \min \{|x_i|\}$$

En Numpy la norma de un vector puede obtenerse mediante el comando `norm(v,p)`. La primera variable de entrada debe ser un vector y la segunda el tipo de norma que se desea calcular. Si se omite la segunda variable de entrada, el comando devuelve la norma 2. Para las normas  $\infty$  y  $-\infty$  Se emplea el símbolo especial `inf` de Numpy. A continuación se incluyen varios ejemplo de utilización,

In Numpy the vector norm can be obtained using the command `norm(v,p)`. The first variable should be a vector and the second one represents the norm we want to calculate. If we leave out the second variable, the function returns norm 2 by default. For norms  $\infty$  y  $-\infty$  we use the symbol `inf` to represent the norm kind.

```
In [422]: a
Out[422]: array([1, 2, 3, 4])

In [423]: np.linalg.norm(a,1)
Out[423]: 10.0

In [424]: np.linalg.norm(a,2)
Out[424]: 5.477225575051661

In [425]: np.linalg.norm(a)
Out[425]: 5.477225575051661

In [426]: np.linalg.norm(a,4)
Out[426]: 4.337613136533361

In [427]: np.linalg.norm(a,np.inf)
Out[427]: 4.0

In [428]: np.linalg.norm(a,-np.inf)
Out[428]: 1.0
```

En general, una norma se define como una función  $\mathbb{R}^n \rightarrow \mathbb{R}$ , que cumple,

In general, a norm can be define as functions  $\mathbb{R}^n \rightarrow \mathbb{R}$ , which satisfies,

$$\begin{aligned}\|x\| &\geq 0, \|x\| = 0 \Rightarrow x = 0 \\ \|x + y\| &\leq \|x\| + \|y\| \\ \|\alpha x\| &= |\alpha| \|x\|, \alpha \in \mathbb{R}\end{aligned}$$



Llamaremos vectores unitarios  $u$ , a aquellos para los que se cumple que  $\|u\| = 1$ .

Dos vectores  $a$  y  $b$  son ortogonales si cumplen que su producto escalar es nulo,  $a^T b = 0 \Rightarrow a \perp b$ . Si además ambos vectores tienen módulo unidad, se dice entonces que los vectores son ortonormales. Desde el punto de vista de su representación geométrica, dos vectores ortogonales, forman entre sí un ángulo recto.

**Traza de una matriz.** La traza de una matriz cuadrada, se define como la suma de los elementos que ocupan su diagonal principal

$$\text{Tr}(A) = \sum_{i=1}^n a_{ii}$$

$$\text{Tr} \left( \begin{pmatrix} 1 & 4 & 4 \\ 2 & -2 & 2 \\ 0 & 3 & 6 \end{pmatrix} \right) = 1 - 2 + 6 = 5$$

La traza de la suma de dos matrices cuadradas  $A$  y  $B$  del mismo orden, coincide con la suma de las trazas de  $A$  y  $B$ ,

$$\text{tr}(A + B) = \text{tr}(A) + \text{tr}(B)$$

Dada una matriz  $A$  de dimensión  $m \times n$  y una matriz  $B$  de dimensión  $n \times m$ , se cumple que,

$$\text{tr}(AB) = \text{tr}(BA)$$

En Python, puede obtenerse directamente el valor de la traza de una matriz  $A$ , mediante la función `np.trace(A)`. En este caso, se trata de un método asociado a cualquier array por lo que también puede expresarse como `A.trace()`

Vectors  $u$  with  $\|u\| = 1$  are called unitary vectors.

Two vectors  $a$  and  $b$  are orthogonal if their scalar product is zero,  $a^T b = 0 \Rightarrow a \perp b$ . Besides, if the vectors have norm one them, they are called orthonormal vectors. From the point of view of the geometrical representation, the angle between two orthogonal vectors is a square angle.

**Matrix's trace.** The trace of a square matrix is the sum of the entries located in the matrix's main diagonal.

The trace of the sum of two square, same dimension matrices  $A$  and  $B$  is equal to the sum of  $A$  trace plus  $B$  trace,

For two matrices  $A$  of dimensions  $m \times n$  and  $B$  of dimensions  $n \times m$  is always true that,

In Numpy, the value of a matrix's trace can be calculated using the function `np.trace(A)`. In this case, the function is a method associated to any matrix and can be also expressed as `A.trace()`

```
In [197]: A = np.array([[1,2,3],[3,-2,3],[0,2,-1]])
```

```
In [198]: A
```

```
Out[198]:
```

```
array([[ 1,  2,  3],
       [ 3, -2,  3],
       [ 0,  2, -1]])
```

```
In [199]: np.trace(A)
```

```
Out[199]: -2
```

```
In [200]: A.trace()
```

```
Out[200]: -2
```

**Determinante de una matriz.** En Numpy, el determinante de una matriz se calcula empleando la función `det`, del submódulo `linalg`. Así, para calcular el determinante de la matriz  $A$  del ejemplo anterior,

**Matrix's determinant.** In Numpy, a matrix's determinant can be calculated using the function `det`, del submódulo `linalg`. We can calculate the determinant of matrix  $A$  of the previous example,

```
In [203]: np.linalg.det(A)
Out[203]: 20.000000000000007
```



El determinante de una matriz  $A$ , se representa habitualmente como  $|A|$  o, en ocasiones como  $\det(A)$ . Para poder definir el determinante de una matriz, necesitamos antes introducir una serie de conceptos previos. En primer lugar, si consideramos un escalar como una matriz de un solo elemento, el determinante sería precisamente el valor de ese único elemento,

$$A = (a_{11}) \rightarrow |A| = a_{11}$$

En álgebra lineal, se denomina menor primero,  $M_{ij}$  de una matriz  $A$ , al determinante de la matriz que resulta de eliminar de la matriz  $A$  la fila  $i$  y la columna  $j$ . Por ejemplo,

The determinant of a matrix,  $A$  is usually represented as  $|A|$  and occasionally as  $\det(A)$ . To define the determinant of a matrix, we need first to establish some previous definitions. First, if we consider a scalar as a matrix of a single entry, the determinant of the matrix is this single entry.

In linear algebra, a first minor  $M_{ij}$  of a square matrix  $A$  is the determinant of the matrix obtained by removing simultaneously row  $i$  and column  $j$  of  $A$ . For instance,

$$A = \begin{pmatrix} 1 & 0 & -2 \\ 3 & -2 & 3 \\ 0 & 6 & 5 \end{pmatrix}, M_{23} = \det \begin{pmatrix} 1 & 0 \\ 0 & 6 \end{pmatrix}$$

$$M_{32} = \det \begin{pmatrix} 1 & -2 \\ 3 & 3 \end{pmatrix}, M_{33} = \det \begin{pmatrix} 1 & 0 \\ 3 & -2 \end{pmatrix} \dots$$

El cofactor  $C_{ij}$  de un elemento  $a_{ij}$  de la matriz  $A$ , se define a partir del menor primero  $M_{ij}$ , que corresponde precisamente a la fila y columna que contiene a  $a_{ij}$ ,

$$C_{ij} = (-1)^{i+j} M_{ij}$$

Podemos ahora definir el determinante de una matriz  $A$  cuadrada de orden  $n$ , empleando la fórmula de Laplace,

$$|A| = \sum_{j=1}^n a_{ij} C_{ij}$$

o alternativamente,

$$|A| = \sum_{i=1}^n a_{ij} C_{ij}$$

En el primer caso, se dice que se ha desarrollado el determinante a lo largo de la fila  $i$ . En el segundo caso, se dice que se ha desarollo a lo largo de la columna  $j$ .

La fórmula de Laplace obtiene el determinante de una matriz de orden  $n \times n$  a partir del cálculo de los menores complementarios de los elementos de una fila o columna; los determinates the  $n$  matrices de orden  $(n-1) \times (n-1)$ . A su vez, podríamos calcular cada menor complementario, aplicando la fórmula de Laplace y así sucesivamente hasta llegar a matrices de orden  $2 \times 2$ . Para una matriz  $2 \times 2$ , si desarrollamos por la primera fila obtenemos su determinante como,

$$\begin{aligned} A &= \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \\ |A| &= \sum_{j=1}^2 a_{1j} C_{1j} = a_{11} C_{11} + a_{12} C_{12} \\ &= a_{11}(-1)^{1+1} |M_{11}| + a_{12}(-1)^{1+2} |M_{12}| \\ &= -a_{11}a_{22} + a_{12}a_{21} \end{aligned}$$

y si desarrollamos por la segunda columna,

$$\begin{aligned} A &= \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \\ |A| &= \sum_{j=1}^2 a_{i2} C_{i2} = a_{12} C_{12} + a_{22} C_{22} \\ &= a_{12}(-1)^{1+2} |M_{12}| + a_{22}(-1)^{2+2} |M_{22}| \\ &= -a_{12}a_{21} + a_{22}a_{12} \end{aligned}$$

We define the cofactor  $C_{ij}$  of a matrix  $A_{ij}$  entry,  $a_{ij}$ , using the first minor  $M_{ij}$  obtained removing the row and column  $a_{ij}$  belongs to.

Now, we can define the determinant of a  $n$  dimensional square matrix  $A$ , using the Laplace's formula,

or also as,

In the first case, we say that we develop the determinant along the row  $i$ . In the second one, we say that we have developed the determinant along the column  $j$

Laplace's formula gets the determinant of a Matrix of dimensions  $n \times n$  using the first minors of a matrix row or column, i.e. the determinants of  $n$  matrix of order  $(n-1) \times (n-1)$ . In turn, we can calculate each minor using Laplace's formula and so on, till we arrive at matrices of dimension  $2 \times 2$ . We obtain the determinant of a  $2 \times 2$  matrix developing it along its first row,

And if we develop using the secon column,

Para una matriz de dimensión arbitraria  $n \times n$ , el determinante se obtiene aplicando recursivamente la fórmula de Laplace,

$$\begin{aligned} |A| &= \sum_{j=1}^n a_{ij} C_{ij} = \sum_{j=1}^n a_{ij} (-1)^{i+j} \left| M_{ij}^{(n-1) \times (n-1)} \right| \\ \left| M_{ij}^{(n-1) \times (n-1)} \right| &= \sum_{k=1}^{n-1} m_{lk} C_{lk} = \sum_{k=1}^{n-1} m_{lk} (-1)^{l+k} \left| M_{lk}^{(n-2) \times (n-2)} \right| \\ &\vdots \\ \left| M_{st}^{1 \times 1} \right| &= (-1)^{s+t} m_{st} \end{aligned}$$

Así, por ejemplo, podemos calcular el determinante de la matriz,

$$A = \begin{pmatrix} 1 & 0 & -2 \\ 3 & -2 & 3 \\ 0 & 6 & 5 \end{pmatrix}$$

desarrollándolo por los elementos de la primera columna, como,

$$\begin{aligned} |A| &= 1 \cdot (-1)^2 \cdot \begin{vmatrix} -2 & 3 \\ 6 & 5 \end{vmatrix} + 3 \cdot (-1)^3 \cdot \begin{vmatrix} 0 & -2 \\ 6 & 5 \end{vmatrix} + 0 \cdot (-1)^4 \cdot \begin{vmatrix} 0 & -2 \\ -2 & -3 \end{vmatrix} \\ &= 1 \cdot (-1)^2 \cdot [(-2) \cdot 5 - 6 \cdot 3] + 3 \cdot (-1)^3 \cdot [0 \cdot 5 - 6 \cdot (-2)] + 0 \cdot (-1)^4 \cdot [0 \cdot 3 - (-2) \cdot (-2)] = -64 \end{aligned}$$

Podemos programar en Python una función recursiva<sup>3</sup> que calcule el determinante de una matriz de dimensiones  $n \times n$ .

<sup>3</sup>El método no es especialmente eficiente pero ilustra el uso de funciones recursivas.

For a matrix of arbitrary dimension we obtain the determinant using the Laplace's formula recursively,

For instance, we can calculate the determinant of the following matrix,

Developing by the first column entries,

We can now program a recursive function to calculate the determinant of any  $n \times n$  dimensions.

<sup>3</sup>The method is not particularly efficient but it helps to show the use of recursive functions

---

determinante.py

---

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3 """
4 Created on Thu May 23 14:50:48 2024
5 Este modulo implementa la funcion dumbdet que calcula el determinante de
6 una matriz empleando la formula de Laplace. La función es recursiva,
7 (se llama a si misma sucesivamente para calcular los cofactores necesarios).
8 Desarrolla siempre por los elementos de la primera columna.
9 (Es un prodigo de ineficiencia numérica, pero permite manejar bucles
10 y funciones recursivas, así que supongo que puede ser útil para los que
11 empiezan a programar).
12 un posible ejercicio para ver lo malo que es el método, consiste ir
13 aumentando la dimensión de la matriz y comparar lo que lo tarde en
14 calcular el determinante con lo que tarda la función de numpy.linalg.det...
15 this module implements the function poordet which calculates a matrix
16 determinant using the Laplace's formulae. It is a recursive function,
```

```

17 (it calls itself on and on to calculate the cofactor needed to get the
18 determinat). It always developed the formula using the elements of the matrix
19 first column.
20 (It is the most inefficient function ever written, but may be an example of loops
21 and recursive functions for beginners)
22 @author: juan
23 """
24 import numpy as np
25 def dumbdet(A):
26
27 #first we check the matrix is square
28 #primero comprobamos si la matriz es cuadrada
29 sz = A.shape[0]
30 d = 0 #variable to save the result/ Variable para guardar el resultado
31
32 if sz != A.shape[1]:
33     print('La matriz no es cuadrada')
34     print('The matrix is not square')
35     d = []
36 elif sz == 1:
37     return A[0,0]
38 else:
39
40     for i in range(0,sz):
41         N = np.delete(A,i,0)
42         d = (-1)**(i+2)*A[i,0]*dumbdet(N[:,1:sz])+d
43
44
45 return d

```

---



Entre las propiedades de los determinantes, destacaremos las siguientes,

1. El determinante del producto de un escalar  $a$  por una matriz  $A$  cumple,

$$|a \cdot A| = a^n \cdot |A|$$

2. El determinante de una matriz es igual al de su traspuesta,

$$|A| = |A^T|$$

3. El determinante del producto de dos matrices es igual al producto de los determinantes,

$$|A_{n \times n} \cdot B_{n \times n}| = |A_{n \times n}| \cdot |B_{n \times n}|$$

Among determinants' properties we will point out the following ones,

1. The determinant of a scalar  $a$  by a matrix  $A$  is,

$$|a \cdot A| = a^n \cdot |A|$$

2. The determinant of a matrix transpose equals the determinant of the matrix,

$$|A| = |A^T|$$

3. The determinant of a matrix product equals the product of their determinants,

$$|A_{n \times n} \cdot B_{n \times n}| = |A_{n \times n}| \cdot |B_{n \times n}|$$

Una matriz es singular si su determinante es cero.

El rango de una matriz  $A$  se define como el tamaño de la submatriz más grande dentro de  $A$ , cuyo determinante es distinto de cero. Así por ejemplo la matriz,

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \rightarrow |A| = 0$$

Es una matriz singular y su rango es dos, | is a singular matrix, and its rank is 2,

$$\begin{vmatrix} 1 & 2 \\ 4 & 5 \end{vmatrix} = -3 \neq 0 \Rightarrow r(A) = 2$$

Para una matriz cuadrada no singular, su rango coincide con su dimensión.

En Numpy se puede obtener el rango de una matriz mediante el comando `matrix_rank`,

```
In [395]: A
Out[395]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
In [396]: np.linalg.matrix_rank(A)
Out[396]: 2
```

**Inversión.** Dada una matriz cuadrada no singular  $A$  de dimensión  $n$  existe una única matriz  $A^{-1}$  de dimensión  $n$  que cumple,

$$A \cdot A^{-1} = I_{n \times n}$$

Donde  $I_{n \times n}$  es la matriz indentidad de dimensión  $n$ . La matriz  $A^{-1}$  recibe el nombre de matriz inversa de  $A$ , y en numpy puede calcularse a partir de  $A$  como `np.linalg.inv(A)`, ó `np.linalg.matrix_power(A, -1)`. En el segundo caso, estamos empleando la función, del submódulo linalg de numpy, `matrix.power(A, n)` que permite calcular el resultado de elevar una matriz a una potencia  $A^n$ .

A singular matrix is a square matrix whose determinant is zero.

The rank of a matrix  $A$  is the size of the largest submatrix, inside  $A$ , whose determinant is non-zero. For instance,

For a non-singular square matrix, its rank meets its dimension.

In Numpy, we can obtain the rank of a matrix using the command `matrix_rank`,

**Inversion.** For a square non-singular matrix  $A$  of dimension  $n$  there is a unique matrix  $A^{-1}$  of dimension  $n$  that satisfies,

$$A \cdot A^{-1} = I_{n \times n}$$

Where  $I_{n \times n}$  is the identity matrix of dimension  $n$ . The matrix  $A^{-1}$  is called the inverse matrix of  $A$ , and we can calculate it in Numpy applying the commands `np.linalg.inv(A)`, ó `np.linalg.matrix_power(A, -1)` to the matrix  $A$ . In the second case, we are using the function `matrix.power(A, n)` which allows us to calculate the result to raise a matrix to a power  $A^n$ .

```
In [409]: B = np.linalg.inv(A)

In [410]: B
Out[410]:
array([[ 1.29166667, -0.58333333, -0.04166667],
       [ 0.58333333, -0.16666667, -0.08333333],
       [-0.48611111,  0.30555556,  0.06944444]]))

In [411]: B @ A
Out[411]:
array([[ 1.00000000e+00, -5.55111512e-17, -1.11022302e-16],
       [ 0.00000000e+00,  1.00000000e+00, -1.11022302e-16],
       [ 0.00000000e+00,  0.00000000e+00,  1.00000000e+00]]))

In [412]: np.linalg.matrix_power(A,-1)
Out[412]:
array([[ 1.29166667, -0.58333333, -0.04166667],
       [ 0.58333333, -0.16666667, -0.08333333],
       [-0.48611111,  0.30555556,  0.06944444]]))
```



La inversa de una matriz puede obtenerse a partir de la expresión,

$$A^{-1} = \frac{1}{|A|} [adj(A)]^T$$

Donde  $adj(A)$  es la matriz adjunta de  $A$ , que se obtiene sustituyendo cada elemento  $a_{ij}$  de  $A$ , por su cofactor  $C_{ij}$ . A continuación incluimos el código en Python de una función, `inver`, que calcula la inversa de una matriz. La función `inver` llama a su vez a la función `dumbdet` descrita más arriba, por lo que debemos importar el módulo `determinante` en el módulo en que vamos a crear la función `inver`.

A matrix inverse can be calculate using the following expression,

Where  $adj(A)$  is the adjugate matrix of  $A$ , which can be obtained replacing each entry  $a_{ij}$  of  $A$  by its cofactor  $C_{ij}$ . Next, We include the Python code of a function, `inver`, to calculate the inverse of a matrix. The function `inver`, in turn, calls the function `dumbdet`. Therefore, we must need to import the module `determinante` in the module where we will create the funtion `inver`

---

inversa.py

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Sat May 25 17:47:28 2024
4 Este módulo implementa la función inver, que calcula la inversa
5 de una matriz cuadrada empleando la expresión clásica
6 A^-1 = [adj(A)]^T * det(A)
7 EL determinante de la matriz se obtiene empleando la función
8 dumbdet incluida en el módulo determinante
9 ©author: abierto
10 """
11 import numpy as np
```

```

12  from determinante import dumbdet
13  def inver(A):
14      """
15          Esta función devuelve la inversa de una matriz cuadrada A
16      """
17      # primero comprobamos que la matriz suministrada
18      #es cuadrada:
19      f = A.shape[0]
20      #creamos una matriz de ceros del mismo tamaño
21      iA = np.zeros((f,f))
22      if f != A.shape[1]:
23          print('la matriz no es cuadrada, Campeón')
24          return();
25      else:
26          #calculamos el determinante de A, si es cero hemos terminado
27          dA=dumbdet(A)
28          if dA==0:
29              print('la matriz es singular, la pobre')
30              return()
31
32          if abs(dA) <= 10*np.finfo(float).eps:
33              print('cuidado determinante proximo a cero')
34
35
36          #Calculamos el cofactor de cada
37          #término de A mediante un doble bucle.
38          for i in range(f):
39              for j in range(f):
40                  # Construimos el menor correspondiente al elemento (i,j)
41                  m=np.delete(np.delete(A,i,0),j,1)
42
43                  #calculamos el cofactor llamando a la función determinante
44                  #lo ponemos ya en la posición que corresponderia a la matriz
45                  # transpuesta de la adjunta.
46                  iA[j,i]=(-1)**(i+j)*dumbdet(m)
47
48
49          #Terminamos la operacion dividiendo por el determinante de A
50          iA=iA/dA
51          return(iA)

```

---



Algunas propiedades relacionadas con la inversión de matrices son,

1. Inversa del producto de dos matrices,

$$(A \cdot B)^{-1} = B^{-1} \cdot A^{-1}$$

Some properties relate with matrix inversion are,

1. Inverse of the the producto of two ma-

2. Determinante de la inversa,

$$|A^{-1}| = |A|^{-1}$$

3. Una matriz es ortogonal si su inversa coincide con su transpuesta,

$$A^{-1} = A^T$$

trices,

$$(A \cdot B)^{-1} = B^{-1} \cdot A^{-1}$$

2. Determinant of the inverse matrix,

$$|A^{-1}| = |A|^{-1}$$

3. A matrix is orthogonal if its transpose is equal to its inverse,

$$A^{-1} = A^T$$

Tabla 3.1: Algunas funciones matemáticas en Numpy de uso frecuente  
Table 3.1: frequently used mathematical functions included in Numpy

Tipo Type	Nombre Name	variables variables	función matemática funcion matemática
Trigonométricas Trigonometric	cos	y=cos(x)	coseno de un ángulo en radianes Cosine of angle in radians
Trigonometricas Trigonometric	sin	y=sin(x)	seno de un ángulo en radianes Sine of an angle in radians
Trigonométricas Trigonometric	tan	y=tan(x)	tangente de un ángulo en radianes Tangent of an angle in radians
Trigonométricas Trigonometric	...	y=arc... (x)	inversa de una función trigonométrica
	arcsin	y=arcsin(x)	Inverse of a trigonometric function
Exponencial Exponential	exp	y=exp(x)	$e^x$
Exponencial Exponential	log	y=log(x)	logaritmo natural Natural logarithm
Exponencial Exponential	log10	log10(x)	logaritmo en base 10 Basis 10 logarithm
Exponencial Exponential	sqrt	y=sqrt(x)	$\sqrt{x}$
Redondeo Rounding	ceil	y=ceil(x)	redondeo hacia $+\infty$ rounding towards $+\infty$
Redondeo Rounding	floor	y=floor(x)	redondeo hacia $-\infty$ rounding towards $-\infty$
Redondeo Rounding	rint	y=rint(x)	redondeo al entero más próximo rounding towards the nearest integer
Redondeo Rounding	fix	y=fix(x)	redondeo hacia 0 rounding towards 0
Redondeo Rounding	mod	r=mod(x,y)	resto de la división entera de y entre x remainder after integer division
Módulos Norms	norm	y=norm(x)	módulo de un vector x Norm of a vector
Módulos Norms	abs	y=abs(x)	valor absoluto de x Absolute value of x
Módulos Norms	sign	y=sign(x)	función signo; 1 si x > 0, -1 si x < 0, 0 si x=0 sign function; 1 if x > 0, -1 if x < 0, 0 if x=0

### 3.2.1. Funciones incluidas en Numpy.

Numpy incluye un gran número de funciones. Muchas de ellas están pensadas para ser aplicadas a arrays o a variables numéricas indistintamente. En el primer caso, la función se aplica elemento a elemento y el resultado es un array con las mismas dimensiones que el original.

En la tabla 3.1, se incluyen algunos ejemplos de las funciones matemáticas más corrientes. No están todas. Para obtener una visión más completa de las funciones disponibles se aconseja acudir a la documentación de Numpy. El uso de estas funciones es directo, por ejemplo, el siguiente código calcula la exponencial de los elementos de una matriz,

```
In [1]: import numpy as np

In [2]: A = np.array([[1,2],[2,0],[3,5]])

In [3]: A
Out[3]:
array([[1, 2],
       [2, 0],
       [3, 5]])

In [4]: np.exp(A)
Out[4]:
array([[ 2.71828183,   7.3890561 ],
       [ 7.3890561 ,    1.        ],
       [20.08553692, 148.4131591 ]])
```



## 3.3. Operadores vectoriales

En esta sección vamos a estudiar el efecto de las operaciones matriciales, descritas en la sección anterior, sobre los vectores. Empezaremos por considerar el producto por un escalar  $\alpha \cdot a$ . El efecto fundamental es modificar el módulo del vector,

$$\alpha \cdot \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} \alpha a_1 \\ \alpha a_2 \\ \alpha a_3 \end{pmatrix} \rightarrow \|\alpha \cdot a\| = \sqrt{\alpha^2 a_1^2 + \alpha^2 a_2^2 + \alpha^2 a_3^2} = |\alpha| \sqrt{a_1^2 + a_2^2 + a_3^2} = |\alpha| \cdot \|a\|$$

### 3.2.1. Functions included in Numpy

Numpy includes a large number of mathematical functions. Many are intended to be applied to arrays and simple variables. In the first case, the function is applied entry-wise, and the result is an array with the same dimension as the original one.

Table 3.1 shows some examples of common mathematical functions. Not every function available in Numpy has been included. To achieve a complete view of available functions, it is advisable to read Numpy documentation. These functions can be used straightforwardly. For instance, the following code calculates the exponential values of a matrix entry,

## 3.3. Vectorial operators

In this section we will study the effect of matrix operation described in the previous section when they are applied to vectors. We start considering the effect if the product of a vector by a scalar  $\alpha \cdot a$ . The main effect is the change of the vector module,

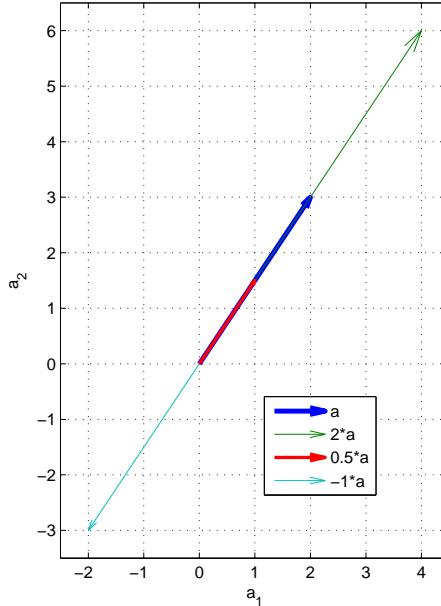


Figura 3.4: efecto del producto de un escalar por un vector

Figure 3.4: effect of the product of a vector by a scalar

Gráficamente, si  $\alpha$  es un número positivo y mayor que la unidad, el resultado del producto será un vector más largo que  $a$  con la misma dirección y sentido. Si por el contrario,  $\alpha$  es menor que la unidad, el vector resultante será más corto que  $a$ . Por último si se trata de un número negativo, a los resultados anteriores se añade el cambio de sentido con respecto a  $a$ . La figura 3.4 muestra gráficamente un ejemplo del producto de un vector por un escalar.

**Combinación lineal.** Combinando la suma de vectores, con el producto por un escalar, podemos generar nuevos vectores, a partir de otros, el proceso se conoce como combinación lineal,

$$c = \alpha \cdot a + \beta \cdot b + \dots + \theta z$$

Así el vector  $c$  sería el resultado de una combinación lineal de los vectores  $a, b, \dots, z$ . Dado un conjunto de vectores, se dice que son linealmente independientes entre sí, si no es posible poner a unos como combinación lineal

If  $\alpha$  is a positive number and greater than one, we observe graphically that the result of the product will be a vector larger than  $a$  and with the same direction and sense. Conversely, if  $\alpha$  is positive but less than one, the resulting vector will be shorter than  $a$ . Lastly if  $\alpha$  is negative, a change of sense with respect to  $a$  is added to the previously obtained results. Figure 3.4 shows graphically and example of the product of a vector by a scalar.

**Linear combination** Combining vector addition with product by a scalar, we can use a set of vectors to generate new vectors. This process is called linear combination.

So, the vector  $c$  is the result of linear combination of vectors  $a, b, \dots, z$ . A set of vectors are linearly independent if it is not possible to represent any of them as a combination of the others.

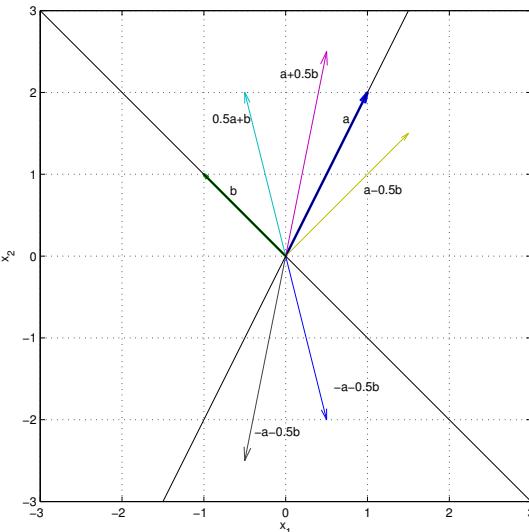


Figura 3.5: Representación gráfica de los vectores  $a = (1, 2)$ ,  $b = (-1, 1)$  y algunos vectores, combinación lineal de  $a$  y  $b$ .

Figure 3.5: A graphic representations of Vectors  $a = (1, 2)$ ,  $b = (-1, 1)$  and some vectors obtained from a lineal combination of  $a$  and  $b$ .

de otros,

$$\alpha \cdot a + \beta \cdot b + \dots + \theta z = 0 \Rightarrow \alpha = \beta = \dots = \theta = 0$$

Es posible expresar cualquier vector de dimensión  $n$  como una combinación lineal de  $n$  vectores linealmente independientes.

Supongamos  $n = 2$ , cualquier par de vectores que no estén alineados, pueden generar todos los vectores de dimensión 2 por ejemplo,

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \alpha \begin{pmatrix} 1 \\ 2 \end{pmatrix} + \beta \begin{pmatrix} -1 \\ 1 \end{pmatrix}$$

La figura 3.5 muestra gráficamente estos dos vectores y algunos de los vectores resultantes de combinarlos linealmente.

Si tomamos como ejemplo  $n = 3$ , cualquier conjunto de vectores que no estén contenidos en el mismo plano, pueden generar cualquier otro vector de dimensión 3. Por ejemplo,

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \alpha \begin{pmatrix} 1 \\ -2 \\ 1 \end{pmatrix} + \beta \begin{pmatrix} 2 \\ 0 \\ -1 \end{pmatrix} + \gamma \begin{pmatrix} -1 \\ 1 \\ 1 \end{pmatrix}$$

It is possible to represent any vector of dimensions  $n$  as a lineal combination of  $n$  vectors linearly independent.

Let's take  $n = 2$ , any pair of no collinear vectors can generate all vectors of dimension 2, for example,

Figure 3.5 shows these two vector and some other vectors obtained by combining them linearly.

If we take now  $n = 3$ , whatever three vectors not contained in the same plane can generate any other vector of dimension three. For example,

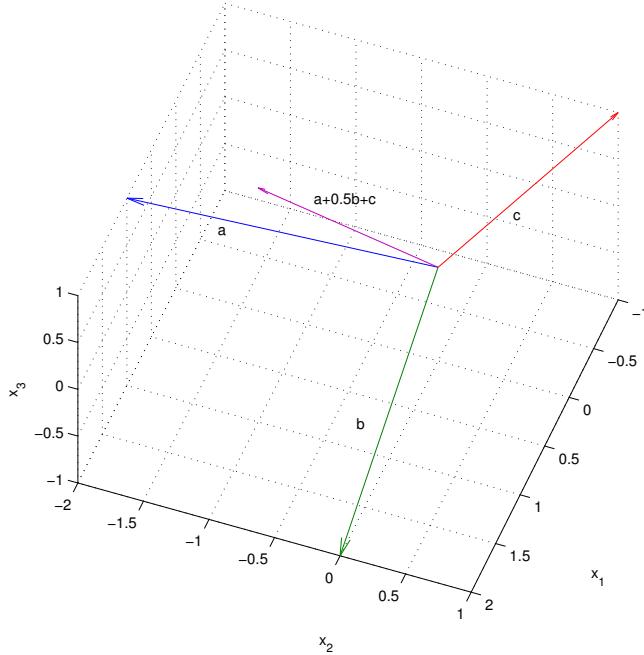


Figura 3.6: Representación gráfica de los vectores  $a = (1, -2, 1)$ ,  $b = (2, 0, -1)$ ,  $c = (-1, 1, 1)$  y del vector  $a - b + c$ .

Figure 3.6: Vectors  $a = (1, -2, 1)$ ,  $b = (2, 0, -1)$ ,  $c = (-1, 1, 1)$  and vector  $a - b + c$ ; a graphic representation

La figura 3.6 muestra gráficamente estos tres vectores y el vector resultante de su combinación lineal, con  $\alpha = 1$ ,  $\beta = -0.5$  y  $\gamma = 1$ . Es fácil ver a partir de la figura que cualquier otro vector de dimensión 3 que queramos construir puede obtenerse a partir de los vectores  $a$ ,  $b$  y  $c$ .

**Espacio vectorial y bases del espacio vectorial.** El conjunto de los vectores de dimensión  $n$ , junto con la suma vectorial y el producto por un escalar, constituye un *espacio vectorial* de dimensión  $n$ .

Como acabamos de ver, es posible obtener cualquier vector de dicho espacio vectorial a partir de  $n$  vectores linealmente independientes del mismo. Un conjunto de  $n$  vectores linealmente independientes de un espacio vectorial de dimensión  $n$  recibe el nombre de base del espacio vectorial. En principio es posible

Figure 3.6 shows these three vector and another one obtained by combining them linearly, taking  $\alpha = 1$ ,  $\beta = -0.5$  y  $\gamma = 1$ . It is easy to deduce using the figure that any other three-dimensional vector can be built using vectors  $a$ ,  $b$  y  $c$ .

**Vector spaces and vector space bases.** The set of all  $n$ -dimensional vectors with the vector addition and the product by scalars define a *vector space* of dimension  $n$ .

We have already seen that it is possible to obtain any vector of this  $n$ -dimensional space using  $n$  linearly independent vectors belonging to the space. A set of  $n$  linearly independent vectors in a vector space of dimension  $n$ , is called a basis of the vector space. We can, in principle, find infinity different basis for an  $n$ -dimensional vector space. Some of these bases

encontrar infinitas bases distintas para un espacio vectorial de dimensión  $n$ . Hay algunas particularmente interesantes,

**Bases ortogonales.** Una base ortogonal es aquella en que todos sus vectores son ortogonales entre sí, es decir cumple que su producto escalar es  $b^i \cdot b^j = 0, i \neq j$ . Donde  $b^i$  representa el  $i$ -ésimo vector de la base,  $\mathcal{B} = \{b^1, b^2, \dots, b^n\}$ .

**Bases ortonormales.** Una base ortonormal, es una base ortogonal en la que, además, los vectores de la base tienen módulo 1. Es decir,  $b^i \cdot b^j = 0, i \neq j$  y  $b^i \cdot b^j = 1, i = j$ . Un caso particularmente útil de base ortonormal es la base canónica, formada por los vectores,

$$\mathcal{C} = \left\{ c^1 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, c^2 = \begin{pmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \dots, c^{n-1} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ 0 \end{pmatrix}, c^n = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix} \right\}$$

Podemos considerar las componentes de cualquier vector como los coeficientes de la combinación lineal de la base canónica que lo representa,

$$a = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_{n-1} \\ a_n \end{pmatrix} = a_1 \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} + a_2 \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} + \dots + a_{n-1} \cdot \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ 0 \end{pmatrix} + a_n \cdot \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}$$

Por extensión, podemos generalizar este resultado a cualquier otra base, es decir podemos agrupar en un vector los coeficientes de la combinación lineal de los vectores de la base que lo generan. Por ejemplo, si construimos, para los vectores de dimensión 3 la base,

$$\mathcal{B} = \left\{ \begin{pmatrix} 1 \\ 2 \\ 0 \end{pmatrix}, \begin{pmatrix} -1 \\ 0 \\ 2 \end{pmatrix}, \begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix} \right\}$$

Podemos entonces representar un vector en la base  $\mathcal{B}$  como,

$$\alpha \cdot \begin{pmatrix} 1 \\ 2 \\ 0 \end{pmatrix} + \beta \cdot \begin{pmatrix} -1 \\ 0 \\ 2 \end{pmatrix} + \gamma \cdot \begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix} \rightarrow a^{\mathcal{B}} = \begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix}$$

have properties that make them specially interesting,

**Ortogonal bases.** An orthogonal basis is built by vectors which are all orthogonal among them. That is, the scalar product is  $b^i \cdot b^j = 0, i \neq j$ . Where  $b^i$  is the  $i$  vector of the basis,  $\mathcal{B} = \{b^1, b^2, \dots, b^n\}$ .

**Ortonormal bases.** An orthogonal basis whose vectors, in addition, have norm 1 is called an orthonormal basis. That is,  $b^i \cdot b^j = 0, i \neq j$  and  $b^i \cdot b^j = 1, i = j$ . A particularly useful orthonormal basis is the canonical basis, composed by the following vectors,

We can consider the components of any vector as the coefficients of the linear combination of the canonical base vectors which represents the vector

We can also generalise and extend this result to any other basis, just grouping in a vector the coefficients of linear combination of basis vectors which generates it. For example, we can build for vector of dimension 3 the following basis,

We can then represent a vector in the basis  $\mathcal{B}$  as,

Donde estamos empleando el superíndice  $\mathcal{B}$ , para indicar que las componentes del vector  $a$  están definidas con respecto a la base  $\mathcal{B}$ .

Así por ejemplo el vector,

$$a^{\mathcal{B}} = \begin{pmatrix} 1.125 \\ 0.375 \\ 0.75 \end{pmatrix}$$

Tendría en la base canónica las componentes,

$$a^{\mathcal{B}} = \begin{pmatrix} 1.125 \\ 0.375 \\ 0.75 \end{pmatrix} \rightarrow a = 1.125 \cdot \begin{pmatrix} 1 \\ 2 \\ 0 \end{pmatrix} + 0.375 \cdot \begin{pmatrix} -1 \\ 0 \\ 2 \end{pmatrix} + 0.75 \cdot \begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1.5 \\ 1.5 \\ 1.5 \end{pmatrix}$$

La figura 3.7, muestra gráficamente la relación entre los vectores de la base canónica  $\mathcal{C}$ , los vectores de la base  $\mathcal{B}$ , y el vector  $a$ , cuyas componentes se han representado en ambas bases.

Podemos aprovechar el producto de matrices para obtener las componentes en la base canónica  $\mathcal{C}$  de un vector representado en una base cualquiera  $\mathcal{B}$ . Si agrupamos los vectores de la base  $\mathcal{B}$ , en una matriz  $B$ ,

$$\mathcal{B} = \left\{ b^1 = \begin{pmatrix} b_{11} \\ b_{21} \\ b_{31} \\ \vdots \\ b_{n1} \end{pmatrix}, b^2 = \begin{pmatrix} b_{12} \\ b_{22} \\ b_{32} \\ \vdots \\ b_{n2} \end{pmatrix}, \dots, b^n = \begin{pmatrix} b_{1n} \\ b_{2n} \\ \vdots \\ b_{(n-1)n} \\ b_{nn} \end{pmatrix} \right\} \rightarrow B = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ b_{31} & b_{32} & \cdots & \vdots \\ \vdots & \vdots & \cdots & b_{(n-1)n} \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}$$

Supongamos que tenemos un vector  $a$  cuyas componentes en la base  $\mathcal{B}$  son,

$$a^{\mathcal{B}} = \begin{pmatrix} a_1^{\mathcal{B}} \\ a_2^{\mathcal{B}} \\ \vdots \\ a_n^{\mathcal{B}} \end{pmatrix}$$

Para obtener las componentes en la base canónica, basta entonces multiplicar la matriz  $B$ , por el vector  $a^{\mathcal{B}}$ . Así en el ejemplo que acabamos de ver,

$$a = B \cdot a^{\mathcal{B}} \rightarrow a = \begin{pmatrix} 1 & -1 & 1 \\ 2 & 0 & -1 \\ 0 & 2 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1.125 \\ 0.375 \\ 0.75 \end{pmatrix} = \begin{pmatrix} 1.5 \\ 1.5 \\ 1.5 \end{pmatrix}$$

Where the superscript  $\mathbb{B}$ , means that vector  $a$  components are defined in basis  $\mathcal{B}$ .

So, for instance, vector,

Would have the following component in the canonical basis,

Figure 3.7, shows the relationship among the canonical basis vectors  $\mathcal{C}$ , the basis  $\mathcal{B}$  vectors, and vector  $a$ , whose component have been represented in both bases.

We can use the matrix product to obtain any vector components in the canonical basis  $\mathcal{C}$  from the vector components in any other basis  $\mathcal{B}$ . If we put together the basis  $\mathcal{B}$  vector in a single matrix  $B$ ,

Suppose now that we vector  $a$  has the following components ins basis  $\mathcal{B}$ ,

we can obtain the componets on the canonical base just multiplying matrix  $B$  for vector  $a^{\mathcal{B}}$ . So in the example we just have seen,

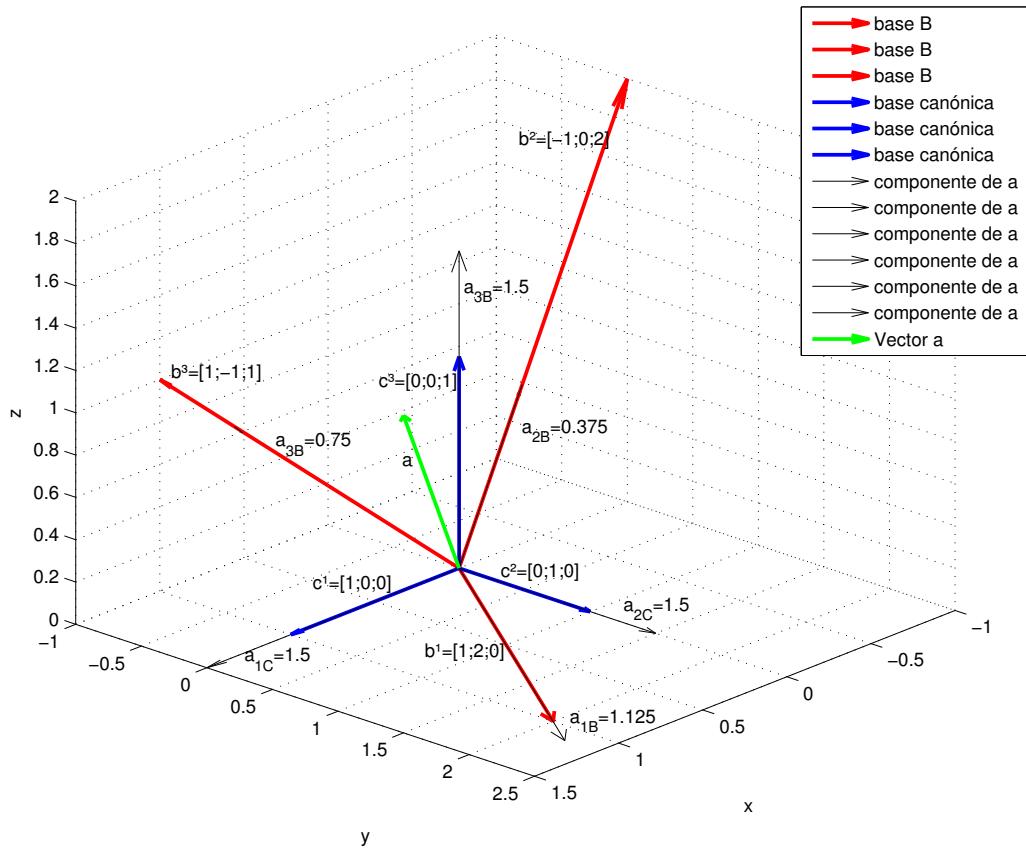


Figura 3.7: Representación gráfica del vector  $a$ , en la base canónica  $\mathcal{C}$  y en la base  $\mathcal{B}$ .  
Figure 3.7: Graphic representation of vector  $a$ , in the canonical basis  $\mathcal{C}$  and in  $\mathcal{B}$  basis.

Por último, podemos combinar el producto de matrices y la matriz inversa, para obtener las componentes de un vector en una base cualquiera a partir de sus componentes en otra base. Supongamos que tenemos dos bases  $\mathcal{B}_1$  y  $\mathcal{B}_2$  y un vector  $a$ . Podemos obtener las componentes de  $a$  en la base canónica, a partir de las componentes en la base  $\mathcal{B}_1$  como,  $a = \mathcal{B}_1 \cdot a^{\mathcal{B}_1}$  y a partir de sus componentes en la base  $\mathcal{B}_2$  como  $a = \mathcal{B}_2 \cdot a^{\mathcal{B}_2}$ . Haciendo uso de la matriz inversa,

Eventually, we can combine the matrix product and the inverse matrix to obtain the vector components in any basis whatsoever, knowing the vector components in other arbitrary basis. Suppose we have two bases  $\mathcal{B}_1$  and  $\mathcal{B}_2$  and a vector  $a$ . We can get the components of  $a$  in the canonical basis from its components on basis  $\mathcal{B}_1$  as,  $a = \mathcal{B}_1 \cdot a^{\mathcal{B}_1}$  and from basis  $\mathcal{B}_2$  as  $a = \mathcal{B}_2 \cdot a^{\mathcal{B}_2}$ . Using the inverse matrix we obtain,

$$a = \mathcal{B}_1 \cdot a^{\mathcal{B}_1} \Rightarrow a^{\mathcal{B}_1} = \mathcal{B}_1^{-1} \cdot a$$

$$a = \mathcal{B}_2 \cdot a^{\mathcal{B}_2} \Rightarrow a^{\mathcal{B}_2} = \mathcal{B}_2^{-1} \cdot a$$

,

y sustituyendo obtenemos,

and, after replace, we obtain,

$$\begin{aligned} a^{\mathcal{B}_1} &= B_1^{-1} \cdot B_2 \cdot a^{\mathcal{B}_2} \\ a^{\mathcal{B}_2} &= B_2^{-1} \cdot B_1 \cdot a^{\mathcal{B}_1} \end{aligned}$$

El siguiente código permite cambiar de base un vector  $y$ , si el vector pertenece a  $\mathbb{R}^3$ , representa gráficamente tanto el vector como las bases antigua y nueva.

The following code allows us to change a vector from one basis to another. Besides, if the vector is in  $\mathbb{R}^3$ , the program represents the vector and the old a the new bases.

---

cambio\_vb.py

---

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3 """
4 Created on Fri Jun  7 15:40:36 2024
5 Vector basis change
6 aB1 : vector a coordinates represented in base b1
7 B1 Basis in which ab1 is represented. Matrix each column represents a vector of
8 the basis
9 B2 Basis in which we want to represent vector aB1
10 aB2: vector a coordinates in base B2
11 If base B2 is omitted the program takes B2 as canonical
12 @author: juan
13 """
14 import numpy as np
15 import matplotlib.pyplot as plt
16
17 eps = np.finfo(np.float64).eps
18
19 def pintavec(v,ax,col='b'):
20     '''just to draw a 3D vector'''
21
22
23     ax.quiver(0, 0, 0, v[0], v[1], v[2],color=col,length=0.1, normalize=True)
24     plt.axis('equal')
25 def basisch(aB1,B1,B2=np.array(())):
26     '''Cambio de base de B1 a B2
27         Basis chage form B1 to B2
28     '''
29     if B2.shape == ():
30         B2 = np.eye(B1.shape[0])
31     if np.linalg.det(B1) < eps or np.linalg.det(B2) < eps:
32         print('los vectores de al menos una de las bases no son linealmente independientes')
33         return([])
34     aB2 = np.linalg.inv(B2)@B1@aB1
35     if B1.shape[0] == 3:
36         #draw the vector
37         ax = plt.figure().add_subplot(projection='3d')
38         for i in B1.T:
39             pintavec(i,ax)
40         for i in B2.T:
```

```

41     pintavec(i,ax,'r')
42     pintavec(B1@aB1,ax,'k')
43     pintavec(B2@aB2,ax,'g')
44     ax.set_xlabel('x')
45     ax.set_ylabel('y')
46     ax.set_zlabel('z')
47     return(aB2)
48
49

```

---

**Operadores lineales.** A partir de los visto en las secciones anteriores, sabemos que el producto de una matriz de  $A$  de orden  $n \times n$  multiplicada por un vector  $b$  de dimensión  $n$  da como resultado un nuevo vector  $c = A \cdot b$  de dimensión  $n$ . Podemos considerar cada matriz  $n \times n$  como un *operador lineal*, que transforma unos vectores en otros. Decimos que se trata de un operador lineal porque las componentes del vector resultante, están relacionadas linealmente con las del vector original, por ejemplo para  $n = 3$ ,

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \rightarrow \begin{aligned} y_1 &= a_{11}x_1 + a_{12}x_2 + a_{13}x_3 \\ y_2 &= a_{21}x_1 + a_{22}x_2 + a_{23}x_3 \\ y_3 &= a_{31}x_1 + a_{32}x_2 + a_{33}x_3 \end{aligned}$$

Entre los operadores lineales, es posible destacar aquellos que producen transformaciones geométricas sencillas. Veamos algunos ejemplos para vectores bidimensionales,

1. Dilatación: aumenta el módulo de un vector en un factor  $\alpha > 1$ . Contracción: disminuye el módulo de un vector en un factor  $0 < \alpha < 1$ . En ambos casos, se conserva la dirección y el sentido del vector original.

**Linear operators.** In previous sections we saw that the product of a matrix  $A$  of dimensions  $n \times n$  by a vector  $b$  of dimension  $n$  cast a new vector  $c$  of dimension  $n$  as a result. We could consider every matrix  $n \times n$  as a *linear operator*, which transforms one vector into another. We define it as a linear operator because the components of the resulting vector are linearly related with the components of the operator (the original vector before the transformation). For example, in  $n = 3$ ,

Some linear operator are particular interesting because they generate simple geometrical transformations. Let's see some examples for bi-dimensional vectors:

1. Dilation: Spans the module of a vector in a factor  $\alpha > 1$ . Contraction: diminishes the module of a vector in a factor  $0 < \alpha < 1$ . In both cases, the direction and sense of the original vector is preserved.

$$R = \begin{pmatrix} \alpha & 0 \\ 0 & \alpha \end{pmatrix} \rightarrow R \cdot a = \begin{pmatrix} \alpha & 0 \\ 0 & \alpha \end{pmatrix} \cdot \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} \alpha \cdot a_1 \\ \alpha \cdot a_2 \end{pmatrix}$$

2. Reflexión de un vector respecto al eje x, conservando su módulo,

2. Reflection of a vector with respect to the x-axis, preserving its norm,

$$R_x = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \rightarrow R_x \cdot a = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} a_1 \\ -a_2 \end{pmatrix}$$

3. Reflexión de un vector respecto al eje y, conservando su módulo,

3. reflection of a vector with respect to the y-axis, preserving its norm,

$$R_y = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \rightarrow R_y \cdot a = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} -a_1 \\ a_2 \end{pmatrix}$$

4. Reflexión respecto al origen: Invierte el sentido de un vector, conservando su módulo y dirección,

$$R = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \rightarrow R \cdot a = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} -a_1 \\ -a_2 \end{pmatrix}$$

Sería equivalente a aplicar una reflexión respecto al eje x y luego respecto al eje y o viceversa,

$$R = R_x \cdot R_y = R_y \cdot R_x.$$

5. Rotación en torno al origen un ángulo  $\theta$ ,

4. reflection with respect to the origin: Invert the sense of the vector, preserving its norm and direction.

It is equivalent to apply a refletion on x-axis and then a reflection on y-axis or vice versa,

5. Rotation an angle  $\theta$  around the origin,

$$R_\theta = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \rightarrow R_\theta \cdot a = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \cdot \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} a_1 \cos(\theta) - a_2 \sin(\theta) \\ a_1 \sin(\theta) + a_2 \cos(\theta) \end{pmatrix}$$

La figura 3.8 muestra los vectores resultantes de aplicar las transformaciones lineales que acabamos de describir al vector,  $a = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$ ,

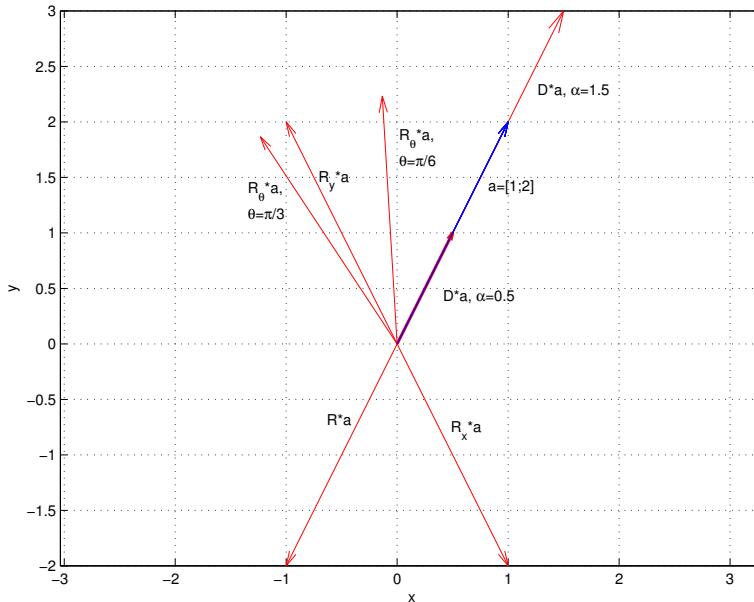


Figura 3.8: Transformaciones lineales del vector  $a = [1, 2]$ .  $D$ , dilatación/contracción en un factor 1.5/0.5.  $R_x$ , reflexión respecto al eje x.  $R_y$ , reflexión respecto al eje y.  $R_\theta$  rotaciones respecto al origen para ángulos  $\theta = \pi/6$  y  $\theta = \pi/3$

Figure 3.8: Linear transformation of vector  $= [1, 1]$ .  $D$ , factor 1.5/0.5 dilation/contraction.  $R_y$  reflection on y-axis.  $R_\theta$   $\theta = \pi/6$  and  $\theta = \pi/3$  angles rotations around the origin.

**Norma de una matriz.** La norma de una matriz se puede definir a partir del efecto que produce al actuar, como un operador lineal, sobre un vector. En este caso, se les llama normas *inducidas*. Para una matriz  $A$  de orden  $m \times n$ ,  $y_{(m)} = A_{(m \times n)}x_{(n)}$ . La norma inducida de  $A$  se define en función de las normas de los vectores  $x$  de su dominio y de las normas de los vectores  $y$  de su rango como,

$$\|A\| = \max_{x \neq 0} \frac{\|y\|}{\|x\|} = \max_{x \neq 0} \frac{\|Ax\|}{\|x\|}$$

Se puede interpretar como el factor máximo con que el que la matriz  $A$  puede *alargar* un vector cualquiera. Es posible definir la norma inducida en función de los vectores unitarios del dominio,

$$\|A\| = \max_{x \neq 0} \frac{\|Ax\|}{\|x\|} = \max_{x \neq 0} \left\| A \frac{x}{\|x\|} \right\| = \max_{\|x\|=1} \|Ax\|$$

Junto a la norma inducida que acabamos de ver, se definen las siguientes normas,

1. Norma 1: Se suman los elementos de cada columna de la matriz, y se toma como norma el valor máximo de dichas sumas,

$$\|A_{m,n}\|_1 = \max_j \sum_{i=1}^m a_{ij}$$

2. Norma  $\infty$ : Se suman los elementos de cada fila y se toma como norma  $\infty$  el valor máximo de dichas sumas.

$$\|A_{m,n}\|_\infty = \max_i \sum_{j=1}^m a_{ij}$$

3. Norma 2: Se define como el mayor de los valores singulares de una matriz. (Ver sección 3.5.5).

$$\|A_{m,n}\|_2 = \sigma_1$$

4. Norma de Frobenius. Se define como la raíz cuadrada de la suma de los cuadrados de todos los elementos de la matriz,

$$\|A_{m,n}\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^m a_{ij}^2}$$

**Matrix norm.** A matrix norm can be defined from the effect of the matrix when operates on a vector as a linear operator. In this case, we call these norms *induced* norms. For a matrix  $A$  of dimensions  $m \times n$ ,  $y(m) = A_{m \times n}x(n)$ . The induced norm of  $A$  is defined using the norms of the vectors  $x$  of its domain and the norms of the vectors  $y$  of its range, according to the following expression,

It can be interpreted as the maximum factor a matrix  $A$  can *enlarge* any vector. Sometimes the induced norm is defined using the unitary vector of its domain,

Besides the induced norm already described, it is possible to define the following norms,

1. Norm 1: we add the entries of each column up and then take the maximum value of these sums as a norm

2. Norm  $\infty$ : We add the entries of each row up and then take the maximum value of these sums as the  $\infty$  norm.

3. Norm 2:it si defined as the largest of the matrix singular values. (See section 3.5.5).

4.Frobenius' norm. It is defined as the square root of the sum of the squared values of the matrix entries,

Que también puede expresarse de forma mas directa como,

$$\|A_{m,n}\|_F = \sqrt{\text{tr}(A^T \cdot A)}$$

En Numpy, es posible calcular las distintas normas de una matriz, de modo análogo a como se calculan para el caso de vectores, mediante el comando `norm(A, p)`. Donde `A`, es ahora un a matriz y `p` especifica el tipo de norma que se quiere calcular. En el caso de una matriz, el parámetro `p` solo puede tomar los valores, 1 (norma 1), 2 (norma 2), `inf` (norma  $\infty$ ), y '`fro`' (norma de frobenius). Esta última es la norma por defecto, y es la que se obtiene si se no se define el tipo de norma. El siguiente ejemplo muestra el cálculo de las normas 1, 2,  $\infty$  y de Frobenius de la misma matriz.

That can be also expressed in a more straightforward way as,

In Numpy is possible to calculate the different matrix norms, similarly to how we calculate the norm of a vector, using the command `norm(A, p)`. Where `A` is now a matrix and `p` specifies the kind of norm we can calculate. In the case of a matrix, parameter `p` can take only the values, 1 (norm 1), 2 (norm 2), `inf`, (norm  $\infty$ ), and '`fro`' (frobenius' norm). This last is the norm by default and, thus, if the norm Numpy calculates if parameter `p` is omitted. The following example shows the calculations of norms 1, 2,  $\infty$ , and Frobenius for the same matrix.

```
In [11]: A = np.array([[1,-1,3],[2,0,-2],[3,1,2]])

In [12]: A
Out[12]:
array([[ 1, -1,  3],
       [ 2,  0, -2],
       [ 3,  1,  2]])

In [13]: np.linalg.norm(A)
Out[13]: 5.744562646538029

In [14]: np.linalg.norm(A, 'fro')
Out[14]: 5.744562646538029

In [15]: np.linalg.norm(A,1)
Out[15]: 7.0

In [17]: np.linalg.norm(A,inf)
Out[17]: 6.0

In [18]: np.linalg.norm(A,2)
Out[18]: 4.552861506620628
```

**Formas cuadráticas.** Se define como forma cuadrática a la siguiente operación entre una matriz cuadrada  $A$  de orden  $n \times n$  y un vector  $x$  de dimensión  $n$ ,

**Quadratic forms.** We define the following operation between a square matrix  $A$  of dimensions  $n \times n$  an a vector  $x$  as a quadratic form,

$$\alpha = x^T \cdot A \cdot x, \quad \alpha \in \mathbb{R}$$

El resultado es un escalar. Así por ejemplo,

$$A = \begin{pmatrix} 1 & 2 & -1 \\ 2 & 0 & 2 \\ 3 & 2 & -2 \end{pmatrix}, \quad x = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \rightarrow (1 \ 2 \ 3) \cdot \begin{pmatrix} 1 & 2 & -1 \\ 2 & 0 & 2 \\ 3 & 2 & -2 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} = 21$$

Para dimensión  $n = 2$ ,

$$\alpha = (x_1 \ x_2) \cdot \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \rightarrow x_3 \equiv \alpha = a_{11}x_1^2 + (a_{12} + a_{21})x_1x_2 + a_{22}x_2^2$$

Lo que obtenemos, dependiendo de los signos de  $a_{11}$  y  $a_{12}$ , es la ecuación de un paraboloid o un hiperboloido. En la figura 3.9 Se muestra un ejemplo,

The result is a scalar. For instance,

$$A = \begin{pmatrix} 1 & 2 & -1 \\ 2 & 0 & 2 \\ 3 & 2 & -2 \end{pmatrix}, \quad x = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \rightarrow (1 \ 2 \ 3) \cdot \begin{pmatrix} 1 & 2 & -1 \\ 2 & 0 & 2 \\ 3 & 2 & -2 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} = 21$$

In  $n = 2$  dimension,

Depending of the sign of  $a_{11}$  y  $a_{12}$  we obtain a paraboloid or a hyperboloid equation. Figure 3.9 shows an example.

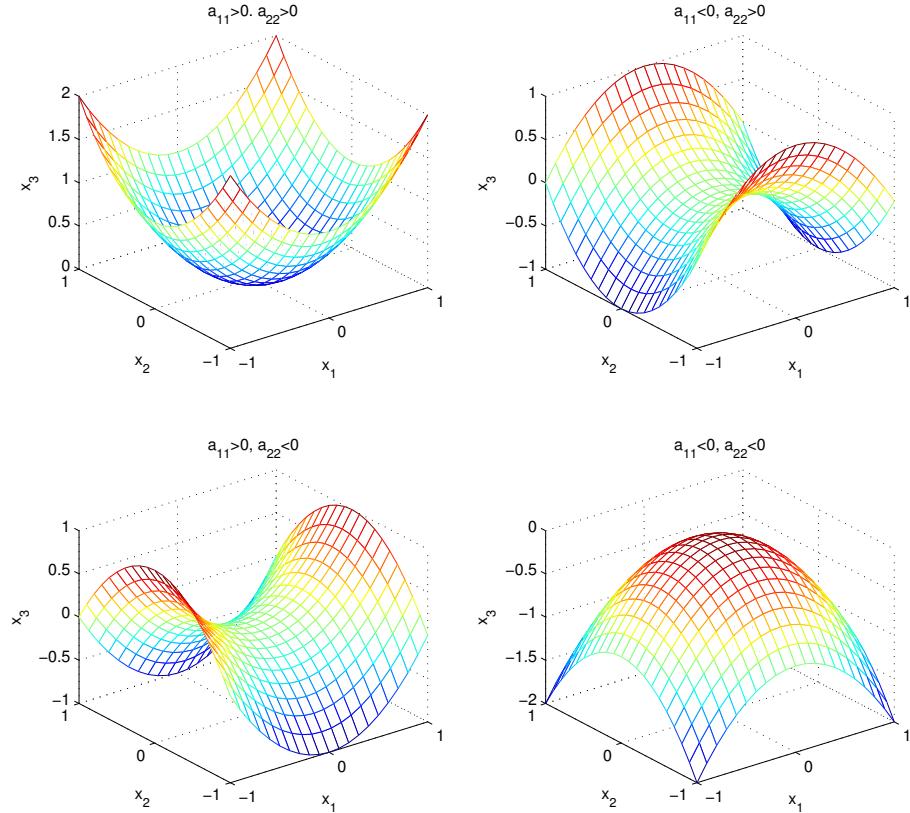


Figura 3.9: Formas cuadráticas asociadas a las cuatro matrices diagonales:  $|a_{11}| = |a_{22}| = 1$ ,  $a_{12} = a_{21} = 0$

Figure 3.9: Quadratic forms obtained using the four diagonal matrices:  $|a_{11}| = |a_{22}| = 1$ ,  $a_{12} = a_{21} = 0$

Veamos brevemente, algunas propiedades de las formas cuadráticas,

1. Una matriz  $A$  de orden  $n \times n$  se dice que es definida positiva si da lugar a una forma cuadrática que es siempre mayor que cero para cualquier vector no nulo,

$$x^T \cdot A \cdot x > 0, \forall x \neq 0$$

2. Una matriz *simétrica* es definida positiva si todos sus *valores propios* (ver sección 3.5.3) son positivos.

3. Una matriz no simétrica  $A$  es definida positiva si su parte simétrica  $A_s = (A + A^T)/2$  lo es.

$$x \cdot A_s \cdot x > 0, \forall x \neq 0 \Rightarrow x \cdot A \cdot x > 0, \forall x \neq 0$$

Briefly we will revise some quadratic forms properties,

1. A matrix  $A$  of dimensions  $n \times n$  is positive definite if it generates a quadratic form always greater than zero for any non-null vector.

2. A *symmetric* matrix is positive definite if all its *eigenvalues* (see section 3.5.3) are positive.

3. A non-symmetric matrix  $A$  is positive definite if its symmetric part  $A_s = (A + A^T)/2$  is so.



### 3.4. Tipos de matrices empleados frecuentemente.

Definimos a continuación algunos tipos de matrices frecuentemente empleados en álgebra, algunos ya han sido introducidos en secciones anteriores; los reunimos todos aquí para facilitar su consulta.

1 Matriz ortogonal: Una matriz  $A_{n \times n}$  es ortogonal cuando su inversa coincide con su traspuesta.

$$A^T = A^{-1}$$

ejemplo,

$$A = \begin{pmatrix} 1/3 & 2/3 & 2/3 \\ 2/3 & -2/3 & 1/3 \\ 2/3 & 1/3 & -2/3 \end{pmatrix} \rightarrow A \cdot A^T = A^T \cdot A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

2. Matriz simétrica: Una matriz  $A_{n \times n}$  es simétrica cuando es igual que su traspuesta,

$$A = A^T \rightarrow a_{ij} = a_{ji}$$

ejemplo,

$$A = \begin{pmatrix} 1 & -2 & 3 \\ -2 & 4 & 0 \\ 3 & 0 & -5 \end{pmatrix}$$

### 3.4. Kinds of matrices frequently used.

We define next some kinds of matrices frequently used in algebra, some of them have been already introduced in previous sections; we put them together here for ease reference.

1 Orthogonal matrix: A matrix  $A_{n \times n}$  is orthogonal whenever its inverse matrix is equal to its transpose.

2. Symmetric Matrix: A matrix  $n \times n$  is symmetric if it is equal to its transpose.

3. Matriz Diagonal: Una matriz  $A$  es diagonal si solo son distintos de ceros los elementos de su diagonal principal,

$$\begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix} \rightarrow a_{ij} = 0, \forall i \neq j$$

4. Matriz triangular superior: Una matriz cuadrada es triangular superior cuando todos los elementos situados por debajo de la diagonal son cero. Es estrictamente diagonal superior si además los elementos de la diagonal también son cero,

3. Diagonal matrix: A matrix  $A$  is diagonal if only the entries of its main diagonal are different from zero.

4. Upper triangular matrix: a square matrix is upper triangular when every entry below its main diagonal is zero, and if it is strictly upper triangular in all the diagonal entries are also zero.

$$TRS \rightarrow a_{ij} = 0, \forall i \geq j$$

$$ETRS \rightarrow a_{ij} = 0, \forall i > j$$

ejemplos,

$$TRS = \begin{pmatrix} 1 & 3 & 7 \\ 0 & 2 & -1 \\ 0 & 0 & 4 \end{pmatrix}$$

$$ETRS = \begin{pmatrix} 0 & 3 & 7 \\ 0 & 0 & -1 \\ 0 & 0 & 0 \end{pmatrix}$$

5. Matriz triangular inferior: Una matriz es triangular inferior si todos los elementos pro encima de su diagonal son cero. Es estrictamente triangular inferior si además los elementos de su diagonal son también cero,

5. Lower triangular matrix: A square matrix is lower triangular when every entry above its main diagonal are zero, and it is strictly lower triangular if all the diagonal entries are also zero.

$$TRI \rightarrow a_{ij} = 0, \forall i \leq j$$

$$ETRI \rightarrow a_{ij} = 0, \forall i < j$$

ejemplos,

$$TRI = \begin{pmatrix} 1 & 0 & 0 \\ 3 & 2 & 0 \\ 7 & -1 & 4 \end{pmatrix}$$

$$ETRI = \begin{pmatrix} 0 & 0 & 0 \\ 3 & 0 & 0 \\ 7 & -1 & 0 \end{pmatrix}$$

6. Matriz definida Positiva. Una matriz  $A_{n \times n}$  es definida positiva si dado un vector  $x$  no nulo cumple,

si,

$$x^T \cdot A \cdot x > 0, \quad \forall x \neq 0,$$

6. Positive definite matrix: A matrix  $A_{n \times n}$  is positive definite if taking a non-null vector  $x$  satisfy,

if,

$$x^T \cdot A \cdot x \geq 0, \quad \forall x \neq 0,$$

entonces la matriz  $A$  es semidefinida positiva.

7. Una matriz es (strictamente) diagonal dominante si cada uno de los elementos de la diagonal en valor absoluto es ()mayor) mayor o igual que la suma de los valores absolutos de los elementos de la fila a la que pertenece.

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|, \quad \forall i$$

ejemplo,

$$A = \begin{pmatrix} 10 & 2 & 3 \\ 2 & -5 & 1 \\ 4 & -2 & 8 \end{pmatrix} \rightarrow \begin{cases} 10 > 2 + 3 \\ 5 > 2 + 1 \\ 8 > 4 + 2 \end{cases}$$

then matrix  $A$  is positive semi-definite.

7. A matrix is (strictly) diagonally dominant if, for every row of the matrix, the magnitude of the diagonal entry in a row is (greater) greater than or equal to the sum of the magnitudes of all the other (off-diagonal) entries in that row.

### 3.5. Factorización de matrices

La factorización de matrices, consiste en la descomposición de una matriz en el producto de dos o más matrices. Las matrices resultantes de la factorización se eligen de modo que simplifiquen, o hagan más robustas numéricamente determinadas operaciones matriciales: Cálculos de determinantes, inversas, etc. A continuación se describen las más comunes.

#### 3.5.1. Factorization LU

Consiste en factorizar una matriz como el producto de una matriz triangular inferior  $L$  por una matriz triangular superior  $U$ ,  $A = L \cdot U$ . Por ejemplo,

$$\begin{pmatrix} 3 & 4 & 2 \\ 2 & 0 & 1 \\ 3 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 2/3 & 1 & 0 \\ 1 & 3/4 & 1 \end{pmatrix} \cdot \begin{pmatrix} 3 & 4 & 2 \\ 0 & -8/3 & -1/3 \\ 0 & 0 & -3/4 \end{pmatrix}$$

### 3.5. Matrix factorization.

Factorizing a matrix consists on divide it into the product of two or more matrices. We choose the resulting factor matrices so that they simplify or make certain matrix operations more robust. For instance: matrix determinant calculation, obtaining inverse matrix , etc. Next, we will describe most common matrix factorizations.

#### 3.5.1. LU factorization.

We factorize a matrix into the product of a lower triangular matrix  $L$  by an upper triangular one  $U$ ,  $A = L \cdot U$ . For instance,

Una aplicación inmediata, es el cálculo del determinante. Puesto que el determinante de una matriz triangular, es directamente el producto de los elementos de la diagonal.

En el ejemplo anterior,

$$|A| = 6 \equiv |L| \cdot |U| = 1 \cdot 1 \cdot 1 \cdot 3 \cdot \left(-\frac{8}{3}\right) \cdot \left(-\frac{3}{4}\right) = 6$$

Describir un método para realizar la factorización LU de una matriz, queda fuera de los objetivos de estos apuntes. Sin embargo sí que utilizaremos el comando `lu` incluido en una librería de Python llamada Scipy para obtener directamente el resultado de una factorización LU. Es una librería parecida a Numpy y que, al igual que esta última contiene un gran número de funciones matemáticas. De hecho Numpy y Scipy solapan un poco y hay determinadas funciones que pueden encontrarse en ambas librerías. El comando pertenece a un submodule de Scipy llamado `linalg`. Admite como variable de entrada una matriz cuadrada  $A$  y nos devuelve como salida una matriz de permutación  $P$ , de la que hablaremos enseguida, y una matriz triangular inferior  $L$  y otra triangular superior  $U$  de modo que se cumple que  $A = PLU$ .

A veces obtener la factorización LU directa de una matriz, puede llevar a cálculos que son numéricamente inestables afectando a la precisión del resultado. Una manera de paliar este efecto es permutar entre sí algunas de las filas de la matriz que se quiere factorizar y factorizar la versión permutada.

La permutación de las filas de una matriz  $A$  de orden  $n \times m$ , se puede definir a partir del producto con las matrices de permutación de orden  $n \times n$ . Éstas se obtienen permutando directamente las filas de la matriz identidad  $I_{n \times n}$ . Si una matriz de permutación multiplica a otra matriz por la izquierda, permuta el orden de sus filas. Si la multiplica por la derecha, permuta el orden de sus columnas. Así por ejemplo, para matrices de orden  $3 \times n$ ,

$$I_{n \times n} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \rightarrow P_{1 \leftrightarrow 3} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

A straightforward application is determinant calculation. Recall that a triangular matrix determinant is the product of their diagonal entries.

So, from the previous example we obtain,

The description of method to calculate a matrix LU factorization is far beyond the scope of this notes. However, we do use the command `lu`, include in a Python library called Scipy, to compute the result of a LU factorization. Scipy is a library very much alike to Numpy and similarly to it, Scipy contains a large number of mathematical functions. In fact, Numpy and Scipy overlaps a little and you can find certain functions in both libraries. The `lu` command belongs to Scipy submodule called `linalg` and takes as input a square matrix  $A$  of dimension  $n \times n$ , and returns as outputs a permutation matrix  $P$  of dimension  $n \times n$  we will describe later on, the lower triangular matrix  $L$ , and the upper triangular matrix  $U$ . The three output matrices satisfy  $A = PLU$ .

Sometimes obtaining directly the LU factorization of a matrix, leads to numerically instable calculations. This, in turn, affect the results precision. One way to overcome this problem is to permute some rows of the matrix we intend to factorize and then, to factorize the permuted version.

We can define the rows permutation of  $A$  matrix of dimensions  $n \times m$  multiplying the matrix by permutation matrices of dimensions  $n \times n$ . We obtain these last permuting the rows of the identity matrix  $I_{n \times n}$ . When a permutation matrix multiplies another matrix by its left side, it permutes the order of the matrix rows. By contrast, if the permutation matrix multiplies another matrix by its right, it permutes the order of the matrix columns. For instance, for matrix of dimensions  $3 \times n$ ,

Si multiplicamos  $P_{1 \leftrightarrow 3}$  con cualquier otra matriz  $A$  de orden  $3 \times n$ , El resultado es equivalente a intercambiar en la matriz  $A$  la fila 1 con la 3. Por ejemplo,

$$P_{1 \leftrightarrow 3} \cdot A = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 5 & 3 \\ 4 & 2 & 3 & 0 \\ 3 & 6 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 3 & 6 & 2 & 1 \\ 4 & 2 & 3 & 0 \\ 1 & 2 & 5 & 3 \end{pmatrix} = A_{1 \leftrightarrow 3}$$

Volviendo a la factorización LU, vamos a ver como se calcularía para la siguiente matriz,

$$A = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 2 & 0 & 1 & -2 \\ 3 & 2 & 1 & 8 \\ 5 & 2 & 3 & 2 \end{pmatrix}$$

Scipy, calcula siempre la factorización LU, permutando las filas para obtener un resultado lo mas preciso posible,

If we multiply  $P_{1 \leftrightarrow 3}$  by any other matrix  $A$  of dimensions  $3 \times n$ , the result is equivalent to interchange row 1 and 3. For example,

Coming back to LU factorization, we are going to calculate it for the following matrix,

Scipy always use row permutation, when calculating an LU factorization, to obtain a result as accurate as possible,

```
In [213]: A = np.array([[3,4,2,5],[2,0,1,-2],[3,2,1,8],[5,2,3,2]])
```

```
In [214]: A
```

```
Out[214]:
```

```
array([[ 3,  4,  2,  5],
       [ 2,  0,  1, -2],
       [ 3,  2,  1,  8],
       [ 5,  2,  3,  2]])
```

```
In [215]: import scipy as sp
```

```
In [216]: [P,L,U] = sp.linalg.lu(A)
```

```
In [217]: P
```

```
Out[217]:
```

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

```
In [218]: L
```

```
Out[218]:
```

```
array([[ 1.        ,  0.        ,  0.        ,  0.        ,  0.        ],
       [ 0.6       ,  1.        ,  0.        ,  0.        ,  0.        ],
       [ 0.6       ,  0.28571429,  1.        ,  0.        ,  0.        ],
       [ 0.4       , -0.28571429,  0.16666667,  1.        ]])
```

```
In [219]: U
Out[219]:
array([[ 5.        ,  2.        ,  3.        ,  2.        ],
       [ 0.        ,  2.8      ,  0.2      ,  3.8      ],
       [ 0.        ,  0.        , -0.85714286,  5.71428571],
       [ 0.        ,  0.        ,  0.        , -2.66666667]]))

In [220]: P @ L @ U
Out[220]:
array([[ 3.,  4.,  2.,  5.],
       [ 2.,  0.,  1., -2.],
       [ 3.,  2.,  1.,  8.],
       [ 5.,  2.,  3.,  2.]])
```

### 3.5.2. Factorización de Cholesky

Dada una matriz cuadrada, simétrica y definida positiva, es siempre posible factorizarla como  $A = L \cdot L^T$ . Donde  $L$  es una matriz triangular inferior,

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{12} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{nn} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 & \cdots & 0 \\ L_{21} & L_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ L_{n1} & L_{n2} & \cdots & L_{nn} \end{pmatrix} \cdot \begin{pmatrix} L_{11} & L_{21} & \cdots & L_{n1} \\ 0 & L_{22} & \cdots & L_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & L_{nn} \end{pmatrix}$$

Numpy tiene una función, en el submódulo `linalg`, llamada `cholesky`, que permite obtener la factorización de Cholesky.

### 3.5.2. Cholesky factorisation

Given a square simetric positive definite matrix, it always possible to factorize it as  $A = L \cdot L^T$ . Where  $L$  is a lower triangular matrix.

Numpy has a function, included in the `linalg` submodule, called `cholesky`, to calculate the `cholesky` factorization of a matrix.

```
In [224]: B
Out[224]:
array([[47, 28, 26, 45],
       [28, 24, 16, 40],
       [26, 16, 15, 22],
       [45, 40, 22, 97]])
```

```
In [225]: L = np.linalg.cholesky(B)
```

```
In [226]: L
Out[226]:
array([[ 6.8556546 ,  0.        ,  0.        ,  0.        ],
       [ 4.08421976,  2.70539257,  0.        ,  0.        ],
       [ 3.79248978,  0.18874832,  0.76249285,  0.        ],
       [ 6.56392462,  4.87599823, -5.00195311,  2.2627417 ]])
```

```
In [227]: L @ L.T
Out[227]:
array([[47., 28., 26., 45.],
       [28., 24., 16., 40.],
       [26., 16., 15., 22.],
       [45., 40., 22., 97.]])
```

Si la matriz no es definida positiva, la función `cholesky` da un mensaje de error.

### 3.5.3. Diagonalización

**Autovectores y autovalores.** Dada una matriz  $A$  de orden  $n \times n$ , se define como autovector, o vector propio de dicha matriz al vector  $x$  que cumple,

$$A \cdot x = \lambda \cdot x, \quad x \neq 0, \quad \lambda \in \mathbb{C}$$

Es decir, el efecto de multiplicar la matriz  $A$  por el vector  $x$  es equivalente a multiplicar el vector  $x$  por un número  $\lambda$ . Tanto  $X$  como  $\lambda$  pueden ser reales o complejos.

$\lambda$  recibe el nombre de autovalor, o valor propio de la matriz  $A$  asociado al autovector  $x$ . Así por ejemplo,

$$A = \begin{pmatrix} -2 & 0 & 0 \\ 0 & 2 & -1 \\ 0 & 0 & 3 \end{pmatrix}$$

tiene un autovalor  $\lambda = 3$  para el vector propio  $x = [0, -3, 3]^T$ ,

$$\begin{pmatrix} -2 & 0 & 0 \\ 0 & 2 & -1 \\ 0 & 0 & 3 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ -3 \\ 3 \end{pmatrix} = 3 \cdot \begin{pmatrix} 0 \\ -3 \\ 3 \end{pmatrix} = \begin{pmatrix} 0 \\ -9 \\ 9 \end{pmatrix}$$

El vector propio asociado a un valor propio no es único, si  $x$  es un vector propio de una matriz  $A$ , asociado a un valor propio  $\lambda$ , Es trivial comprobar que cualquier vector de la forma  $\alpha \cdot x$ ,  $\alpha \in \mathbb{C}$  también es un vector propio asociado a  $\lambda$ ,

$$A \cdot x = \lambda x \Rightarrow A \cdot \alpha x = \lambda \alpha x \tag{3.1}$$

El conjunto de todos los autovalores de una matriz  $A$  recibe el nombre de espectro de  $A$  y se representa como  $\Lambda(A)$ . Una descomposición en autovalores de una matriz  $A$  se define como,

$$A = X \cdot D \cdot X^{-1}$$

If the matrix is not definite positive, then function `cholesky` returns an error message.

### 3.5.3. Diagonalisation

**Eigenvectors and eigenvalues.** We define an eigenvector or characteristic vector  $x$  of the  $n \times n$  matrix  $A$  as,

Namely, the result of multiply matrix  $A$  for its eigenvector  $x$  is equal to multiply the number,  $\lambda$  for the eigen vector.  $x$  and  $\lambda$  could be real or complex.

$\lambda$  is call de eigenvalue or characteristic values of the matrix  $A$ , associated to the eigenvector  $x$ . For example,<sup>4</sup>

It has an eigenvalue  $\lambda = 3$  for the eigenvector  $x = [0, -3, 3]^T$ ,

There is not a single eigenvector associated to an eigenvalue, if  $x$  is an eigenvector of a matrix  $A$ , associated to an eigenvalue  $\lambda$ , It is easy to check that any other vector that we build as  $\alpha \cdot x$ ,  $\alpha \in \mathbb{C}$  is also an eigenvector associated to  $\lambda$ ,

we define the set of all eigenvalues of a matrix  $A$  as the spectrum of  $A$  and we represented it as  $\Lambda(A)$ . The eigendecomposition of a matrix  $A$  is defined as,

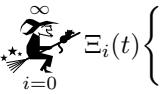
Donde  $D$  es una matriz diagonal formada por los autovalores de  $A$ .

$$\begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \cdots & 0 \\ 0 & 0 & \cdots & \lambda_n \end{pmatrix}$$

Esta descomposición no siempre existe.

Where  $D$  is a diagonal matrix built using the eigenvalues of  $A$ .

The eigendecomposition does not always exist.



$$\Xi_i(t) \left\{ \begin{array}{l} \\ \\ \\ \end{array} \right.$$

En general, si dos matrices  $A$  y  $B$  cumplen,

In general, if two matrices  $A$  and  $B$  satisfy,

$$A = X \cdot B \cdot X^{-1}$$

se dice de ellas que son similares. A la transformación que lleva a convertir  $B$  en  $A$  se le llama una transformación de semejanza. Una matriz es diagonalizable cuando admite una transformación de semejanza con una matriz diagonal de autovalores.

Volviendo al ejemplo anterior podemos factorizar la matriz  $A$  como,

$$A = \begin{pmatrix} -2 & 0 & 0 \\ 0 & 2 & -1 \\ 0 & 0 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & -3 \\ 0 & 0 & 3 \end{pmatrix} \cdot \begin{pmatrix} -2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & -1 \\ 0 & 0 & 1/3 \end{pmatrix}$$

Por tanto en este ejemplo, los autovalores de  $A$  serían  $\lambda_1 = -2$ ,  $\lambda_2 = 2$  y  $\lambda_3 = 3$ . Para estudiar la composición de la matriz  $X$  Reescribimos la expresión de la relación de semejanza de la siguiente manera,

Their a similar matrices. The transformation that convert  $A$  into  $B$  is denoted a similarity transformation. A matrix is diagonalisable when there is a similarity transformation that converts it in a diagonal matrix formed by its eigenvalues.

Coming back to the previous example we can factorise matrix  $A$  as,

So, in this example the  $A$  eigenvectors would be  $\lambda_1 = -2$ ,  $\lambda_2 = 2$  and  $\lambda_3 = 3$ . To study the structure of matrix  $X$ , we rewrite the similarity relationship as follows,

$$A = X \cdot D \cdot X^{-1} \rightarrow A \cdot X = X \cdot D$$

Si consideramos ahora cada columna de la matriz  $X$  como un vector,

Now, we can consider each column of matrix  $X$  as a vector,

$$A \cdot X = X \cdot D \rightarrow A \cdot (x_1 | x_2 | \cdots | x_n) = (x_1 | x_2 | \cdots | x_n) \cdot \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \cdots & 0 \\ 0 & 0 & \cdots & \lambda_n \end{pmatrix}$$

Es fácil comprobar que si consideramos el producto de la matriz  $A$ , por cada una de las columnas de la matriz  $X$  se cumple,

It is easy to check that the product of matrix  $A$  by each column of matrix  $X$  satisfy,

$$A \cdot x_1 = \lambda_1 \cdot x_1$$

$$A \cdot x_2 = \lambda_2 \cdot x_2$$

⋮

$$A \cdot x_n = \lambda_n \cdot x_n$$

Lo que nos lleva a que cada columna de la matriz  $X$  tiene que ser un autovector de  $A$  puesto que cumple,

$$A \cdot x_i = \lambda_i \cdot x_i$$

En realidad, cada valor propio expande un subespacio  $E_\lambda S$  de  $\mathbf{C}^n$ . Aplicar la matriz  $A$ , a un vector de  $E_\lambda S$  es equivalente a multiplicar el vector por  $\lambda$ . Cada subespacio asociado a un autovalor recibe el nombre de autosubespacio o subespacio propio.

**Polinomio característico.** El polinomio característico de una matriz  $A$  de dimensión  $n \times n$ , se define como,

$$P_A(z) = \det(zI - A)$$

Donde  $I$  es una matriz identidad de dimensión  $n \times n$ . Así por ejemplo para una matriz de dimensión  $2 \times 2$ ,

$$\begin{aligned} P_A(z) &= \det \left( z \cdot \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} - \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \right) = \begin{vmatrix} z - a_{11} & -a_{12} \\ -a_{21} & z - a_{22} \end{vmatrix} = \\ &= (z - a_{11}) \cdot (z - a_{22}) - a_{12} \cdot a_{21} = z^2 - (a_{11} + a_{22}) \cdot z + a_{11} \cdot a_{22} - a_{12} \cdot a_{21} \end{aligned}$$

Los autovalores  $\lambda$  de una matriz  $A$  son las raíces del polinomio característico de la matriz  $A$ ,

$$p_A(\lambda) = 0$$

Las raíces de un polinomio pueden ser simples o múltiples, es decir una misma raíz puede repetirse varias veces. Por tanto, los autovalores de una matriz, pueden también repetirse. Se llama multiplicidad algebraica de un autovalor al número de veces que se repite.

Además las raíces de un polinomio pueden ser reales o complejas. Para una matriz cuyos elementos son todos reales, si tiene autovalores complejos estos se dan siempre como pares de números complejos conjugados. Es decir si  $\lambda = a + bi$  es un autovalor entonces  $\lambda^* = a - bi$  también lo es.

Veamos algunos ejemplos:

La matriz,

$$A = \begin{pmatrix} 3 & 2 \\ -2 & -2 \end{pmatrix}$$

But then, each column of matrix  $x$  must be an eigenvector of  $A$  because they satisfy,

Actually, each eigenvalue expand a subspace  $E_\lambda S$  of  $\mathbf{C}^n$ . To apply matrix  $A$  to a vector in  $E_\lambda S$  is identical to multiply the eigenvector by  $\lambda$ . Each subspace associated to an eigenvector is called an eigensubspace.

**Characteristic polynomial.** The characteristic polynomial of a matrix  $A$  with dimensions  $n \times n$ , is defined as,

Where  $I$  is the  $n \times n$  identity matrix . For instance, for a matrix with dimensions  $2 \times 2$ ,

The eigenvalues  $\lambda$  of matrix  $A$  are the roots of the characteristic polynomial of matrix  $A$ .

The roots of a polynomial may be single or multiple that is the same root can be repeated several times. So, the autovalues of a matrix can also be repeated.

Besides, the roots of a polynomial may be real or complex. For a matrix with real elements, if it has complex eigenvalues their always pairs complex conjugated numbers. That is, if  $\lambda = a + bi$  is an eigenvalue then  $\lambda^* = a - bi$  is also an eigenvalue.

Let's have a look to some examples:

The matrix,

Tiene como polinomio característico,

$$P_A(z) = \begin{vmatrix} z-3 & -2 \\ 2 & z+2 \end{vmatrix} = z^2 - z - 2$$

Igualando el polinomio característico a cero y obteniendo las raíces de la ecuación de segundo grado resultante, obtenemos los autovalores de la matriz  $A$ ,

has characteristic polynomial,

If we equals the characteristic polynomial to zero and compute the roots of the resulting quadratic equation, we obtain the eigenvalues of matrix  $A$

$$\lambda^2 - \lambda - 2 = 0 \rightarrow \begin{cases} \lambda_1 = 2 \\ \lambda_2 = -1 \end{cases}$$

Un vector propio asociado a  $\lambda_1 = 2$  sería  $x_1 = [2, -1]^T$ ,

An associated eigenvector to  $\lambda_1 = 2$  would be  $x_1 = [2, -1]^T$ ,

$$\begin{pmatrix} 3 & 2 \\ -2 & -2 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ -1 \end{pmatrix} = 2 \cdot \begin{pmatrix} 2 \\ -1 \end{pmatrix} = \begin{pmatrix} 4 \\ -2 \end{pmatrix}$$

y un vector propio asociado a  $\lambda_2 = -1$  sería  $x_2 = [1, -2]^T$ ,

An an associated vector to  $\lambda_2 = -1$  would be  $x_2 = [1, -2]^T$ ,

$$\begin{pmatrix} 3 & 2 \\ -2 & -2 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ -2 \end{pmatrix} = -1 \cdot \begin{pmatrix} 1 \\ -2 \end{pmatrix} = \begin{pmatrix} -1 \\ 2 \end{pmatrix}$$

La matriz,

the matrix,

$$B = \begin{pmatrix} 4 & -1 \\ 1 & 2 \end{pmatrix}$$

tiene como polinomio característico,

has characteristic polynomial,

$$P_B(z) = \begin{vmatrix} z-4 & -1 \\ 1 & z-2 \end{vmatrix} = z^2 - 6z + 9$$

procediendo igual que en el caso anterior, obtenemos los autovalores de la matriz  $B$ ,

repeating the same procedure as in the previous example, we obtain the eigenvalues of matrix  $B$ ,

$$\lambda^2 - 6\lambda + 9 = 0 \rightarrow \begin{cases} \lambda_1 = 3 \\ \lambda_2 = 3 \end{cases}$$

En este caso, hemos obtenido para el polinomio característico una raíz doble, con lo que obtenemos un único autovalor  $\lambda = 3$  de multiplicidad algebraica 2.

In this case, we obtain a double root for the characteristic polynomial so, we get a single eigenvector with algebraic multiplicity 2. Matrix,

La matriz,

$$C = \begin{pmatrix} 2 & -1 \\ 1 & 2 \end{pmatrix}$$

tiene como polinomio característico,

has characteristic polynomial,

$$P_C(z) = \begin{vmatrix} z-2 & 1 \\ -1 & z-2 \end{vmatrix} = z^2 - 4z + 5$$

Con lo que sus autovalores resultan ser dos números complejos conjugados,

$$\lambda^2 - 4\lambda + 5 = 0 \rightarrow \begin{cases} \lambda_1 = 2 + i \\ \lambda_2 = 2 - i \end{cases}$$

Para que una matriz  $A$  de orden  $n \times n$  sea diagonalizable es preciso que cada autovalor tenga asociado un número de autovectores linealmente independientes, igual a su multiplicidad algebraica. La matriz  $B$  del ejemplo mostrado más arriba tiene tan solo un autovector,  $x = [1, 1]^T$  asociado a su único autovalor de multiplicidad 2. No es posible encontrar otro linealmente independiente; por lo tanto  $B$  no es diagonalizable.

La matriz,

$$G = \begin{pmatrix} 3 & 0 & 1 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Tiene un autovalor  $\lambda_1 = 3$ , de multiplicidad 2, y un autovalor  $\lambda_2 = 1$  de multiplicidad 1. Para el autovalor de multiplicidad 2 es posible encontrar dos autovectores linealmente independientes, por ejemplo:  $x_1 = [1, 0, 0]^T$  y  $x_2 = [0, 1, 0]^T$ . Para el segundo autovalor un posible autovector sería,  $x_3 = [-2, 0, 1]^T$ . Es posible por tanto diagonalizar la matriz  $G$ ,

$$G = \begin{pmatrix} 3 & 0 & 1 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{pmatrix} = X \cdot D \cdot X^{-1} = \begin{pmatrix} 1 & 0 & -2 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 3 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 2 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

So, its eigenvalues are two complex conjugate numbers.

For an  $A$  matrix of dimension  $ntimesn$  be diagonalisable it is necessary that any eigenvalue has a number of linearly independent eigenvector that meet its algebraic multiplicity. Matrix  $B$  in the example showed above has a single eigenvector,  $x = [1, 1]^T$  associated to its single eigenvalue of multiplicity 2; thus, it is impossible to find another eigenvector linearly independent and  $B$  is not a diagonalisable matrix.

The matrix,

has an eigenvalue  $\lambda_1 = 3$ , with multiplicity 2, and an eigenvalue  $\lambda_2 = 1$  with multiplicity 1. It is possible to find to linearly independent vectors for the eigen vector of multiplicity 2, for example:  $x_1 = [1, 0, 0]^T$  y  $x_2 = [0, 1, 0]^T$ . A possible eigenvector for the second eigenvalues may be,  $x_3 = [-2, 0, 1]^T$ . Thus, it is possible to diagonalise matrix  $G$ ,



**Propiedades asociadas a los autovalores.**  
Damos a continuación, sin demostración, algunas propiedades de los autovalores de una matriz,

1. La suma de los autovalores de una matriz, coincide con su traza,

$$\text{tr}(A) = \sum_{i=1}^n \lambda_i$$

**Eigenvalues associated properties** We include below, without proofs, some properties of a matrix eigenvalues

1. the addition of the eigenvalues of a matrix equals the matrix trace.

2. El producto de los autovalores de una matriz coincide con su determinante,

2. the product of the eigenvalues of a matrix equals the matrix determinant

$$|A| = \prod_{i=1}^n \lambda_i$$

3. Cuando los autovectores de una matriz  $A$  son ortogonales entre sí, entonces la matriz  $A$  es diagonalizable ortogonalmente,

3. When the eigenvectors of a matrix  $A$  are orthogonal among them, then matrix  $a$  is orthogonally diagonalisable

$$A = Q \cdot D \cdot Q^{-1}; Q^{-1} = Q^T \Rightarrow A = Q \cdot A \cdot Q^T$$

Cualquier matriz simétrica posee autovalores reales y es diagonalizable ortogonalmente. En general, una matriz es diagonalizable ortogonalmente si es normal:  $A \cdot A^T = A^T \cdot A$

4. El mayor de los autovalores en valor absoluto, de una matriz  $A$  de orden  $n$ . recibe el nombre de *radiopectral* de dicha matriz,

Any symmetric matrix has real eigenvalues and is orthogonally diagonalisable. In general, a matrix is orthogonally diagonalisable if it is normal:  $A \cdot A^T = A^T \cdot A$

4. The maximum absolute value of a  $n \times n$  matrix  $A$  eigenvalues is called the spectral radius of matrix  $A$ .

$$\rho(A) = \max_{i=1}^n |\lambda_i|$$

Numpy incluye en el submodule `linalg` la función `eig` para calcular los autovectores y autovalores de una matriz. Esta función admite como variable de entrada una matriz cuadrada de dimensión arbitraria y devuelve como variable de salida un vector con los autovalores de la matriz de entrada y una matriz cuadrada en la que cada columna corresponde a un autovalor. A continuación, se muestran dos formas distintas de llamar a la función `eig`. En la línea In [281] se ha realizado la operación  $X \cdot D \cdot X^{-1}$  para comprobar que se cumple que coincide con la matriz  $A$ .

Numpy includes, in the submodule `linalg`, the function `eig` to calculate the eigenvalues and eigenvector of a matrix. Function `eig` takes a square matrix of arbitrary dimensions as input and returns a vector with the eigenvalues of the input matrix and a square matrix, each of its column is an eigenvector. We show next two different ways of calling function `eig`. In line In [281] we computed the operation  $X \cdot D \cdot X^{-1}$  to check that the results coincides with matrix  $A$ .

In [273]: A

Out[273]:

```
array([[ 3,  4,  2,  5],
       [ 2,  0,  1, -2],
       [ 3,  2,  1,  8],
       [ 5,  2,  3,  2]])
```

In [274]: [autovalores, autovectores] = np.linalg.eig(A)

In [275]: autovalores

Out[275]: array([10.71154715, -4.45608147, 0.70096017, -0.95642586])

```
In [276]: autovectores
Out[276]:
array([[-0.5482369 , -0.40905141, -0.34168564,  0.53836293],
       [-0.05940265,  0.51767323, -0.46625182, -0.17480032],
       [-0.63104339, -0.60944455,  0.78709433, -0.82325502],
       [-0.54561146,  0.43962336,  0.2152735 ,  0.04314365]]))

In [277]: Results = np.linalg.eig(A)
In [278]: Results
Out[278]:
EigResult(eigenvalues=array([10.71154715, -4.45608147,  0.70096017,
   -0.95642586]), eigenvectors=array([[-0.5482369 , -0.40905141, -0.34168564,  0.53836293],
       [-0.05940265,  0.51767323, -0.46625182, -0.17480032],
       [-0.63104339, -0.60944455,  0.78709433, -0.82325502],
       [-0.54561146,  0.43962336,  0.2152735 ,  0.04314365]]))

In [279]: Results.eigenvalues
Out[279]: array([10.71154715, -4.45608147,  0.70096017, -0.95642586])

In [280]: Results.eigenvectors
Out[280]:
array([[-0.5482369 , -0.40905141, -0.34168564,  0.53836293],
       [-0.05940265,  0.51767323, -0.46625182, -0.17480032],
       [-0.63104339, -0.60944455,  0.78709433, -0.82325502],
       [-0.54561146,  0.43962336,  0.2152735 ,  0.04314365]])

\begin{center}
\begin{minipage}{\textwidth}
\begin{minted}[language=python]{}
In [281]: autovectores@np.diag(autovalores)@np.linalg.inv(autovectores)
Out[281]:
array([[ 3.00000000e+00,  4.00000000e+00,  2.00000000e+00,
         5.00000000e+00],
       [ 2.00000000e+00, -2.33146835e-15,  1.00000000e+00,
        -2.00000000e+00],
       [ 3.00000000e+00,  2.00000000e+00,  1.00000000e+00,
        8.00000000e+00],
       [ 5.00000000e+00,  2.00000000e+00,  3.00000000e+00,
        2.00000000e+00]])
\end{minted}

```

### 3.5.4. Factorización QR

La idea es factorizar una matriz  $A$  en el producto de dos matrices; una matriz ortogonal  $Q$  y una matriz triangular superior  $R$ .

### 3.5.4. QR Factorisation

The objective of QR factorisation is to divide a matrix  $A$  in the product of two matrices; one of them an orthogonal matrix  $Q$  and the other one an upper triangular matrix  $R$ .

$$A = Q \cdot R, \leftarrow Q \cdot Q^T = I$$

No vamos tampoco a ver en detalle como se calcula la factorización QR. Podemos calcularla empleando la función de Numpy, incluida en el submodule `linalg`, `[Q,R]=qr(A)`,

```
In [287]: A = np.array([[1,-2,4],[2,5,3],[1,3,-3]])
```

```
In [288]: [Q,R] = np.linalg.qr(A)
```

```
In [289]: Q@Q.T
```

```
Out[289]:
```

```
array([[1.0000000e+00, 2.56739074e-16, 8.32667268e-17],
       [2.56739074e-16, 1.0000000e+00, 5.55111512e-17],
       [8.32667268e-17, 5.55111512e-17, 1.0000000e+00]])
```

```
In [290]: R
```

```
Out[290]:
```

```
array([[-2.44948974, -4.4907312 , -2.85773803],
       [ 0.          , -4.22295315,  3.51254982],
       [ 0.          ,  0.          , -3.67359866]])
```

### 3.5.5. Factorización SVD

Dada una matriz cualquiera  $A$  de orden  $m \times n$  es posible factorizarla en el producto de tres matrices,

$$A = U \cdot S \cdot V^T$$

Donde  $U$  es una matriz de orden  $m \times m$  ortogonal,  $V$  es una matriz de orden  $n \times n$  ortogonal y  $S$  es una matriz diagonal de orden  $m \times n$ . Además los elementos de  $S$  son positivos o cero y están ordenados en orden no creciente,

$$S = \begin{pmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_i \end{pmatrix}; \quad \sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_i; \quad i = \min(m, n)$$

Los elementos de la diagonal de la matriz  $S$ ,  $(\sigma_1, \sigma_2, \dots, \sigma_i)$ , reciben el nombre de *valores singulares* de la matriz  $A$ . De ahí el nombre que recibe esta factorización; SVD son las siglas en inglés de *Singular Value Decomposition*.

No vamos a describir ningún algoritmo para obtener la factorización SVD de una matriz. En Numpy existe la función `[U,s,VT]=svd(A)`,

We are not going to get into details on how to calculate a QR factorisation. We can use a Numpy function included in submodule `linalg`, `[Q,R]=qr(A)`,

### 3.5.5. SVD Factorisation

[eng] Any matrix  $A$  of dimensions  $m \times n$  can be factorised as the product of three matrices,

Where  $U$  is an orthogonal matrix of dimensions  $m \times m$ ,  $V$  is an orthogonal matrix of dimensions  $n \times n$  and  $S$  Is a diagonal matrix of dimensions  $m \times n$ . Besides, the entries of  $S$  are positive or zero and are ordered in non increasing order.

The entries of the diagonal of matrix  $S$ ,  $(\sigma_1, \sigma_2, \dots, \sigma_i)$ , are known as the singular values of matrix  $A$ . This is the reason why this factorisation is named *Singular Value Decomposition*, SVD.

We are not going to describe any algorithm to get the SVD factorisation of a matrix. In Numpy, the function `[U,s,VT]= svd(A)`, which belongs to the `linalg` submodule, allow

dentro del submodulo `linalg`, que permite obtener directamente la factorización SDV de una matriz  $A$  de dimensión arbitraria. Es importante destacar que Numpy devuelve un vector  $s$  con los valores singulares y, directamente, la matriz  $V^T$ . Para reconstruir la matriz  $S$  es preciso crear un matriz de dimensión  $m \times n$  a partir del vector  $s$ , primero se crea una matriz diagonal a partir de  $s$  y después se le añaden las filas o columnas de ceros necesarias, hasta completar la mayor de las dimensiones. Si  $n > m$  debemos añadir filas, si  $n < m$  columnas. A continuación se incluyen un ejemplo de uso para una matriz  $A$  no cuadrada,

```
In [93]: A = np.array([[1,3,4],[2,3,2],[2,4,5],[3,2,3]])

In [94]: A
Out[94]:
array([[1, 3, 4],
       [2, 3, 2],
       [2, 4, 5],
       [3, 2, 3]])

In [102]: U,s,VT = np.linalg.svd(A)

In [105]: S[:s.shape[0],:s.shape[0]] = np.diag(s)

In [106]: S
Out[106]:
array([[10.25447551,  0.          ,  0.          ],
       [ 0.          ,  1.90114896,  0.          ],
       [ 0.          ,  0.          ,  1.10966868],
       [ 0.          ,  0.          ,  0.          ]])

In [107]: U@S@VT
Out[107]:
array([[1., 3., 4.],
       [2., 3., 2.],
       [2., 4., 5.],
       [3., 2., 3.]])
```

Como la matriz  $A$  tiene más filas que columnas, la matriz  $S$  resultante termina con una fila de ceros.

A continuación enunciamos sin demostración algunas propiedades de la factorización SVD.

1. El rango de una matriz  $A$  coincide con

us to get straightforwardly the SVD matrix factorisation for a matrix  $A$  with arbitrary dimensions. It is important to notice that Numpy returns a vector  $s$  with the singular values and the matrix  $V^T$ , directly. To build the matrix  $S$  we need to create a matrix of dimensions  $m \times n$  from vector  $s$ . To do so, we first create a diagonal matrix with the entries of  $s$  and then, we pad the matrix with rows or columns of zeros till complete the larger of the matrix dimension. If  $n > m$  we have to add rows, if  $n < m$  columns. The following code shows and example for a non-square matrix

Matrix  $A$  has more rows than columns. For this reason, matrix  $S$  ends with a row of zeros.

Next, we include without demonstrations some SVD factorisation properties.

1. The rank of a matrix  $A$  coincides with the number of its non-zero singular values.

el número de sus valores singulares distintos de cero.

2. La norma-2 inducida de una matriz  $A$  coincide con su valor singular mayor  $\sigma_1$ . 3. La norma de Frobenius de una matriz  $A$  cumple:

$$\|A\|_F = \sqrt{\sigma_1^2 + \sigma_2^2 + \cdots + \sigma_r^2}$$

4. Los valores singulares de una matriz  $A$  distintos de cero son iguales a la raíz cuadrada positiva de los autovalores distintos de cero de las matrices  $A \cdot A^T$  ó  $A^T \cdot A$ . (Los autovalores distintos de cero de estas dos matrices son iguales),

$$\sigma_i^2 = \lambda_i(A \cdot A^T) = \lambda_i(A^T \cdot A)$$

5. El valor absoluto del determinante de una matriz cuadrada  $A, n \times n$ , coincide con el producto de sus valores singulares,

$$|\det(A)| = \prod_{i=1}^n \sigma_i$$

6. El número de condición de una matriz cuadrada  $A n \times n$ , que se define como el producto de la norma-2 inducida de  $A$  por la norma-2 inducida de la inversa de  $A$ , puede expresarse como el cociente entre el el valor singular mayor de  $A$  y su valor singular más pequeño,

$$k(A) = \|A\|_2 \cdot \|A^{-1}\|_2 = \sigma_1 \cdot \frac{1}{\sigma_n} = \frac{\sigma_1}{\sigma_n}$$

El número de condición de una matriz, es una propiedad importante que permite estimar cómo de estables serán los cálculos realizados empleando dicha matriz, en particular aquellos que involucran directa o indirectamente el cálculo de su inversa.

2. The induced 2-norm of a matrix  $A$  is equal to its largest singular value  $\sigma_1$ .

3. The Frobenius norm of matrix satisfies,

4. The non-zero singular values of a matrix  $A$  are equal to the positive square root of the non-zero eigen values of the matrices  $A \cdot A^T$  ó  $A^T \cdot A$ . (The non-zero eigenvalues of these matrix are equal by obvious reasons),

5. The absolute value of a square  $n$  times  $n$  matrix  $A$  determinant its equal to the product of the matrix singular values.

6. The condition number of a square  $n \times n$  matrix  $A$ , which is defined as the product of the  $A$  induced 2-norm by the induced 2-norm of the inverse of  $A$ , may be expressed as the quotient between the largest singular value of  $A$  and its lower singular value,

The condition number of a matrix, is very useful property that allows us to estimate how stable will be the operations performed using the matrix, i particular, those operations with involve directly or indirectly the computing of the matrix inverse.

## 3.6. Ejercicios

Para los siguientes ejercicios, es necesario importar Numpy. Se puede hacer de tres maneras distintas:

```
import numpy
import numpy as alias
from numpy import *
```

En el primer caso, hay que escribir las funciones precedidas del nombre del módulo: `numpy.fun`. EN el segundo caso, hay que precederlas del alias elegido, en los ejercicios que siguen es la opción elegida

y el alias empleado es np: np.fun. EN el tercer caso, importamos directamente las funciones y submódulos por lo que pudríamos emplearlos usando directamente su nombre fun. Este último método tiene el riesgo de sobreescribir accidentalmente alguna función de Numpy.

1. Arrays en Numpy; matrices y vectores. Escribe, por orden, las siguientes expresiones en la *línea de comandos* de Ipython e interpreta los resultados que obtienes. En los casos en que Python devuelva un mensaje de error, trata de averiguar la razón. **Nota:** Es importante hacerlos por orden ya que algunos operaciones se apoyan en los resultados de operaciones anteriores.

```

1 a =np.array([[1, 2, 3],[4, 5,
6]])
2 b =np.array([[1, 2, 3],[4, 5]])
3 c = np.array([1,2,3])
4 d = np.array([[1,2,3]])
5 e = np.array([[1],[2],[3]])
6 f = np.zeros((3,3))
7 f1 = np.zeros(3)
8 f2 = np.zeros((3,))
9 f3 = np.zeros((3,1))
10 g = np.eye(4)
11 h = np.eye(4,3)
12 i = np.eye(4,6)
13 j = np.identity(4)
14 q = np.arange(0,1,0.2)
15 k = np.diag(c)
16 l = np.diag(k)
17 r=np.diag(q,1)
18 r=np.diag(q,2)
19 r=np.diag(q,0)
20 r=np.diag(q,-1)
21 r=np.diag(q,-2)
* indexing and slicing
22 n = a[0,1]
23 du = a[0]
24 duf = a[:,1]
25 duc = a[1,:]
26 dd = a[0:2,1:3]
* Operadores aplicados a matrices
27 ash = np.shape(a)
28 fa,ca =a.shape
29 fc,cc = np.shape(c)
30 shc = c.shape
31 a1= a.shape[0]
32 a2= c.shape[1]
33 o1=ones(a.shape[1])
34 o2=ones(c.shape[0])
35 o3=ones((c.shape[0],a.shape[0]))
36 tt=eye(1,3)
37 p1=a*c
38 p2=c*a
39 p3 = a*e
40 p4 = a@c
41 p5 = c@a
42 p6 = a@e
43 p7 = e@a
44 p10 = dd@duf
45 p11 = duf@dd
46 p8 = a*3
47 b = np.array([[3,5,6],[5,6,7]])
48 bt=b.T
49 aT=a.T
50 r=a-b
51 z=a-a
52 j1=a+b
53 j2=a.T+b
54 j3 = a.T+b.T
55 d=(a+b)@b.T
56 f=(a+b)*b.T
57 f=(a-b).T*b.T
58 p=a**2
59 r=a**2-a*a

```

```

60 A = np.append(a, [[3,-2,1]],axis=0)
61 B = np.append(b, [[1,0,-1]],axis=0)
    * Encadenando operaciones
62 w1 = A@np.linalg.inv(B)
63 w2 = A@np.linalg.matrix_power(B,-1)
64 w3 = A/B
65 w4 = B/A
    * importamos linalg / we import
      linalg
66 from numpy import linalg as la
67 w5 = A@la.inv(B)-A@la.matrix_power(B,-1)
68 w6 = A@la.matrix_power(B,-1)
      -la.solve(la.inv(A),la.inv(B))
69 C = np.append(A, [[4,0,-3],[2,2,4]],axis=0)
70 C = np.append(C, array([[4,3,2,1,0]]).T, axis
      = 1)
71 C1 = C[0:4:2,3]
72 C2=C[0:3,:]
73 C3 = C[1:3,1:3]

```

2. Define en un terminal de Ipython las matriz  $M$  y  $N$  e interpreta el resultado de las condiciones lógicas impuestas.

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}, N = \begin{pmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \\ 8 & 9 & 10 \end{pmatrix}$$

nota & es equivalente a [and](#)

```

y1=M<2
y2=M<=2
y3=M>2
y4=M>=2
y5=M==2
y6=M!=2
y7=any(M<2)
y8=any(M<2,0)
y9=any(M<2,1)
y10=any(M==2)
y11=all(M>=2)
y12=all(M!=0)
y13=all(any(M<2))
y14=any(all(M<2))
index=np.argwhere(M>2)
y15=any(M<2)&any(N>4)
y16=any(M<2)&any(N>10)
y17=any(M<2)&any(N>6)
y18=any(M<2)&any(N>7)
y19=any(M<2)&any(N>3)
y20=any(M<2)&any(N>2)
y21=any(M>2)&any(N>2)
y22=any(M>4)&any(N>2)
y23=any(M>4,0)&any(N>2)
y24=any(M>4,1).T&any(N>2)

```

Repite el estudio de las condiciones lógicas cambiando & por |, (equivalente a [or](#))

3. Obtén empleando el terminal de Ipython el resultado de las siguientes expresiones:

$\frac{5 + 3x^2}{6y}$ $\frac{1 + 4\sqrt{2x}}{6}y$ $\frac{-x + \sqrt{x^2 - 4xy}}{2x}$	$\frac{1}{\sqrt{2\pi}}e^{-\frac{1}{2}(x-1)^2}$ $\sin^2(2x) + \cos^2(2x)$ $\arctan(\infty)$ $\arctan\left(\frac{y}{x}\right)$
--	--

Emplea para  $x$  e  $y$  tanto valores escalares como matrices.

4. Dada una matriz  $A \in \mathbb{R}^{n \times n}$  y un vector columna  $v \in \mathbb{R}^n - \mathbf{0}$ . Se dice que el vector  $v$  pertenece al *Kernel* de  $A$  si se cumple que  $Av = \mathbf{0}$ .

Crea una función que tome como variables de entrada una matriz  $A$  cuadrada de cualquier dimensión  $n \times n$  y un vector columna de dimensión  $n$  y compruebe si pertenece o no al *kernel* de  $A$ . **Nota:** Considera que la condición se cumple si todos los elementos del vector  $Av$  son menores que  $1e - 12$ . Hazlo sin emplear el operador  $\circledast$ .

5. Escribe —empleando bucles una función que reciba como argumentos de entrada dos matrices  $X$  e  $Y$  y devuelva el valor de su suma  $S = X + Y$ . El programa debe comprobar si es posible la operación indicada y, caso de no serlo, interrumpir la ejecución y mostrar un mensaje de error.
6. Implementa una función `SumoDiag.m` que tome como entrada una matriz cuadrada  $A$  y que devuelva la suma de todos los elementos de las 2 diagonales principales de dicha matriz. No utilizar la función `diag()`.
7. Escribe —empleando bucles y sin hacer uso del operador  $\circledast$  — que reciba como argumentos de entrada dos matrices  $X$  e  $Y$  y devuelva el valor del producto  $P = XY$ . El programa debe comprobar si es posible la operación indicada y, caso de no serlo, interrumpir la ejecución y mostrar un mensaje de error.

## Capítulo/Chapter 4

# Introducción a matplotlib Introduction to matplotlib

El punto geométrico es invisible. De modo que lo debemos definir como un ente abstracto. Si pensamos en él materialmente, el punto se asemeja a un cero. Cero que, sin embargo, oculta diversas propiedades «humanas». Para nuestra percepción este cero —el punto geométrico—está ligado a la mayor concisión. Habla, sin duda, pero con la mayor reserva.

---

Vasili Kandinski, Punto y línea sobre plano

Visualizar los datos es fundamental para poder comprender y transmitir ideas e información en ciencias e ingeniería. Python tiene numerosas funciones gráficas que para mostrar muchos tipos de gráficos. Una de las librerías para visualizar datos de uso más extendido es **matplotlib** a la que dedicamos este capítulo.

Importaremos la biblioteca matplotlib de la siguiente manera

```
import matplotlib as mpl
```

**matplotlib.pyplot** es una colección de funciones que hacen que matplotlib funcione como MATLAB. Cada función de pyplot realiza algún cambio en una figura: por ejemplo, crea

Visualising data is essential for understanding and conveying ideas and information in science and engineering. Python has numerous graphics functions for displaying many types of graphics. One of the most widely used data visualisation libraries is **matplotlib** to which we dedicate this chapter.

We can import matplotlib as follows:

**matplotlib.pyplot** is a collection of functions that make matplotlib work like MATLAB. Each pyplot function makes some change to a figure: for example, it creates a figure, creates a

una figura, crea un área de trazado en una figura, traza algunas líneas en un área de trazado, decora el trazado con etiquetas, etc.

En matplotlib.pyplot la figura se conserva a través de las llamadas a funciones, de modo que las funciones de trazado se ejecutan sobre los ejes actuales.

Para crear gráficas usando Matplotlib necesitaremos una figura. Cada figura tiene un par (o más) de ejes y un área que es donde se dibujarán los puntos en el sistema de coordenadas que decidamos. Además podremos poner títulos, leyendas etc...

La manera más sencilla de crear una figura es empleando el método `figure` que creará una figura en la que posteriormente podremos añadir ejes, títulos y más cosas. Simplemente llamando a `figure` creamos un objeto de tipo figura, aunque también podemos darle de manera opcional parámetros como el tamaño, el color de fondo y más.

```
import matplotlib.pyplot as plt
fig = plt.figure(figsize=(2, 2), facecolor='lightskyblue',
                  layout='constrained')
fig = plt.figure
```

## 4.1. Dibujar en 2D

La manera más sencilla de dibujar usando matplotlib.pyplot es mediante el método `plot`. A este método le pasaremos dos vectores con las coordenadas  $x$  e  $y$  de los puntos que queremos representar (evidentemente tendremos que asegurarnos de que tienen la misma longitud).

Por ejemplo si generamos unos vectores de coordenadas como los siguientes  $x = (0, 2, -1, -2)$  e  $y = (0, 3, 2, -4)$ , y se los pasamos al método `plot` pyplot dibujará los puntos  $(x, y)$  y los unirá con rectas, como puede verse en la figura 4.1.

```
import numpy as np
import matplotlib.pyplot as plt

x=np.array([0, 2, -1, -2])
y=np.array([0, 3, 2, -4])
plt.figure
plt.plot(x,y)
```

plot area in a figure, draws some lines in a plot area, decorates the plot with labels, etc.

In matplotlib.pyplot the figure is preserved across function calls, so that the plotting functions are executed on the current axes.

To create graphs using Matplotlib we will need a figure. Each figure has a pair (or more) of axes and an area which is where the points will be drawn in the coordinate system of your choice. In addition we can put titles, legends etc...

The easiest way to create a figure is to use the `figure` method which will create a figure to which we can later add axes, titles and more. Simply by calling `figure` we create an object of type `figure`, although we can also optionally give it parameters such as size, background colour and more.

The easiest way to draw using matplotlib.pyplot is by using the `plot` method. To this method we will pass two vectors with the  $x$  and  $y$  coordinates of the points we want to represent (obviously we will have to make sure that they have the same length).

For example, if we generate coordinate vectors such as  $x = (0, 2, -1, -2)$  and  $y = (0, 3, 2, -4)$ , and pass them to the `pyplot` method, it will draw the points  $(x, y)$  and join them with straight lines, as can be seen in the figure 4.1.

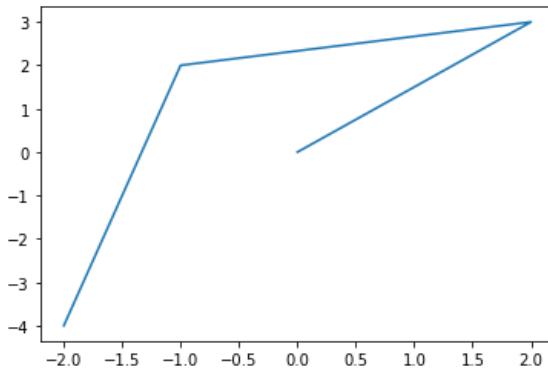


Figura 4.1: Ejemplo sencillo de pyplot.

Figure 4.1: Simple pyplot example.

Símbolo	Color
'b'	azul
'r'	rojo
'g'	verde
'm'	magenta
'c'	cian
'y'	amarillo
'k'	negro
'w'	blanco

Tabla 4.1: Colores de los puntos

Además de las coordenadas de los puntos  $(x, y)$  al método `pyplot.plot` podemos darle un tercer argumento indicando el formato (color y forma) con el que queremos que se grafiquen los puntos y opcionalmente la línea que los une.

Algunos de los colores y formatos admitidos están en las tablas 4.1, 4.2 y 4.3. Para más información consulta la ayuda de `pyplot.plot`.

Se puede combinar un símbolo de cada tipo en un mismo `plot`. Así por ejemplo si queremos representar los datos anteriores unidos mediante una línea de puntos,

```
plt.plot(x,y,':')
```

Si queremos que pinte solo los puntos sin unirlos con líneas y en color rojo,

```
plt.plot(x,y,'.r')
```

Si queremos que pinte los puntos representados por triángulos con el vértice hacia arri-

Symbol	Color
'b'	blue
'r'	red
'g'	green
'm'	magenta
'c'	cyan
'y'	yellow
'k'	black
'w'	white

Tabla 4.1: Point colors

In addition to the coordinates of the points  $(x, y)$ , we can give the `pyplot.plot` method a third argument indicating the format (colour and shape) in which we want the points to be plotted and optionally the line that joins them.

Some of the colours and formats supported are in the tables 4.1, 4.2 and 4.3. For more information, see the help for `pyplot.plot`.

It is possible to combine one symbol of each type in the same text. So for example if we want to represent the above data joined by a dotted line,

```
plt.plot(x,y,':')
```

If we want it to paint only the dots without joining them with lines and in red,

```
plt.plot(x,y,'.r')
```

If we want it to paint the points represented by triangles with the vertex upwards, joined by a continuous line and in black,

Símbolo	Formato punto
'.'	punto
','	píxel
'o'	círculo
's'	cuadrado
'>'	triángulo hacia abajo
'v'	triángulo hacia arriba
'p'	pentágono
'*'	asterisco

Tabla 4.2: Algunos de los marcadores de puntos

Símbolo	Tipo de línea
'_'	sólida
'-'	discontínua
'-.'	punto-rayo
'.'	de puntos

Tabla 4.3: Tipos de línea

ba, unidos mediante una línea continua y en color negro,

```
plt.plot(x,y, '-^k')
```

La figura 4.2 muestra los resultados de las combinaciones de símbolos que acabamos de describir.

**Scatter.** También podemos dibujar los puntos  $x$ ,  $y$ , por separado sin unirlos usando el método `scatter(x,y)`.

#### 4.1.1. Trabajar con múltiples figuras

Con `matplotlib.pyplot` se pueden dibujar diferentes gráficas en una misma figura usando el método `subplot`. Con este método crearemos una figura donde las distintas gráficas estarán dispuestas a modo de tabla. A este método le indicaremos mediante parámetros de entrada el número de filas, el número de columnas y la posición donde se va a dibujar la siguiente gráfica.

En el siguiente ejemplo se crean dos gráficas dentro de la misma figura en dos filas y una columna. En la primera se dibuja la gráfi-

Symbol	Point format
'.'	point
','	pixel
'o'	circle
's'	square
'>'	down triangle
'v'	up triangle
'p'	pentagon
'*'	star

Tabla 4.2: Some point markers

Symbol	Line type
'_'	solid
'-'	dashed
'-.'	dash and dot
'.'	dotted

Tabla 4.3: Line types

```
plt.plot(x,y, '-^k')
```

Figure 4.2 shows the results of the symbol combinations just described.

**Scatter.** We can also draw the points  $x$ ,  $y$ , separately without joining them using the method `scatter(x,y)`.

#### 4.1.1. Working with multiple figures

With `matplotlib.pyplot` you can draw different graphs in the same figure using the `subplot` method. With this method we will create a figure where the different graphs will be arranged as a table. To this method we will indicate by means of input parameters the number of rows, the number of columns and the position where the next graph is going to be drawn.

In the following example, two graphs are created within the same figure in two rows and one column. In the first one the cosine graph

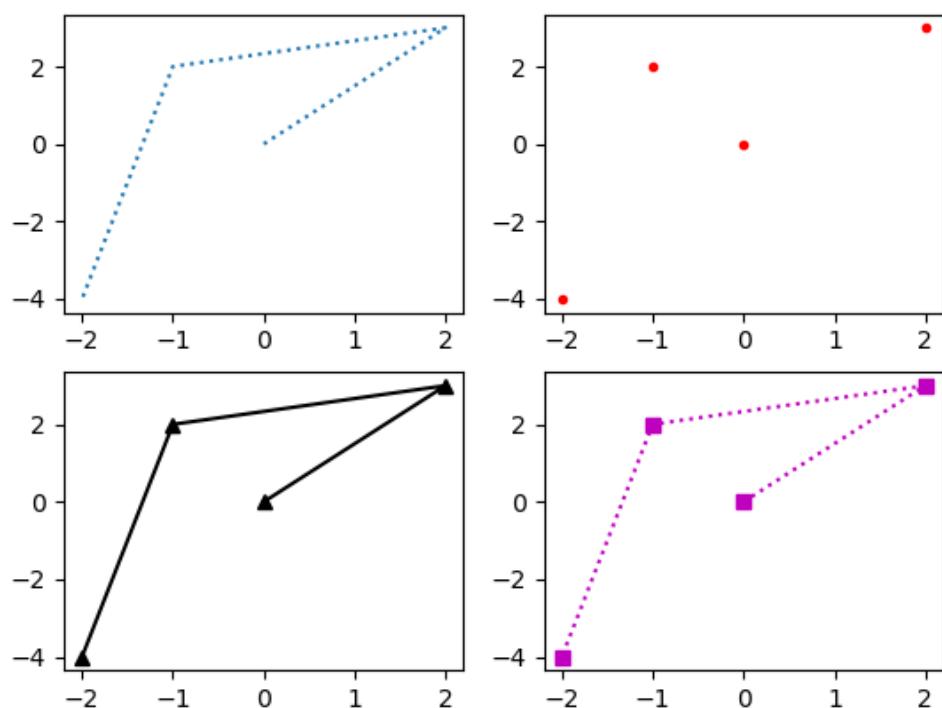


Figura 4.2: Ejemplos de dibujos con diferente formato  
Figure 4.2: Different uses of plot

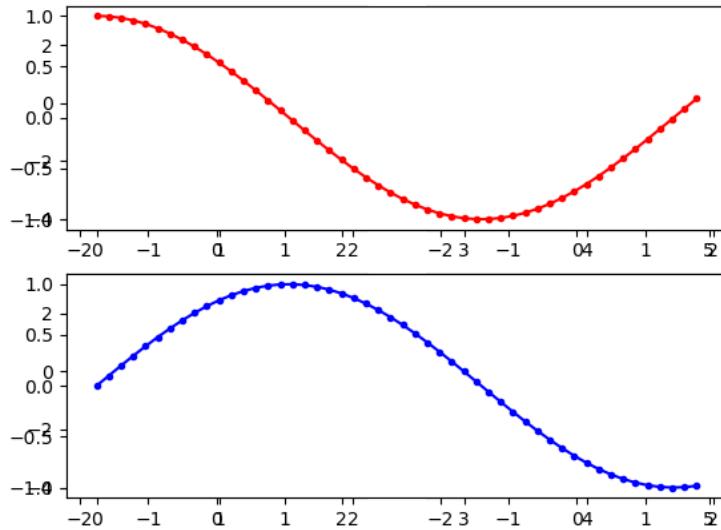


Figura 4.3: Dos gráficas en una misma figura usando subplot

Figure 4.3: Two graphs in one figure using subplot

ca del coseno y en la segunda la del seno. El resultado puede verse en la figura 4.3.

```
import numpy as np
import matplotlib.pyplot as plt
t1 = np.arange(0.0, 5.0, 0.1)
plt.subplot(2,1,1)
plt.plot(t1,np.cos(t1),'r.-')
plt.subplot(2,1,2)
plt.plot(t1,np.sin(t1),'b.-')
```

Se puede añadir texto en cualquier parte del dibujo mediante el método `text` indicando las coordenadas donde se quiere poner el texto y el texto deseado. También se puede poner un texto como título de la figura, usando `title`, o etiquetando los ejes con `xlabel`, `ylabel`.

Por ejemplo el siguiente código da lugar a la figura 4.4.

```
import numpy as np
import matplotlib.pyplot as plt
plt.figure()
t=np.arange(0.0,5,0.01)
y=np.exp(-t) * np.cos(2*np.pi*t)
plt.plot(t,y)
```

is drawn and in the second one the sine graph is drawn. The result can be seen in figure 4.3.

```
import numpy as np
import matplotlib.pyplot as plt
t1 = np.arange(0.0, 5.0, 0.1)
plt.subplot(2,1,1)
plt.plot(t1,np.cos(t1),'r.-')
plt.subplot(2,1,2)
plt.plot(t1,np.sin(t1),'b.-')
```

Text can be added anywhere on the drawing using the `text` method, indicating the coordinates where the text is to be placed and the desired text. You can also put a text as the title of the figure, using `title`, or by labelling the axes with `xlabel`, `ylabel`.

For example, the following code results in the following figure 4.4.

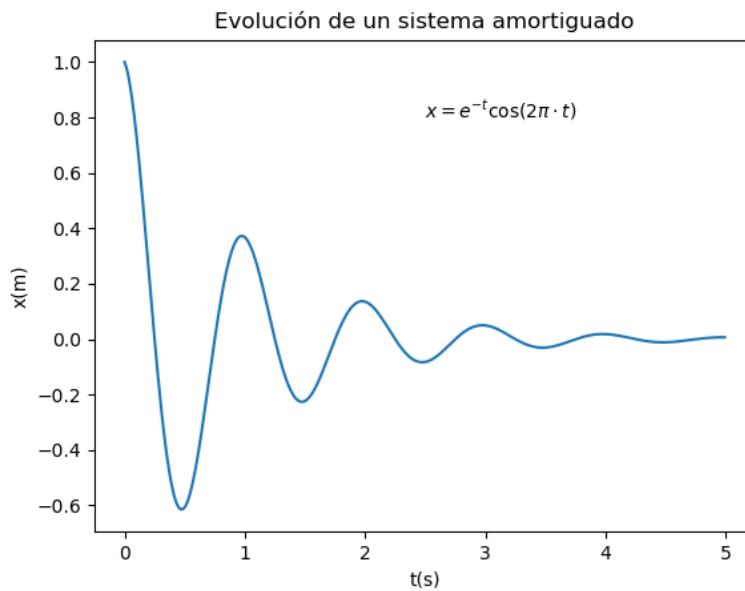


Figura 4.4: Figura con texto en etiquetas  
Figure 4.4: Figure with text and labels

```
plt.title("Evolución de un sistema amortiguado")
plt.xlabel('t(s)')
plt.ylabel('x(m)')
plt.text(2.5,0.8,'$x=e^{-t}\cos(2\pi \cdot t)$')
```

#### 4.1.2. Ejes no lineales

`Matplotlib.pyplot` cuenta con un método para definir la escala de los ejes, son los métodos `xscale` e `yscale`. Estos métodos pueden recibir como argumento el tipo de escala del eje, a elegir entre los siguientes:

- "linear" para definir una escala lineal
- "log" para definir una escala logarítmica de eje positivo
- "symlog" para definir una escala logarítmica con tramo positivo y negativo centrado en el cero (como los valores del logaritmo en torno al cero tienden a infinito hay una zona en torno al cero donde considera la escala lineal)
- "logit" escala logarítmica entre 0 y 1

#### 4.1.2. Nonlinear axis

`Matplotlib.pyplot` has a method to define the scale of the axes, these are the methods `xscale` and `yscale`. These methods can take as an argument the scale type of the axis, choose from the following:

- 'linear' to define a linear scale
- 'log' to define a positive-axis logarithmic scale
- 'symlog' to define a logarithmic scale with a positive and negative span centred at zero (as the values of the logarithm around zero tend to infinity there is a zone around zero where the linear scale is considered)
- 'logit' logarithmic scale between 0 and 1.

Tanto `xscale` como `yscale` permiten definir la escala mediante una función que le definimos nosotros.

En el siguiente ejemplo de código puede verse un ejemplo de uso de los métodos `xscale` e `yscale` para generar la figura 4.5.

Como se ve en la figura 4.5 las divisiones del eje logarítmico (eje y en la segunda figura y ambos en la tercera) aparecen marcadas como potencias de 10. Como estamos representando empleando el logaritmo decimal de la variable, las divisiones se corresponden con el exponente de la potencia de 10 de cada división,  $\log_{10}(10^n) = n$ .

En segundo lugar hemos empleado un nuevo método: `grid()`. Este método añade una retícula al gráfico de modo que sea más fácil ver los valores que toman las variables en cada punto de la gráfica. Si se invoca el método `grid()` sin parámetros cambia la retícula de visible a invisible y viceversa.

```
import numpy as np
import matplotlib.pyplot as plt
x=np.linspace(0,10,100)
y=np.exp(x)
plt.figure()

plt.subplot(3,1,1)
plt.plot(x,y)
plt.grid()
plt.xlabel("Escala lineal")
plt.ylabel("Escala lineal")
plt.title("$y=e^x$")

plt.subplot(3,1,2)
plt.plot(x,y)
plt.yscale('log')
plt.xlabel("Escala lineal")
plt.ylabel("Escala logarítmica")
plt.grid()

plt.subplot(3,1,3)
plt.plot(x,y)
plt.xscale('log')
plt.yscale('log')
plt.xlabel("Escala logarítmica")
plt.ylabel("Escala logarítmica")
plt.grid()
```

Both `xscale` and `yscale` allow the scale to be defined by a function that we define.

The following code example shows an example of using the `xscale` and `yscale` methods to generate the 4.5 figure.

As can be seen in the figure 4.5 the divisions of the logarithmic axis (y-axis in the second figure and both in the third figure) are marked as powers of 10. As we are representing using the decimal logarithm of the variable, the divisions correspond to the exponent of the power of 10 of each division,  $\log_{10}(10^n) = n$ .

Secondly, we have employed a new method: `grid()`. This method adds a grid to the graph so that it is easier to see the values taken by the variables at each point on the graph. Calling the `grid()` method without parameters toggles the visibility of the grid.

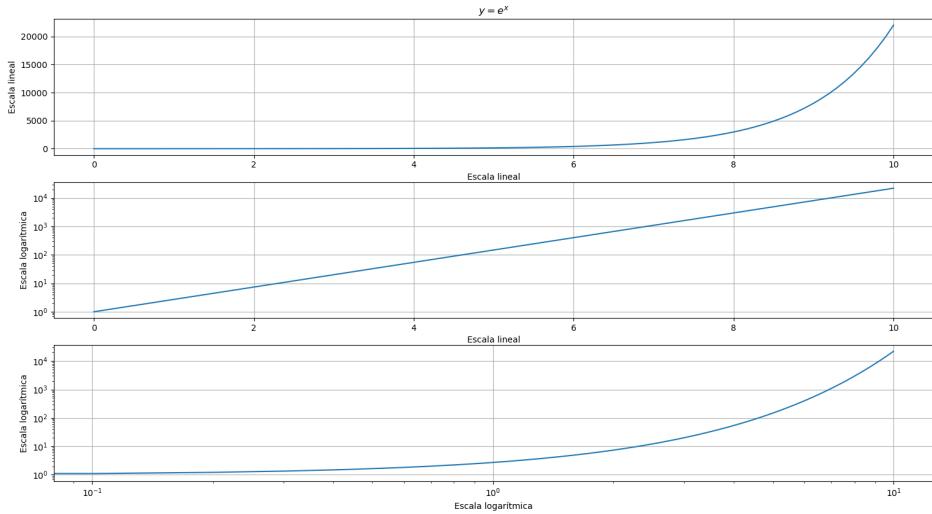


Figura 4.5: Gráfica de la función  $y = e^x$  con ejes en escala lineal y logarítmica  
Figure 4.5: Graph of the function  $y = e^x$  with axes on linear and logarithmic scale

#### 4.1.3. Gráfica en coordenadas polares

Usando el método `polar` podemos representar funciones en coordenadas polares. El primer argumento es un ángulo en radianes y el segundo el correspondiente radio. La figura 4.6 muestra la espiral,

$$r = 2 \cdot \sqrt{\theta}$$

Para el intervalo angular  $[0, 8\pi]$ .

#### 4.1.3. Polar coordinates graph

Using the `polar` method we can plot functions in polar coordinates. The first argument is an angle in radians and the second the corresponding radius. The figure 4.6 shows the spiral,

$$r = 2 \cdot \sqrt{\theta} \quad (4.1)$$

For the angular interval  $[0, 8\pi]$ .

```
import numpy as np
import matplotlib.pyplot as plt

theta=np.linspace(0,8*np.pi,100)
radio=2*theta
plt.figure()
plt.polar(theta,radio,'b.-')
plt.grid(visible=True)
plt.title("Espiral en polares")
```

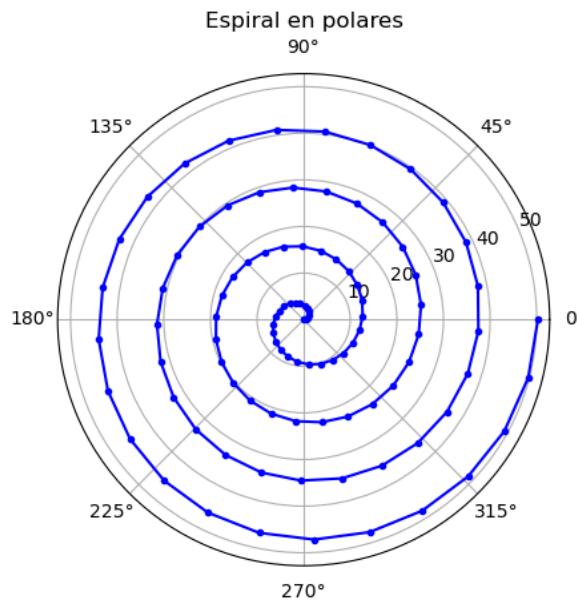


Figura 4.6: Espiral en coordenadas polares  
Figure 4.6: Spiral plot using polar coordinates

#### 4.1.4. Otras representaciones gráficas

Con `matplotlib.pyplot` tenemos más opciones para hacer otro tipo de representaciones gráficas.

**hist** . Este método permite dibujar el histograma de una colección de datos. El histograma representa cuántos datos de una colección caen dentro de un intervalo dado. El método `hist` necesita como parámetro de entrada un vector de datos. Si no se le indica otra cosa los datos se agrupan en 10 intervalos. Si se quiere cambiar el número de intervalos se puede indicar con el parámetro `bin`. En el siguiente ejemplo se dibuja el histograma de los datos de coches por cada 1000 habitantes usando el método `hist` con el número de intervalos por defecto, con 5 intervalos y con 10. Los datos se cargan de un fichero llamado `coches.dat` y la figura 4.7 es la figura resultante.

```
import numpy as np
import matplotlib.pyplot as plt
```

#### 4.1.4. Other graphical representations

With `matplotlib.pyplot` we have more options to make other types of graphical representations.

**hist** . This method allows you to draw the histogram of a collection of data. The histogram represents how much data in a collection falls within a given interval. The `hist` method requires a vector of data as input parameter. Unless otherwise specified, the data is grouped into 10 intervals. If you want to change the number of intervals you can specify it with the parameter `bin`. In the following example the histogram of the car data per 1000 inhabitants is drawn using the `hist` method with the default number of intervals, with 5 intervals and with 10 intervals. The data is loaded from a file called `cars.dat` and the resulting figure 4.7 is the resulting figure.

```

# Abrir el archivo en modo lectura
with open('coches.dat', 'r') as file:
    lineas=file.readlines()
    f=len(lineas)
    n=np.zeros([f])
    i=0
    for row in lineas:
        n[i]=float(row)
        i+=1

plt.figure()

plt.subplot(1,3,1)
plt.hist(n)
plt.grid(visible=True)
plt.title("10 intervalos")
plt.xlabel("Coches por cada 1000 habitantes")
plt.ylabel("Número de países")
plt.subplot(1,3,2)
plt.hist(n, bins=5)
plt.grid(visible=True)
plt.title("5 intervalos")
plt.xlabel("Coches por cada 1000 habitantes")
plt.ylabel("Número de países")
plt.subplot(1,3,3)
plt.hist(n,bins=20)
plt.grid(visible=True)
plt.title("20 intervalos")
plt.xlabel("Coches por cada 1000 habitantes")
plt.ylabel("Número de países")

```

## 4.2. Dibujar en 3D

En tres dimensiones es posible representar dos tipos de gráficos: puntos y curvas, análogos a los representados en dos dimensiones y además superficies en el espacio.

**subplots** Este método encapsula todo lo necesario para crear una figura y un conjunto de subfiguras contenidas en ella. Podemos indicarle muchos parámetros como el número de filas y de columnas de la rejilla de subfiguras, si se comparten o no los ejes entre ellas etc. También podemos indicar, mediante un diccionario qué tipo de proyección queremos usar en los ejes creados. Concretamen-

## 4.2. 3D plots

In three dimensions it is possible to represent two types of graphics: points and curves, analogous to those represented in two dimensions, and also surfaces in space.

**subplots** This method encapsulates everything needed to create a figure and a set of subfigures contained in it. We can indicate many parameters such as the number of rows and columns of the grid of subfigures, whether or not the axes are shared between them, etc. We can also indicate, by means of a dictionary, which type of projection we want to use in the axes created. Concretely, by means of

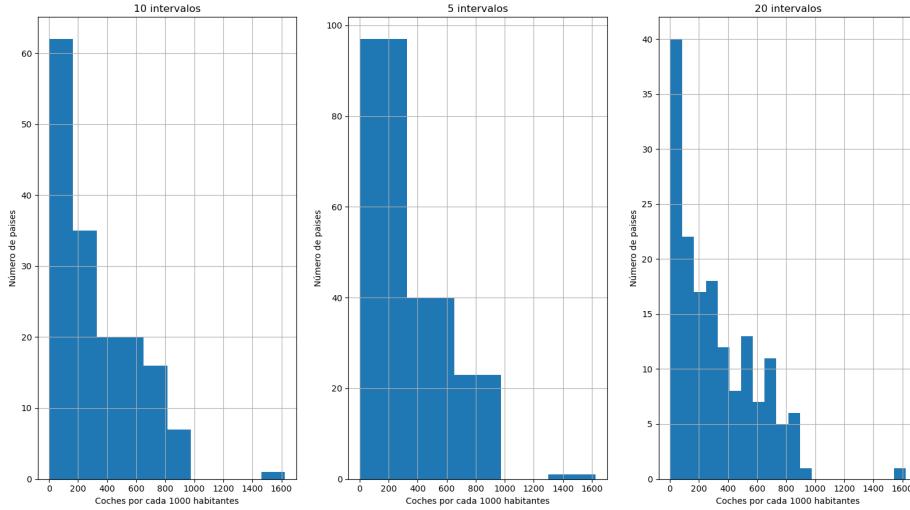


Figura 4.7: Histogramas del número de automóviles por cada 1000 habitantes

Figure 4.7:

te mediante el par clave-valor siguiente estamos indicando que queremos ejes en 3d `subplot_kw="projection": "3d"`.

```
matplotlib.pyplot.subplots(nrows=1, ncols=1, *, sharex=False,
sharey=False, squeeze=True, width_ratios=None, height_ratios=None,
subplot_kw=None, gridspec_kw=None, **fig_kw)
```

Por ejemplo, podemos representar la curva,

$$\begin{aligned}y &= \sin(2\pi x) \\z &= \cos(2\pi x)\end{aligned}$$

Para ello, seleccionamos un intervalo de valores para  $x \in (0, 2)$ , y calculamos los correspondientes valores de  $y$  y  $z$ . Los representamos usando el método `scatter`, figura 4.8.

```
import matplotlib.pyplot as plt
import numpy as np

x=np.linspace(0,2,100)
y=np.sin(2*np.pi*x)
z=np.cos(2*np.pi*x)

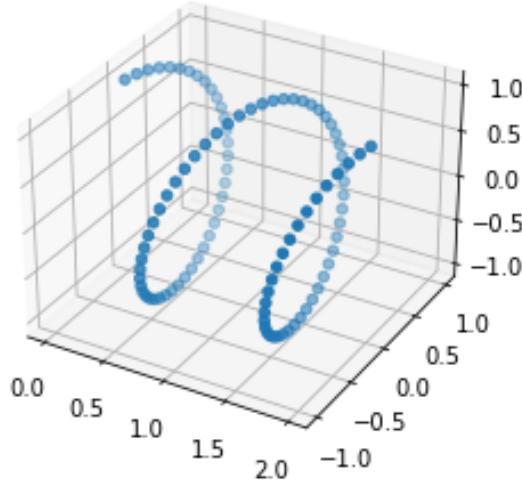
fig,ax=plt.subplots(subplot_kw={"projection": "3d"})
```

the following key-value pair, we are indicating that we want 3d axes `subplot_kw='projection': '3d'`.

For example, we can represent the curve,

$$\begin{aligned}y &= \sin(2\pi x) \\z &= \cos(2\pi x)\end{aligned}$$

To do this, we select an interval of values for  $x \in (0, 2)$ , and calculate the corresponding values of  $y$  and  $z$ . Then, we plot them using the `scatter` method, Figure 4.8.

Figura 4.8: Representación en 3 dimensiones usando *scatter*Figure 4.8: 3D plot using *scatter*

```
ax.scatter(x,y,z)
```

#### 4.2.1. Superficies

Para dibujar una superficie emplearemos el método *plot\_surface*. Igual que en el caso del método *scatter* tendremos que crear previamente una figura y unos ejes en 3 dimensiones usando *subplots* e indicando que se trata de una proyección en 3D.

Para poder dibujar una superficie con *plot\_surface* hay que generar una rejilla  $(X_m, Y_m)$  y en cada uno de los puntos de esa rejilla calcular el valor de la superficie.

Para definir dicha retícula se necesitan dos matrices. Una de ellas  $X_m$  contiene las coordenadas  $x$  de los nodos de la retícula y la otra  $Y_m$  las coordenadas  $y$ . Los elementos que ocupan la misma posición en ambas matrices, representan —juntos— un punto en el plano.

Matplotlib emplea dichas matrices como matrices de *adyacencia*. Cada nodo,  $(x_m(i, j), y_m(i, j))$ , aparecerá en la gráfica conectado por una arista a cada uno de sus cuatro puntos vecinos,  $(x_m(i-1, j), y_m(i-1, j)), (x_m(i, j-1), y_m(i, j-1)), (x_m(i+1, j), y_m(i+1, j)), (x_m(i, j+1), y_m(i, j+1))$ . Supongamos que empleamos las siguientes

To draw a surface we will use the *plot\_surface* method. As in the case of the *scatter* method, we will have to previously create a figure and axes in 3 dimensions using *subplots* and indicating that it is a 3D projection.

In order to draw a surface with *plot\_surface* we must generate a grid  $(X_m, Y_m)$  and at each of the points of this grid calculate the value of the surface.

To define such a grid, two matrices are needed. One of them  $X_m$  contains the  $x$  coordinates of the grid nodes and the other  $Y_m$  the  $y$  coordinates. Elements occupying the same position in both matrices represent - together - a point in the plane.

Matplotlib uses these matrices as adjacency matrices. Each node,  $(x_m(i, j), y_m(i, j))$ , will appear in the graph connected by an edge to each of its four neighbouring points,  $(x_m(i-1, j), y_m(i-1, j)), (x_m(i, j-1), y_m(i, j-1)), (x_m(i+1, j), y_m(i+1, j)), (x_m(i, j+1), y_m(i, j+1))$ . Suppose we use the following matrices,  $X_m$  and  $Y_m$  to define a lattice on which to draw a surface,  $z$ .

matrices,  $X_m$  y  $Y_m$  para definir una retícula sobre la que dibujar una superficie,

$$X_m = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \end{pmatrix}, Y_m = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \end{pmatrix}$$

posiciones →      nodos →

(0, 0)	—	(1, 0)	—	(2, 0)	—	(3, 0)
(0, 1)	—	(1, 1)	—	(2, 1)	—	(3, 1)
(0, 2)	—	(1, 2)	—	(2, 2)	—	(3, 2)
(0, 3)	—	(1, 3)	—	(2, 3)	—	(3, 3)

```
Axes3D.plot_surface(X, Y, Z, *, norm=None, vmin=None, vmax=None,
                     lightsource=None, **kwargs)
```

La estructura de las matrices  $X_m$  e  $Y_m$  del ejemplo anterior, es la típica de las matrices de adyacencia de una retícula cuadrada; la matriz  $X_m$  tiene la filas repetidas y la matriz  $Y_m$  tiene repetidas las columnas. En el ejemplo las matrices son cuadradas y definen una retícula de  $4 \times 4$  nodos. En general, podemos definir una retícula rectangular de  $m \times n$  nodos. En este caso las matrices empleadas para definir la retícula tendrían dimensión  $m \times n$ .

Para dibujar con Matplotlib superficies podemos en primer lugar definir la retícula a partir de dos vectores de coordenadas empleando el método de numpy `meshgrid`. En el ejemplo que acabamos de ver, hemos empleado una retícula que cubre el intervalo,  $x \in [0, 3]$  e  $y \in [0, 3]$ . para definirlo creamos los vectores,

```
x=np.arange(0,4)

x
Out[23]: array([0, 1, 2, 3])

y=np.arange(0,4)

y
Out[25]: array([0, 1, 2, 3])
```

A continuación empleamos el método `meshgrid` para construir las dos matrices de adyacencia. Numpy se encargará de repetir las filas y columnas necesarias,

```
[Xm,Ym]=np.meshgrid(x,y)
Xm
Out[20]:
array([[0, 1, 2, 3],
```

The structure of the  $X_m$  and  $Y_m$  matrices in the previous example is typical of the adjacency matrices of a square lattice; the  $X_m$  matrix has repeated rows and the  $Y_m$  matrix has repeated columns. In the example the matrices are square and define a lattice of  $4 \times 4$  nodes. In general, we can define a rectangular lattice of  $m \times n$  nodes. In this case the matrices used to define the lattice would have dimension  $m \times n$ .

To draw with Matplotlib surfaces we can first define the grid from two coordinate vectors using the numpy method `meshgrid`. In the example we have just seen, we have used a grid covering the interval,  $x \in [0, 3]$  and  $y \in [0, 3]$ . To define it we create the vectors,

```
x=np.arange(0,4)

x
Out[23]: array([0, 1, 2, 3])

y=np.arange(0,4)

y
Out[25]: array([0, 1, 2, 3])
```

Next, we use the method `meshgrid` to construct the two adjacency matrices. Numpy will take care of repeating the necessary rows and columns,

```
[Xm,Ym]=np.meshgrid(x,y)
Xm
Out[20]:
array([[0, 1, 2, 3],
       [0, 1, 2, 3],
```

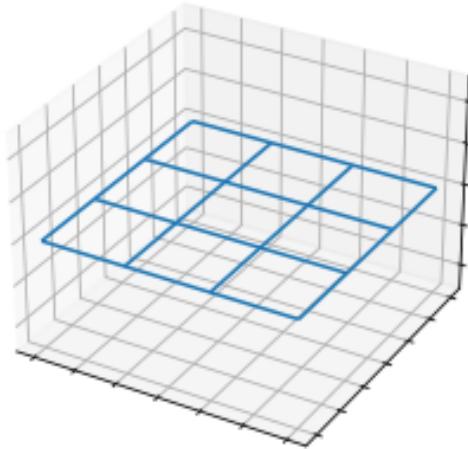


Figura 4.9: Retícula plana

Figure 4.9: Flat grid

```
[0, 1, 2, 3],  
[0, 1, 2, 3],  
[0, 1, 2, 3])  
  
Ym  
Out[21]:  
array([[0, 0, 0, 0],  
       [1, 1, 1, 1],  
       [2, 2, 2, 2],  
       [3, 3, 3, 3]])
```

Una vez construidas las matrices de adyacencia, solo necesitamos una matriz de valores para  $Z_m$ . Si definimos por ejemplo,

```
Zm=np.zeros(np.shape(Xm))
```

```
Zm  
Out[35]:  
array([[0., 0., 0., 0.],  
       [0., 0., 0., 0.],  
       [0., 0., 0., 0.],  
       [0., 0., 0., 0.]])
```

Podríamos representar la retícula plana de la figura 4.9, empleando por ejemplo el comando `plot_wireframe(Xm, Ym, Zm)`.

Una vez que hemos visto como construir

```
[0, 1, 2, 3],  
[0, 1, 2, 3]))  
  
Ym  
Out[21]:  
array([[0, 0, 0, 0],  
       [1, 1, 1, 1],  
       [2, 2, 2, 2],  
       [3, 3, 3, 3]])
```

Once the adjacency matrices are constructed, we only need a matrix of values for  $Z_m$ . If we define for example,

```
Zm=np.zeros(np.shape(Xm))
```

```
Zm  
Out[35]:  
array([[0., 0., 0., 0.],  
       [0., 0., 0., 0.],  
       [0., 0., 0., 0.],  
       [0., 0., 0., 0.]])
```

We could represent the flat grid of the figure 4.9, using for example the command `plot_wireframe(Xm, Ym, Zm)`.

Now that we have seen how to construct

una retícula rectangular sobre la que construir una superficie, veamos como dibujarla con un ejemplo. Supongamos que queremos dibujar la superficie,

$$z = x^3 + y^2$$

En la región del plano,  $x \in [-1.5, 1.5]$ ,  $y \in [-2, 2]$ .

Igual que en el ejemplo inicial, lo primero que debemos hacer es construirnos una matrices de adyacencia que definan una retícula en la región de interés,

```
x=np.linspace(-1.5,1.5,25)
y=np.linspace(-2,2,50)
[Xm,Ym]=np.meshgrid(x,y)
```

Es interesante notar que la región de interés no es cuadrada y que las matrices de adyacencia tampoco los son ( $50 \times 25$ ). Además los puntos no están espaciados igual en los dos ejes.

A continuación obtenemos la matriz de coordenadas z, aplicando la función a los puntos de la retícula,

```
Zm=Xm**3+Ym**2
```

Podemos representar la superficie usando el método *wireframe* o el método *surface*. En el primer caso se dibuja la superficie como una rejilla, figura 4.10(a) y en el segundo como las caras rellenas de color de la retícula, figura 4.10(b).

a rectangular grid on which to build a surface, let's see how to draw it with an example. Suppose we want to draw the surface,

$$z = x^3 + y^2$$

In the region of the plane,  $x \in [-1.5, 1.5]$ ,  $y \in [-2, 2]$ .

As in the initial example, the first thing to do is to construct adjacency matrices that define a grid in the region of interest,

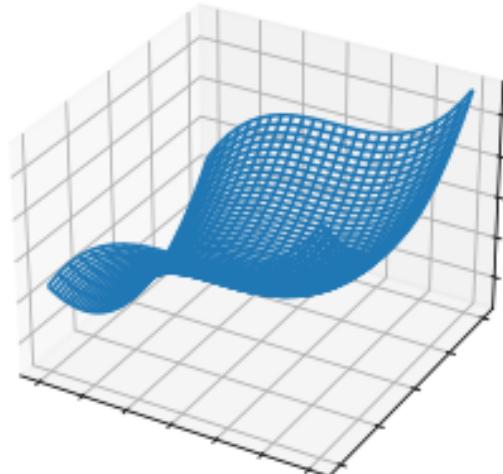
```
x=np.linspace(-1.5,1.5,25)
y=np.linspace(-2,2,50)
[Xm,Ym]=np.meshgrid(x,y)
```

It is interesting to note that the region of interest is not square and that the adjacency matrices are not square either ( $50 \times 25$ ). Also the points are not equally spaced on the two axes.

Next we obtain the z-coordinate matrix by applying the function to the grid points,

```
Zm=Xm**3+Ym**2
```

We can represent the surface using the *wireframe* method or the *surface* method. In the first case the surface is drawn as a grid, figure 4.10(a), and in the second as the colour-filled faces of the grid, figure 4.10(b).



(a) Función  $z = x^3 + y^2$  representada con *wireframe*

## 4.3. Animaciones

Matplotlib también proporciona una interfaz para generar animaciones utilizando el módulo *animation*. Una animación es una secuencia de fotogramas en la que cada fotograma corresponde a un trazado sobre una figura.

Para crear una animación en python usando *matplotlib.animation* seguiremos los siguientes pasos:

1. Inicializar los meta datos para la animación.
2. Inicializar el fondo de la animación.
3. Definir los objetos que van a ir cambiando en cada fotograma.

**Inicializar los metadatos** . Inicializar los meta datos para la animación creando un diccionario con ellos y pasándoselos al método que va a crear la animación *writers*. Le indicamos tanto los metadatos como el número de frames por segundo con el parámetro *fps*.

```
FFMpegWriter=manimation.writers["ffmpeg"]
metadata = dict(title="GIF Test", artist ="Me",
comment="A red circle following a blue sine wave")
writer = FFMpegWriter(fps=15, metadata=metadata)
```

**Inicializar el fondo de la animación** , es decir aquellos elementos que no van a cambiar. Lo haremos definiendo una figura y todo aquello que va a permanecer constante durante la animación, por eso lo llamamos el fondo. En este caso es la curva del seno. También inicializamos el punto vacío rojo, al que le damos unas coordenadas vacías, para que Python sepa que esas coordenadas se actualizarán posteriormente. Como las etiquetas no cambian también las ponemos aquí.

```
fig =plt.figure()
n=1000
x=np.linspace(0, 6*np.pi,n)
y=np.sin(x)
sine_line, =plt.plot(x,y,"b")
red_circle,=plt.plot([],[],"ro",markersize=10)
```

## 4.3. Animations

Based on its plotting functionality, Matplotlib also provides an interface to generate animations using the *animation* module. An animation is a sequence of frames where each frame corresponds to a plot on a figure.

To create an animation in python using *matplotlib.animation* we will follow the following steps:

1. Initialise the meta data for the animation.
2. Initialise the background of the animation.
3. Define the objects that are going to change in each frame.

**Initialize the meta data.** Initialise the meta data for the animation by creating a dictionary of it and passing it to the method that will create the animation *writers*. We indicate both the metadata and the number of frames per second with the parameter *fps*.

**Initialise the background of the animation** , i.e. those elements that are not going to change. We will do this by defining a figure and everything that will remain constant during the animation, which is why we call it the background. In this case it is the sine curve. We also initialise the red empty point, to which we give empty coordinates, so that Python knows that these coordinates will be updated later. Since the labels don't change, we also put them here.

```
plt.xlabel("x")
plt.ylabel("sin(x)")
```

**Definir los objetos que van a ir cambiando en cada fotograma**. Salvaremos un fichero, indicando qué figura vamos a salvar, el nombre del fichero (`movie_test.mpg`) y la resolución de la figura en puntos por pulgada (dpi). En el bucle `for` se actualiza repetidamente la figura para crear el movimiento, en cada iteración se actualiza la posición del punto rojo (usando `set_data`). El método `grab_frame` captura estos cambios y los muestra con los frames por segundo especificados anteriormente.

```
with writer.saving(fig, "movie_test.mp4", 100):
    for i in range(n):
        x0=x[i]
        y0=y[i]
        red_circle.set_data([x0],[y0])
        writer.grab_frame()
```

**Define the objects that are going to change in each frame.** We will save a file, indicating which figure we are going to save, the name of the file (`movie_test.mpg`) and the resolution of the figure in dots per inch (dpi). In the `for` loop the figure is repeatedly updated to create the movement, in each iteration the position of the red dot is updated (using `set_data`). The `grab_frame` method captures these changes and displays them at the frames per second specified above.

## 4.4. Ejercicios

1. 1. Escribe una función que muestre paso a paso en una figura la trayectoria del tiro parabólico para  $v_0 = 40$  (m/s)  $\theta = 45$  grados y para  $v_0 = 60$  (m/s)  $\theta = 60$  grados. En cada instante se mostrará con un círculo rojo la posición ( $x$ ,  $y$ ).

$$x = v_0 \cdot \cos(\theta) \cdot t$$

$$y = v_0 \cdot \sin(\theta) \cdot t - \frac{1}{2} \cdot g \cdot t^2$$

Añade el nombre de cada eje y el título de la figura.

2. Usa únicamente una sola figura para representar en diferentes paneles las siguientes funciones.

- $y = e^{-x^2}$  en el intervalo  $[-2, 2]$  empleando 100 datos equiespaciados
- $y = e^{-((\frac{x}{2})^2)}$
- $y = e^{-(2x)^2}$
- La función  $y = e^{-(2x)^2}$  pero en escala logarítmica

3. Crear una función *dibujatriangul* que tenga como argumento de entrada los 3 vértices de un triángulo y devuelva una figura con el triángulo dibujado.
4. Crea una función *dibujaMUC* que dibuje la trayectoria circular de una partícula conociendo su velocidad angular  $\omega$  y el radio de giro  $r$ . Las ecuaciones cartesianas son:

$$x = r \cos(\omega \cdot t)$$

$$y = r \sin(\omega \cdot t)$$

- Añadir a la trayectoria en cada punto el vector velocidad con el comando *quiver* mediante las ecuaciones de la velocidad:

$$v_x = -r \cdot \omega \cdot \sin(\omega \cdot t)$$

$$v_y = r \cdot \omega \cdot \cos(\omega \cdot t)$$

5. Crea una función que tenga como parámetros de entrada el radio  $r$  y la velocidad angular  $\omega$  y dibuje la trayectoria en 3 dimensiones de una partícula que describe un MCU en el plano XY moviéndose en el eje z con una velocidad constante  $v_z$ .
6. Crea una función que represente el tiro parabólico en 3D sabiendo que la función tiene como entrada los ángulos  $\psi$  y  $\theta$ , de acuerdo con las siguientes expresiones:

$$x = v_0 \cdot \cos(\psi) \cdot \cos(\theta) \cdot t$$

$$y = v_0 \cdot \sin(\psi) \cdot \cos(\theta) \cdot t$$

$$z = v_0 \cdot \sin(\theta) \cdot t - \frac{1}{2} \cdot g \cdot t^2$$

7. Usando el Comando *meshgrid*, crea una retícula cuadrada en el intervalo  $x = [-11]$  e  $y = [-22]$  emplea para ello un paso de malla de valor 0.1. Crea una matriz de ceros del tamaño de las matrices que definen la retícula. Representa, empleando el comando *mesh*, la matriz de ceros creada sobre la retícula. Representa gráficamente la superficie  $z = e^{-(x^2+y^2)}$ .

8. Genera 1000 números aleatorios distribuidos normalmente (usa para ello la función `np.random.randn`). Usa la función `plt.hist` para dibujar el histograma de los números aleatorios generados.
- Sepáralos en 10 contenedores.
  - Crea un diagrama de barras de los datos del apartado anterior usando `plt.bar`.
  - ¿Crees que la función `np.random.randn` es una buena aproximación de una distribución normal?
9. Crea una función `my_poly_plot(n,x)` que dibuje los polinomios  $p(x) = x^k$  para  $k = 1, \dots, n$ .
10. Supón que tienes tres puntos que forman las esquinas de un triángulo equilátero  $P_1 = (0, 0)$ ,  $P_2 = (0.5, \frac{\sqrt{2}}{2})$  y  $P_3 = (1, 0)$ . Crea una función que:
- Genere un conjunto de  $n$  puntos  $p_i = (x_i, y_i)$  tales que  $p_1 = (0, 0)$  y  $p_{i+1}$  es el punto medio entre  $p_i$  y  $P_1$  con un 33 % de probabilidad, el punto medio entre  $p_i$  y  $P_2$  con un 33 % de probabilidad y el punto medio entre  $p_i$  y  $P_3$  con un 33 % de probabilidad.
  - Dibuje los puntos obtenidos en el plano.
  - Prueba la función para  $n = 100$  y  $n = 1000$  puntos.

## 4.5. Test del curso 2020/21

1. El movimiento de un cuerpo en el plano viene descrito por las siguientes ecuaciones,

$$x(t) = \sin(2\pi \omega_1 t) \quad (4.1)$$

$$y(t) = \cos(2\pi \omega_2 t) \quad (4.2)$$

$$z(t) = a \cos(2\pi \omega_1 t) \quad (4.3)$$

donde  $x, y, z$  representan las coordenadas del cuerpo en el instante de tiempo  $t$  medidas en metros,  $\omega_1$  y  $\omega_2$  son frecuencias fijas medidas en  $\text{rad} \cdot \text{s}^{-1}$  y  $a$  es una amplitud fija medida en metros.

a) (1.5 puntos) Escribe una función que:

- 1) Tome como variables de entrada las frecuencias  $\omega_1$ ,  $\omega_2$ , la amplitud  $a$ .
  - 2) Haga el cálculo de la trayectoria en tres dimensiones descrita por el cuerpo en el intervalo de tiempo  $[0, 1]$ . Considera incrementos de tiempo de  $0.01s$ .
  - 3) Dibuje en una gráfica la trayectoria descrita por el cuerpo en el espacio. Cada eje del gráfico deberá llevar una etiqueta indicando de qué variable se trata  $x$ ,  $y$  ó  $z$ .
  - 4) La salida serán tres vectores con los valores de  $x$ ,  $y$  y  $z$  calculados para cada instante de tiempo.
- b) (1.5 puntos) Añade a la función creada en el apartado anterior el código necesario para que:
- 1) **Solo si** se le dan dos variables de entrada en lugar de tres, entonces calcule la trayectoria que describiría el cuerpo en el plano si se eliminara la ecuación (4.3) de las ecuaciones del sistema. (NOTA: La función debe seguir funcionando igual que en el apartado anterior si se le dan tres entradas. Emplea el comando  `nargin`).

- 2) El resultado deberá también representarse gráficamente, pero solo en dos dimensiones, y devolver la variable  $z$  como un vector vacío.
- c) **(1.5 puntos)** Genera dos vectores de frecuencias:  $W1$  y  $W2$  de modo que el primero contenga los números impares comprendidos entre 1 y 7 y el segundo los números pares comprendidos entre 2 y 8. Emplea la función creada en el apartado anterior para calcular el valor de las trayectorias obtenidas tomando como entradas para  $\omega_1, \omega_2$ , todos los pares posibles de la forma:  $W1(i)$  y  $W2(j)$ . Supón que no hay tercera entrada  $a$ . Realiza los cálculos empleando bucles.
- d) **(1 punto)** Añade a tu programa el código necesario para que dibuje cada resultado en un *subplot* de modo que la figura resultante tenga  $4 \times 4$  *subplots* (ver figura al dorso). Debes crear los *subplots* empleando los mismos bucles del apartados anterior.
- e) **(0.5 puntos)** Repite los cálculos de los apartados c) y d) pero tomando ahora  $a = 1$ .
2. Una matriz cuadrada  $A$ , de dimensión arbitraria ( $n \times n$ ), cuyos elementos  $a_{ij} \in \mathbb{Z}$ , se define como *buenrollista* cuando la suma de los valores pares de cada fila es mayor o igual que la suma de los valores impares de la columna correspondiente, i.e., satisface la siguiente relación:

$$\sum_{j=1}^n a_{ij(\text{par})} \geq \sum_{j=1}^n a_{ji(\text{impar})}, \quad \forall i \in \{1, \dots, n\} \quad (4.4)$$

- a) **(1.5 puntos)** Escribe una función que tome como variable de entrada una matriz cuadrada de cualquier dimensión y calcule un vector con las sumas de los valores pares de los elementos de cada una de su filas y otro vector con las sumas de los elementos impares de cada una de sus columnas.
- b) **(1.5 puntos)** Añade a la función anterior el código necesario para que compruebe, empleando los vectores obtenidos en el apartado anterior, si la función es *buenrollista* y muestre un mensaje por pantalla indicando si lo es o no.
- c) **(1 punto)** Aplica la función desarrollada a la siguiente matriz:

$$\begin{pmatrix} 2 & 3 & 4 & 6 & 1 \\ 5 & 4 & -2 & 3 & 0 \\ 4 & -1 & 5 & 4 & -6 \\ 4 & 2 & 4 & 5 & 8 \\ 3 & 0 & -3 & -3 & 6 \end{pmatrix}$$


---

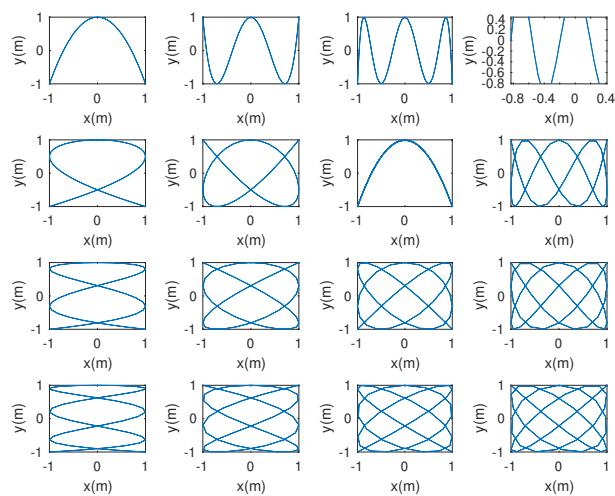


Figura 4.11: vista de la figura con los ocho subplots

## Capítulo/Chapter 5

# Aritmética del Computador y Fuentes de error

## Computer Arithmetics and Errors

“What’s the use?” a few hundred rivets chattered. “We’ve given - we’ve given, and the sooner we confess that we can’t keep the ship together and go off our little heads, the easier it will be. Not rivet forged can stand this strain”.

“No one rivet was ever meant to. Share it among you”, the Steam answered.

---

Rudyard Kipling, The Ship that Found Herself

En el capítulo 1, introdujimos la representación binaria de números así como la conversión de binario a decimal y decimal a binario. A lo largo de este capítulo vamos a profundizar más en el modo en que el ordenador representa y opera con los números así como en una de sus consecuencias inmediatas: la imprecisión de los resultados.

### 5.1. Representación binaria y decimal

Los números reales pueden representarse empleando para ello una recta que se extiende entre  $-\infty$  y  $+\infty$ . La mayoría de ellos no

In chapter one, we introduced the binary representation of numbers and the conversion from decimal to binary and from binary to decimal. In this chapter, we will get deep into how a computer represents and operates numbers. We also discuss one of its direct consequences: the imprecision of the results.

#### 5.1. Binary and denary representations

Real numbers can be represented on a line extending from  $-\infty$  to  $\infty$ . Most do not have exact numerical representations because they have infinite decimals.

admiten una representación numérica exacta, puesto que poseen infinitos decimales.

Los números enteros,  $1, -1, 2, -2, 3, -3, \dots$  admiten una representación numérica exacta. En cualquier caso, rara vez manejamos números enteros con una cantidad grande de dígitos, habitualmente, los aproximamos, expresándolos en notación científica, multiplicándolos por una potencia de 10. Tomemos un ejemplo de la química: la cantidad de átomos o moléculas que constituyen un mol de una sustancia se expresa por el número de Avogadro,  $N_A = 6.02214179 \times 10^{23}$ , dicho numero, —que debería ser un entero— se expresa empleando tan solo 9 de sus 23 cifras significativas (a veces tan solo se dan las tres primeras).

Cuando truncamos un número, es decir, despreciamos todos sus dígitos hacia la derecha a partir de uno dado, estamos approximando dicho número por un número racional, es decir, por un número que puede expresarse como el cociente entre dos números enteros.  $1/2, 3/4, \dots$

Algunos números racionales se reducen a su vez a números enteros,  $6/3$ , otros dan lugar a números cuya representación exige un número finito de dígitos:

$$11/2 = (5.5)_{10} = 5 \times 10^0 + 5 \times 10^{-1}$$

Si representamos el mismo número en base 2 obtenemos,

$$11/2 = (101.1)_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1}$$

Sin embargo, el que un número racional admita una representación finita, depende de la base en que se representa. Por ejemplo:  $1/10 = (0.1)_{10}$  no admite representación finita en base 2.

$$1/10 = (0.0001100110011\dots)_2 = 1 \times 2^{-4} + 1 \times 2^{-5} + 0 \times 2^{-6} + 0 \times 2^{-7} + 1 \times 2^{-8} + \dots$$

En este último caso, se trata de una representación que, aunque no termina, es repetitiva o periódica; tras el primer cero decimal se repite indefinidamente la secuencia: 0011. Los números racionales admiten siempre una representación finita ó infinita periódica.

El número racional  $1/3 = (0.333\dots)_{10} =$

The integer numbers  $1 - 1, 2, -2, 3 - 3, \dots$  can be represented in scientific notation. However, we rarely deal with numbers containing a large amount of digits. Usually, we take an approximate value, written then in scientific notation, by multiplying the number by a power of 10. Let's take an example from chemistry: the number of atoms or molecules that make up a mole of a chemical substance is defined as Avogadro's number  $N_A = 6.02214179 \times 10^{23}$ . This number should be an integer, but we represent it in scientific notation given just a few, nine in our example, of its 23 digits.

We truncate a number when we miss off digits past a certain point in the number. Then, we approximate the truncated number by a rational number, i.e., a number that the quotient of two integers can represent.  $1/2, 3/4, \dots$

Some fractional numbers reduce, in turn, to integer numbers  $6/3$ . Others are numbers that need an infinity number of digits to be represented:

If we represent the same number in base 2 we obtain,

However, A rational number may or may not have a finite representation, depending on the base used to represent it. For example,  $1/10 = (0.1)_{10}$  has no finite representation in base 2.

For this case, the number representation is infinite but repetitive or periodic; following the first zero after the decimal point, the sequence 0011 repeats endlessly. Rational numbers always have a finite representation or a periodic infinite one.

$(0.010101\cdots)_2$  solo admite representación infinita periódica tanto en base 10 como en base 2. Sin embargo, admite representación finita si lo representamos en base 3:  $1/3 = (0.1)_3 = 0 \times 3^0 + 1 \times 3^{-1}$ .

Habitualmente, en la vida ordinaria, representamos los números en base 10, sin embargo, como hemos visto en el capítulo 1 el ordenador emplea una representación binaria. Como acabamos de ver, una primera consecuencia de esta diferencia, es que al pasar números de una representación a otra estemos alterando la precisión con la que manejamos dichos números. Esta diferencia de representación supone ya una primera fuente de errores, que es preciso tener en cuenta cuando lo que se pretende es hacer cálculo científico con un ordenador. Como iremos viendo a lo largo de este capítulo, no será la única.

## 5.2. Representación de números en el ordenador

**Enteros positivos** La forma más fácil de representar números sería empleando directamente los registros de memoria para guardar números en binario. Si suponemos un ordenador que tenga registros de 16 bits. Podríamos guardar en ellos números comprendidos entre el 0 y el  $2^{16} - 1 = 65535$ . Este último se representaría con un 1 en cada una de las posiciones del registro, en total 16 unos. Se trata del entero más grande que podríamos representar con un registro de 16 bits:

posición	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
valor	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Hasta ahora, nuestro sistema de representación solo admite números enteros positivos. El menor número que podemos representar es el cero, y el mayor —dependerá del tamaño  $n$  de los registros del ordenador—,  $2^n - 1$ . Es evidente que este sistema de representación resulta insuficiente.

**Enteros positivos y negativos.** Un primera mejora consistiría en ampliar el sistema de representación para que admita también números negativos. Una posibilidad sería

The rational number  $1/3 = (0.333\cdots)_{10} = (0.010101\cdots)_2$  has a periodic infinite representation both in basis ten and in basis 2. However, it has a finite representation in base 3:  $1/3 = (0.1)_3 = 0 \times 3^0 + 1 \times 3^{-1}$ .

Usually, in ordinary life, we represent the number using base-ten. However, as we saw in chapter 1, computers use a binary representation. The first consequence of this difference is that we alter number precision when we change them from one representation to another. Thus, this representation difference is, in fact, the first cause of errors that we must take into account when we want to use a computer to perform scientific computing. In this chapter, we will find more causes.

## 5.2. Computer number representation

**positive integers.** The most straightforward method for representing numbers would be using the memory registers directly to store the numbers in binary format. Considering a computer with 16-bit size registers, we can save numbers between 0 and  $2^{16} - 1 = 65535$ . This maximum number would be represented by filling each register position with a 1, so 16 ones in total. This number is the largest one we can represent using 16 bits:

Our current representation system only accepts positive integer numbers. The lowest number we can represent is zero, and the largest one —which will depend on the computer register size—, would be  $2^n - 1$ . It's clear that this representation system is insufficient.

**Positive and negative integers.** A first improvement would be to expand the system so that it also admits negative numbers. One option would be to reserve a bit of the register to represent the sign, using the remaining re-

reservar uno de los bits del array para representar el signo , y usar los restantes para representar el número. En nuestro ejemplo de un registro de 16 bits, el número más grande sería ahora  $2^{15} - 1$  y el más pequeño sería  $-2^{15} + 1$ . El cero tendría una doble representación, una de ellas con signo más y la otra con signo menos.

Una representación alternativa, cuyo uso está muy extendido en las máquinas, es la representación conocida con el nombre de *Complemento a 2* . Supongamos un registro de n bits:

- Un entero  $x$  no negativo se almacena empleando directamente su representación binaria. el intervalo de enteros no negativos representables es:  $0 \leq x \leq 2^{(n-1)} - 1$ .
- Un entero negativo  $-x$ , donde  $1 \leq x \leq 2^{(n-1)}$ , se almacena como la representación binaria del entero positivo:  $2^n - x$
- El número así construido, recibe el nombre de complemento a 2 de x.

En general, dado un número entero  $N < 2^n$  su complemento a dos en una representación binaria de  $n$  dígitos se define como:

$$C_2^N = 2^n - N$$

En la tabla 5.1 se muestra un ejemplo de esta representación, para el caso de un registro de 4 bits:

La representación en complemento a 2 tiene las siguientes características:

- La representación de los números positivos coincide con su valor binario
- La representación del número 0 es única.
- El rango de valores representables en una representación de N bits es  $-2^{N-1} \leq x \leq 2^{N-1} - 1$ .
- Los números negativos se obtienen como el complemento a dos, cambiado de signo, del valor binario que los representa.

gister to represent the (absolute value of the) number. Returning to our example of a 16-bit register, the largest number representable would now be  $2^{15} - 1$  and the lowest  $-2^{15} + 1$ . Zero would have a double representation, one with a plus sign and another with a minus sign.

An alternative representation, frequently used by computers, is the *2's complement* representation. We will describe the representation for a generic  $n$ -bits register:

- A non-negative integer is stored using directly its binary representation. The range of representable non-negative numbers is  $0 \leq x \leq 2^{(n-1)} - 1$ .
- A negative integer  $-x$  where  $1 \leq x \leq 2^{(n-1)}$ , is stored as the binary representation of the positive integer  $2^n - x$ .
- The number built following the procedure just described is called the 2's complement of  $x$ .

In general, we define the 2's complement of an integer  $N < 2^n$  in a binary representation of  $n$  digits as:

$$C_2^N = 2^n - N$$

Table 5.1 shows an example of the 2's complement representation for a 4-bit register.

2's complement representation has the following properties:

- For a positive integer, its 2's complement representation fits its standard binary representation.
- We get a single representation for the number zero.
- The range of representable values in an N-bit representation is  $-2^{N-1} \leq x \leq 2^{N-1} - 1$ .
- We obtain the negative number by computing the 2's complement of their bi-

Decimal	(4 bits)	$C_2 = 2^n - x$	n. represent-ado-ed
15	1111	$(16 - 1) = 15$	-1
14	1110	$(16 - 2) = 14$	-2
13	1101	$(16 - 3) = 13$	-3
12	1100	$(16 - 4) = 12$	-4
11	1011	$(16 - 5) = 11$	-5
10	1010	$(16 - 6) = 10$	-6
9	1001	$(16 - 7) = 9$	-7
8	1000	$(16 - 8) = 8$	-8
<hr/>			
7	0111		7
6	0110		6
5	0101		5
4	0100		4
3	0011		3
2	0010		2
1	0001		1
0	0000		0

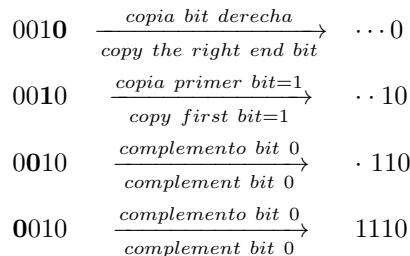
Tabla 5.1: Representación en complemento a dos para un registro de 4 bits

Table 5.1: 2's complement representation for a 4-bit register

Una forma práctica de obtener el complemento a dos de un número binario es copiar el número original de derecha a izquierda hasta que aparezca el primer 1. Luego de copiar el primer 1, se complementan el resto de los bits del número (si es 0 se pone un 1 y si es 1 se pone un 0). Por ejemplo el complemento a dos del número 2, (0010 en una representación de 4 bits) sería 1110,

nary representation and changing the sign of the result

A practical method for computing the 2's complement of a binary number is to copy the number from right to left till we reach the first 1. Once we have copied the first 1, we *complement* the remaining number bits. This means changing the zeros to ones and the ones to zeros. For instance, the 2'complemet of the number 2 (0010 in a -bit representation) will be 1110,



Una propiedad importante de la representación de complemento a dos, es que la diferencia de dos números puede realizarse directamente sumando al primero el complemento a dos del segundo. Para verlo podemos usar los dos números del ejemplo anterior: 0010 es

An important property of 2's complement representation is that we can compute the difference between two numbers just by adding the first number to the 2's complement of the second one. We can see it using the two numbers of the previous example: 0010 is the bi-

la representación binaria del número 2, si empleamos un registro de cuatro bits. Su complemento a dos, 1110 representa al número  $-2$ . Si los sumamos <sup>1</sup>:

<sup>1</sup>Sumar en binario es como sumar en decimal. Se procede de bit a bit, de derecha a izquierda y cuando se suman 1 y 1, el resultado es 10 (base 2), de modo que el bit resultante se coloca en 0 y se lleva el 1 hacia el siguiente bit de la izquierda.

nary representation of the number 2 if we use a 4-bit representation. It 2's complement, 1110 represent the number  $-2$ . If we add them <sup>1</sup>:

<sup>1</sup>The addition of binary numbers is like the addition of decimal numbers. We add then, bitwise, from right to left, and when we add 1 and 1, the result is 10 (base 2). So, the resulting bit is set to zero, and we carry on the 1 to the next left bit.

	0	0	1	0
	1	1	1	0
<b>1</b>	0	0	0	0
	0	0	0	0

El resultado de la suma nos da un número cuyos primeros cuatro bits son 0. El bit distinto de cero, no puede ser almacenado en un registro de cuatro bits, se dice que el resultado de la operación *desborda* el tamaño del registro. Ese bit que no puede almacenarse se descarta al realizar la operación de suma, con lo que el resultado sería cero, como corresponde a la suma de  $2 + (-2)$ . Esta es la motivación fundamental para emplear una representación en complemento a dos: No hace falta emplear circuitos especiales para calcular la diferencia entre dos números, ya que ésta se representa como la suma del minuendo con el complemento a dos del sustraendo.

### 5.2.1. Números no enteros: Representación en punto fijo y en punto flotante

La representación de números no enteros se emplea para representar de modo aproximado números reales. Como hemos dicho antes, los números racionales periódicos y los números irracionales no pueden representarse de forma exacta mediante un número finito de decimales. Por esta razón, su representación es siempre aproximada. Para los números racionales no periódicos, la representación será exacta o aproximada dependiendo del número que se trate, el tamaño de los registros del ordenador y el sistema empleado para la representación. Los dos sistemas de representación más conocidos son la representación en

The result of the addition yields a number with its first four bits as 0. The bit with value one cannot be accommodated in a 4-bit register. We refer to the result as an *overflow* of the register size. This bit that can not be stored is discarded after the addition. Thus, the result will be zero, as can be expected after adding  $2 + (-2)$ . This is a fundamental motivation to use the 2's complement representation: we do not need any additional special circuits to compute the difference between two numbers because we can represent it straightforwardly by adding the minuend plus the 2's complement of the subtrahend.

### 5.2.1. Non-integer numbers: fixed-point and floating-point representations.

We use the non-integer representation to approximately represent real numbers. As we have seen before, periodic rational numbers and irrational numbers cannot be exactly represented using a finite number of decimals. This is the reason why their representation is always an approximation. In the case of non-periodic rational numbers, the representation will be exact or approximated depending on the number itself, the size of the computer registers and the representation system we use. The two most common representation systems are the fixed-point and the floating-point representations.

punto fijo y, de especial interés para el cálculo científico, la representación en punto flotante.

**Representación en punto fijo.** En la representación en punto fijo, el registro empleado para almacenar un número se divide en tres partes:

- 1 bit para almacenar el signo del número.
- Un campo de bits de tamaño fijo para representar la parte entera del número.
- Un campo para almacenar la parte decimal del número

Por ejemplo, si tenemos un registro de 16 bits podemos reservar 1 bit para el signo, 7 para la parte entera y 8 para la parte decimal. La representación en punto fijo del número binario  $-10111.0011101101$  sería:

s	p. entera/ integer p.							p. decimal/ decimal p.						
1	0	0	1	0	1	1	1	0	0	1	1	1	0	1

Es interesante notar cómo para representar la parte entera, nos sobran dos bits en el registro, ya que la parte entera solo tiene cinco cifras y tenemos 7 bits para representarla. Sin embargo, la parte decimal tiene 10 cifras; como solo tenemos 8 bits para representar la parte decimal, la representación truncará el número eliminando los dos últimos decimales.

Si asociamos cada bit con una potencia de dos, en orden creciente de derecha a izquierda, podemos convertir directamente el número representado de binario a decimal,

**fixed-point representation.** To represent a number using a fixed-point representation, we divide the register into three parts:

- We set aside one bit to store the number sign.
- We represent the integer part of the number using a fixed-size field of bits.
- We represent the decimal part of the number using also a fixed-size field of bits.

For instance, if we are using a 16-bit register we can take a bit to represent the number sign, seven bits for the integer part and eight bits for the decimal part. following this register sharing, the representation of the number  $-10111.0011101101$  would be:

It is interesting to notice how we have two extra bits to represent the integer part of the number because the integer part has only five digits and we have seven bits to represent it. However, to represent the decimal part of the number, we have ten digits and only eight bits to represent it. So, the representation truncates the number cutting out the last two decimals.

If we associate each bit with a power of two, in increasing order from right to left, we can straightforwardly convert the number from its binary representation to its decimal form.

s	p. entera / integer p.							p. decimal / decimal p.						
1	0	0	1	0	1	1	1	0	0	1	1	1	0	1
-	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	$2^{-1}$	$2^{-2}$	$2^{-3}$	$2^{-4}$	$2^{-5}$	$2^{-6}$	$2^{-7}$

Por tanto el número representado sería,

$$-(0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} + 1 \cdot 2^{-5} + 0 \cdot 2^{-6} + 1 \cdot 2^{-7} + 1 \cdot -8) = -23.23046875$$

De modo análogo podemos calcular cual sería el número más grande representable,

Then, the number represented will be,

Similarly, we can calculate the largest representable number,

s	p. entera / integer p.							p. decimal/decimal p.							
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
0	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	$2^{-1}$	$2^{-2}$	$2^{-3}$	$2^{-4}$	$2^{-5}$	$2^{-6}$	$2^{-7}$	$2^{-8}$

$$1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} + \\ + 1 \cdot 2^{-5} + 1 \cdot 2^{-6} + 1 \cdot 2^{-7} + 1 \cdot 2^{-8} = 127.99609375$$

El número menor representable,

The lowest representable number,

s	p. entera / integer p.							p. decimal/ decimal p.							
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
-	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	$2^{-1}$	$2^{-2}$	$2^{-3}$	$2^{-4}$	$2^{-5}$	$2^{-6}$	$2^{-7}$	$2^{-8}$

$$-(1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} + \\ + 1 \cdot 2^{-5} + 1 \cdot 2^{-6} + 1 \cdot 2^{-7} + 1 \cdot 2^{-8}) = -127.99609375$$

El número entero más grande,

The largest integer,

s	p. entera / integer p.							p. decimal/ decimal p.							
	0	1	1	1	1	1	1	0	0	0	0	0	0	0	
0	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	$2^{-1}$	$2^{-2}$	$2^{-3}$	$2^{-4}$	$2^{-5}$	$2^{-6}$	$2^{-7}$	$2^{-8}$

$$1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} + 0 \cdot 2^{-4} + \\ + 0 \cdot 2^{-5} + 0 \cdot 2^{-6} + 0 \cdot 2^{-7} + 0 \cdot 2^{-8} = 127$$

El número decimal positivo más próximo  
a 1,

The positive decimal number closest to 1,

s	p. entera / integer p.							p. decimal/ decimal p.							
	0	0	0	0	0	0	0	1	1	1	1	1	1	1	
0	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	$2^{-1}$	$2^{-2}$	$2^{-3}$	$2^{-4}$	$2^{-5}$	$2^{-6}$	$2^{-7}$	$2^{-8}$

$$0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} + \\ + 1 \cdot 2^{-5} + 1 \cdot 2^{-6} + 1 \cdot 2^{-7} + 1 \cdot 2^{-8} = 0.99609375$$

O el número decimal positivo más próximo  
a 0,Or the positive decimal number closest to  
0,

s	p. entera / integer p.							p. decimal/ decimal p.							
	0	0	0	0	0	0	0	0	0	0	0	0	0	1	
0	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	$2^{-1}$	$2^{-2}$	$2^{-3}$	$2^{-4}$	$2^{-5}$	$2^{-6}$	$2^{-7}$	$2^{-8}$

$$0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} + 0 \cdot 2^{-4} + \\ + 0 \cdot 2^{-5} + 0 \cdot 2^{-6} + 0 \cdot 2^{-7} + 1 \cdot 2^{-8} = 0.00390625$$

La representación en punto fijo fue la más usada en los primeros computadores. Sin embargo, es posible con el mismo tamaño de registro representar un espectro mucho más amplio de números empleando la representación en punto flotante.

**Representación en punto flotante.** La representación de números en el ordenador en formato de punto flotante, es similar a la forma de representar los números en la llamada notación científica. En notación científica los números se representan como el producto de una cantidad por una potencia de diez:

$$234.25 \equiv 2.345 \times 10^2, \quad 0.000234 \equiv 2.34 \times 10^{-5}, \quad -56.238 \equiv -5.6238 \times 10^1.$$

La idea es desplazar el punto decimal hasta que solo quede una cifra significativa en la parte entera del número y multiplicar el resultado por 10 elevado a la potencia adecuada para recuperar el número original.

Muchas calculadoras y programas de cálculo científico presentan los resultados por pantalla en formato científico. Habitualmente lo hacen con la siguiente notación,

The fixed point representation was widely used in the first computers. However, using the same register size, it is possible to represent a much wider spectrum of numbers using the floating-point representation.

**floating-point representation.** The floating-point representation of numbers in a computer is similar to the numeric representation known as scientific notation. In scientific notation, number are represented as the product of a quantity by a power of ten:

We shift the decimal point till just a single figure remains at its left, i.e, only a figure is used to represent the integer part of the number. We need to multiply the result for the power of ten that allows us to recovery the original number.

Many calculator a scientific computing software show the computed results on the screen using the scientific format. Usually they utilize the following notation to do it,

$$-5.3572 \times 10^{-3} \xrightarrow[\text{calculator}]{\text{calculadora}} -5.3572e-03$$

Es decir, en primer lugar se representa el signo del número si es negativo (si es positivo lo habitual es omitirlo). A continuación la parte significativa, 5.3572, que recibe el nombre de mantisa . Y por último se representa el valor del exponente de la potencia de 10,  $-3$ , precedido por la letra e, —e de exponente—. En notación científica se asume que el exponente corresponde siempre a una potencia de diez. Es evidente que tenemos el número perfectamente descrito sin más que indicar los valores de su signo, mantisa y exponente,

First, if the number is negative, we represent the sign. (Usually, if the number is positive we omit the sign symbol). Then, the part of the number 5.3572 which, when multiplied by the power of ten, retrieves the number represented, this part is called the mantissa of the number. Eventually, we represent the value of the (10) power's exponent,  $-3$ , preceded by the letter e, which stand for exponent. Scientific notation always assumes that the exponent belongs to a power of ten. It is clear that we have the number thoroughly described if we supply its sing, mantissa and exponent.

numero number	r. científica scientific r.	r. calculadora calculator r.	signo sign	mantisa mantissa	exponente exponent
-327.43	$-3.2743 \times 10^2$	$-3.2743e2$	-	3.2743	2

La representación binaria en punto flotante sigue exactamente la misma representación que acabamos de describir para la notación científica. La única diferencia es que en lugar de trabajar con potencias de diez, se trabaja con potencias de dos, que son las que corresponden a la representación de números en binario.

Así, el número binario  $-1101.00101$  se representaría como  $-1.10100101 \times 2^3$ , y el número  $0.001011101$  se representaría como  $1.011101 \times 2^{-3}$ .

El término punto flotante viene del hecho de que el punto decimal de la mantisa, no separa la parte entera del número de su parte decimal. Para poder separar la parte entera de la parte decimal del número es preciso emplear el valor del exponente. Así, para exponente 0, el punto decimal de la mantisa estaría en el sitio que corresponde al número representado, para exponente 1 habría que desplazarlo una posición a la izquierda, para exponente  $-1$  habría que desplazarlo una posición a la derecha, para exponente 2, dos posiciones a la izquierda, para exponente  $-2$ , dos posiciones a la derecha, etc.

¿Cómo representar números binarios en punto flotante empleando los registros de un computador? Una solución inmediata es dividir el registro en tres partes. Una para el signo, otra para la mantisa y otra para el exponente. Supongamos, como en el caso de la representación en punto fijo, que contamos con registros de 16 bits. Podemos dividir el registro en tres zonas, la primera, de un solo bit la empleamos para guardar el signo. La segunda de, por ejemplo 11 bits, la empleamos para guardar la mantisa del número y por último los cuatro bits restantes los empleamos para guardar el exponente en binario. Podríamos entonces representar el número  $-1.10100101 \times 2^3$  como,

The floating-point follows exactly the same representation that we just described for the scientific notation. The only difference is that we now use powers of two, which are the proper powers of binary representation, instead of powers of ten.

So, we would represent the binary number  $-1101.00101$  as  $-1.10100101 \times 2^3$ , and the binary number  $0.001011101$  as  $1.011101 \times 2^{-3}$ .

The term floating-point means that the mantissa decimal point does not split the integer part of the number from its decimal part. We need to use the exponent value to split the integer and decimal part of the number. So, for an exponent equal to zero, the mantissa decimal point is in the correct position to represent the number. For an exponent equal to 1, we must move the decimal point one position towards the left. For an exponent equal to  $-1$ , one position towards the right. For exponent 2, two positions towards the left. For exponent  $-2$ , two positions towards the right. And so on.

How can we represent floating points binary numbers using a computer register? A straightforward solution is to divide the register into three parts, One for the sign, another to store the mantissa and the third part for the exponent. Suppose that, as in the case of the fixed point representation, we have got 16-bit registers. We can divide the register in three zones. The first one, with a single bit to save the number sign. A second one of, let say 11 bits, to save the number mantissa and we reserve the remaining four bits to hold the exponent in binary representation. We could, then, represent the number  $-1.10100101 \times 2^3$  as,

sign.	mantissa												exponent.			
1	1	1	0	1	0	0	1	0	1	0	0	0	0	0	1	1
-	$2^0$	$2^{-1}$	$2^{-2}$	$2^{-3}$	$2^{-4}$	$2^{-5}$	$2^{-6}$	$2^{-7}$	$2^{-8}$	$2^{-9}$	$2^{-10}$		$2^3$	$2^2$	$2^1$	$2^0$

Podríamos entonces representar el número  $-1.10100101 \times 2^3$  como,

Si bien la representación que acabamos de ver sirve para el número representado, es evi-

Although the representation just described suits well the represented number, it is clear

dente que no podemos representar así un número con exponente negativo como por ejemplo  $1.011101 \times 2^{-3}$ . Además la división que hemos hecho del registro de 16 bits es arbitraria; podríamos haber elegido un reparto distinto de bits entre mantisa y exponente. De hecho, hasta la década de los noventa del siglo XX, cada ordenador adoptaba su propia representación de números en punto flotante. La consecuencia era que los mismos programas, ejecutados en máquinas distintas daban diferentes soluciones.

Ya en 1976 John Palmer, de la compañía INTEL, promueve la creación de un estándar, es decir, de una representación en punto flotante que fuera adoptada por todos los fabricantes de computadoras. El primer borrador del estándar fue elaborado por William Kahan (Intel), Jerome Coonen (Berkeley) y Harold Stone (Berkeley): documento K-C-S. Inspirado en el trabajo de Kahan en el IBM 7094 (1968). El proceso de estandarización fue bastante lento. La primera versión no fue operativa hasta 1985. El resultado de estos trabajos fue el estándar actual. Conocido como IEEE 754. IEEE son las siglas de Institute of Electrical and Electronics Engineers. Se trata de una asociación que nació en Estados Unidos en el siglo XIX y que goza actualmente de carácter internacional. El IEEE se ocupa, entre otras cosas, de establecer estándares industriales en los campos de la electricidad, la electrónica y la computación. Veamos en detalle el estándar IEEE 754.

**El estándar IEEE 754.** En primer lugar, el estándar se estableció pensando en dos tamaños de registro el primero de ellos emplea 32 bits para guardar los números en punto flotante y recibe el nombre de estándar de precisión simple. El segundo emplea 64 bits y se conoce como estándar de precisión doble.

**Simple precisión.** Empecemos por describir el estándar de precisión simple. En primer lugar, se reserva un bit para contener el signo. Si el bit vale 1, se trata de un número negativo, y si vale 0 de un número positivo. De los restantes 31 bits, 23 se emplean para

that we cannot represent with it a negative number like, for example,  $1.011101 \times 2^{-3}$ . Furthermore, we have arbitrarily divided the 16-bit register; we could have chosen a different allocation of bits between the mantissa and exponent. In fact, until the 1990s, each computer adopted its own representation for floating-point numbers. As a consequence, the same programs run on different computers yield different results.

already in 1974, John Palmer from INTEL company, promote the creation of a standard, that is, of a floating-point representation which might be adopted for all computers manufacturers. The first draft of the standard, inspired in Kahan's work for the IBM 7094 computer (1968), was prepared by William Kahan (Intel), Jerome Coonen (Berkeley) and Harold Stone (Berkeley): K-C-S document. The standardisation process was slow. The first version of the standard was not ready till the year 1985. The result of these works was the current standard, known as the IEEE 754 standard. IEEE are the acronym of Institute of Electrical and Electronic Engineers. The Institute is an association born in the United States in the XIX century, and it has evolved to become an international. The IEEE is devoted, among other things, to establish international industrial standards on the electricity, electronics and computer fields. Let's see in some detail the IEEE 754 standard.

**the IEEE 745 standard.** The standard was established thinking in register of two different sizes. The first one uses 32 bits to save floating-point numbers. It is known as the simple precision standard. The second one uses 64 bits and it is known as the double precision standard.

**Simple precision.** We will start describing the simple precision standard. First we take a bit to save the sign. If the bit is one, then the number is negative. If the bit is zero the number is positive. We use 23 bits of the remaining 32, to represent the number mantissa and the other last 8 bits to represent the exponent.

representar la mantisa y 8 para representar el exponente.

Si analizamos como es la mantisa de un número binario en representación de punto flotante, nos damos cuenta de que la parte entera siempre debería valer 1. Por tanto, podemos guardar en la mantisa solo la parte del número que está a la derecha del punto decimal, y considerar que el 1 correspondiente a la parte entera está implícito. Por ejemplo, si tenemos el número binario  $1.010111 \times 2^3$  solo guardariamos la cifra 010111. El 1 de la parte entera sabemos que está ahí pero no lo representamos. A este tipo de mantisa la llamaremos normalizada, más adelante veremos por qué.

En cuanto al exponente, es preciso buscar una forma de representar exponentes positivos y negativos. El estándar establece para ello lo que se llama la representación en *exceso*. Con ocho bits podemos representar hasta  $2^8 = 256$  números distintos. Éstos irían desde el [00000000] = 0 hasta el [11111111] = 255.

Si partimos nuestra colección de números en dos, podríamos emplear la primera mitad para representar números negativos, reservando el mayor de ellos para representar el cero, y la segunda para representar números positivos. Para obtener el valor de un número basta restar al contenido del exponente el número reservado para representar el cero. Se dice entonces que la representación se realiza *en exceso* a dicho número.

En el caso del estándar de simple precisión la primera mitad de los números disponibles iría del 0 al 127 y la segunda mitad del 128 al 255. La representación se realiza entonces en exceso a 127. La tabla 5.2 muestra unos cuantos ejemplos de cálculo de la representación *en exceso*.

Tenemos por tanto que el exponente de 8 bits del estándar de precisión simple permite representar todos los exponentes enteros desde el -127 al 128.

Ya tenemos casi completa la descripción del estándar. Solo faltan unos detalles –aunque muy importantes– relativos a la representación de números muy grandes y muy pequeños. Empecemos con los números muy grandes.

¿Cuál es el número más grande que podría-

If we examine a binary number's mantissa when using the floating-point representation, we realize that it must always take a value equal to one. Therefore, we only need to save in the chunk of the register reserved for the mantissa the decimal part, i.e., the digits located on the right of the decimal point, and consider the leading 1, belonging to the integer part of the number, implicit. For instance, if we have the binary number  $1.010111 \times 2^3$  we only save the figure 010111. We know the leading 1 of integer part is there, but we do not represent it. We will call this kind of mantissa a normalised mantissa, we shall see why later on.

We need a way to represent positive and negative exponents. The standard prescribes the use of representation known as biased exponent representation. Using 8 bits, we can represent up to  $2^8 = 256$  different numbers. These numbers expand from [00000000] = 0 to [11111111] = 255.

We divide our numbers into two halves and take the lower half to represent negative numbers, using the largest number to represent the zero. Of course, we use the upper half to represent positive numbers. It is enough to subtract the number reserved to represent the zero to recover the *real* exponent of the number. So we can consider that our representation of the exponent is *biased* by the number used to represent the zero.

For the simple precision standard, the first half of available numbers ranges from 0 to 127 and the second half from 128 to 255. Then, the representation is biased by 127. Table 5.2 shows several examples on how to calculate this *biased* representation.

In conclusion: the 8 bits of the simple precision standard allows us to represent all integer exponents from -127 to 128.

We have almost ready the description of the IEEE standard. However, some essential details about representing very large and very small numbers remain. Let's start with the very large numbers. What is the most significant number we could represent using the simple precision standard? We could suppose

Bits de exponente Exponent's bits	equivalente decimal base-ten equivalent	Exponente representado represented exponent
00000000	0	$0 - 127 = -127$
00000001	1	$1 - 127 = -126$
00000010	2	$2 - 127 = -125$
⋮	⋮	⋮
01111110	126	$126 - 127 = -1$
01111111	127	$127 - 127 = 0$
10000000	128	$128 - 127 = 1$
⋮	⋮	⋮
11111110	254	$254 - 127 = 127$
11111111	255	$255 - 127 = 128$

Tabla 5.2: Representación en exceso a 127, para un exponente de 8 bits.

Table 5.2: 127-biased representation for a 8-bit exponent

mos representar en estándar de simple precisión? En principio cabría suponer que el correspondiente a escribir un 1 en todos los bits correspondientes a la mantisa (la mayor mantisa posible) y escribir también un 1 en todos los bits correspondientes al exponente (el mayor exponente posible). Sin embargo, el estándar está pensado para proteger las operaciones del computador de los errores de desbordamiento (ver sección 5.3 más adelante). Para ello reserva el valor más alto del exponente. Así cuando el exponente contenido en un registro es 128, dicho valor no se considera propiamente un número, de hecho es preciso mirar el contenido de la mantisa para saber qué es lo que representa el registro:

- Si los bits de la mantisa son todos cero, el registro contiene la representación del infinito  $\infty$ . El estándar especifica el resultado de algunas operaciones en el infinito:  $1/\infty = 0$ ,  $\arctan(\infty) = \pi/2$ .
- Si por el contrario, el contenido de la mantisa es distinto de cero, se considera el contenido del registro como no numérico (NaN, abreviatura en inglés de *Not a Number*). Este tipo de resultados, considerados no numéricos, surgen en algunas operaciones que carecen de sentido matemático:  $0/0$ ,  $\infty \times 0$ ,  $\infty - \infty$ .

that, in principle, the number with all its mantissa values set to one (the most significant possible mantissa) and all its exponent values set to one (the largest possible exponent). However, the standard was thought to protect the computer operations against overflow errors (see section 5.3 below). That protection is carried out by reserving the largest possible value of the exponent. Therefore, when a register holds an exponent equal to 128, this value is not considered a proper number. It is necessary to see the mantissa to know what the register does represent:

- If the mantissa bits are all zero, the register holds the representation of infinite  $\infty$ . The standard specifies the results of some operations on infinite:  $1/\infty = 0$ ,  $\arctan(\infty) = \pi/2$ .
- On the contrary, if the content of the mantissa is not zero, the content of the register is considered as non-numeric (NaN, acronym of Not a Number). Some computations, which lack of mathematical sense like  $0/0$ ,  $\infty \times 0$ ,  $\infty - \infty$ , yields this NaN kind of results.

Therefore, the standard limits to 127 the maximum exponent which can be used to represent a finite number. We can build as an

Todo esto hace que el exponente mayor que realmente representa un número sea el 127. Como ejemplo podemos construir el número más grande que podemos representar distinto de infinito. Se trata del número con la mantisa más grande posible, una mantisa formada exclusivamente por unos, seguida por el exponente más grande posible. El exponente más grande en exceso es 127, que corresponde a un exponente en binario  $127 + 127 = 254 \equiv [11111110]$ . Por tanto el número se representaría como,

sig.	←mantisa, 23 bits →	← exponente, 8 bits →
0	1111111111111111111111111	11111110

En base 10 el número representado sería,  $(1 \cdot \mathbf{2^0} + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdots + 1 \cdot 2^{-23}) \times 2^{127} \approx 3.402823466385289 \times 10^{38}$ . Donde hemos representado en negrita el 1 implícito, no presente en la mantisa.

Veamos ahora los números más pequeños, de acuerdo con lo dicho, el número más próximo a cero que podríamos construir sería aquel que tuviera la mantisa más pequeña y el exponente más pequeño, es decir una mantisa formada solo por ceros y un exponente formado solo por ceros. Ese número sería  $(1 \cdot 2^0) \cdot 2^{-127}$  (se debe recordar que la mantisa en la representación del estándar esta normalizada; lleva un 1 entero implícito).

Es evidente que si no fuera por el 1 implícito de la mantisa sería posible representar números aún más pequeños. Por otro lado, si consideramos siempre la mantisa como normalizada, es imposible representar el número 0. Para permitir la representación de números más pequeños, incluido el cero, los desarrolladores del estándar decidieron añadir la siguiente regla: *Si los bits del exponente de un número son todos ceros, es decir, si el exponente representado es -127, se considera que la mantisa de ese número no lleva un 1 entero implícito. Además el exponente del número se toma como -126.* Los números así representados reciben el nombre de números desnormalizados. Veamos algunos ejemplos.

example the largest number we can represent, different to infinity. This number should have the largest possible mantissa, i.e. a mantissa with all values set to one, and the largest possible exponent, 127, using the biasing representation of the simple precision standard. This exponent correspond to a binary exponent  $127 + 127 = 254 \equiv [11111110]$ . Thus, the largest number for simple precision representation would be,

This number in base-ten will take the form,  $(1 \cdot \mathbf{2^0} + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdots + 1 \cdot 2^{-23}) \times 2^{127} \approx 3.402823466385289 \times 10^{38}$ . Where we have written in bold letter the implicit 1, which is not present in the mantissa.

Let's turn now to the smallest numbers, according to the method we are using to represent the numbers, the number closest to zero we can build would be the number with the smallest mantissa and the smallest exponent. That is, a mantissa with all its bits set to zero and also an exponent with only zeros. This number will be,  $(1 \cdot 2^0) \cdot 2^{-127}$  (remember that the standard uses a normalised mantissa, hence it has a implicit integer 1).

If the standard would not use a mantissa with an implicit one, we could represent smaller numbers. On the other hand, using a normalised mantissa, it is impossible to represent the number zero. The standard developers decided to add the following rule to allow representing small numbers, including the zero: *If the exponent bits of a number are all equal to zero, that is, if the exponent of the number is -127, the standard considers that the number's mantissa has not got an integer 1 implicit. In addition, the exponent of the number is taken as -126.* This special numbers are called denormalised numbers. Let's see some examples.

sig.	←mantisa, 23 bits →	← exponent., 8 bits →
0	10100000000000000000000000000000	00000000

El exponente del número representado en la tabla anterior es 0. Por tanto, en exceso a 127 el exponente sería  $0 - 127 = -127$ . Este exponente corresponde a un número desnormalizado por tanto el número expresado en base 10 sería,

$$(1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 0 \dots) \times 2^{-126}$$

Nótese como el exponente ha sido reducido en una unidad (-126) a la hora de calcular la representación decimal del número.

El número más próximo a cero que podemos representar mediante números desnormalizados, será aquel que tenga la menor mantisa posible distinta de cero,

ssig.	←mantissa, 23 bits →	← exponent., 8 bits →
0	00000000000000000000000000001	00000000

que representado en base 10 sería el número:  $(0 \cdot 2^{-1} + 0 \dots + 1 \cdot 2^{-23}) \times 2^{-126} = 2^{-149} \approx 1.401298464324817 \times 10^{-45}$ . Podemos calcular para comparar cual sería el número normalizado más próximo a cero, se trata del número que tiene una mantisa formada solo por ceros y el exponente más pequeño distinto de cero (un exponente igual a cero, implica automáticamente un número desnormalizado),

sig.	←mantissa, 23 bits →	← exponent., 8 bits →
0	00000000000000000000000000000	00000001

Si lo representamos en formato decimal obtenemos:  $(1 \cdot 2^0 + 0 \cdot 2^{-1} + 0 \dots + 0 \cdot 2^{-23}) \times 2^{-126} = 2^{-126} \approx 1.175494350822288 \times 10^{-38}$

Como un último ejemplo, podemos calcular cual es el número desnormalizado más grande. Debería tener la mantisa más grande posible (todo unos) con un exponente formado exclusivamente por ceros,

sig.	←mantissa, 23 bits →	← exponent., 8 bits →
0	11111111111111111111111111111111	00000000

En formato decimal, el número sería:  $(1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \dots + 1 \cdot 2^{-23}) \times 2^{-126} \approx 1.175494210692441 \times 10^{-38}$ . Los dos últimos números representados, son muy próximos entre sí. De hecho, hemos calculado ya su diferencia, al obtener el número desnormalizado más próximo a cero,  $(1 \cdot 2^0 + 0 \cdot 2^{-1} + 0 \dots + 0 \cdot 2^{-23}) \times 2^{-126} = 2^{-126} \approx 1.175494350822288 \times 10^{-38}$ .

The exponent of the number represented in the table above is 0. Therefore, using a 127 biased representation the exponent is  $0 - 127 = -127$ . Then, this exponent belongs to a denormalised and so, the number represented in base-ten will be,

$$(1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 0 \dots) \times 2^{-126}$$

Notice how the exponent has been reduced one unit (-126) to calculate the decimal representation of the number.

The closest number to zero we can represent using denormalised number, will be that with the minimum non-zero mantissa,

Which represented in base-ten will be the number:  $(0 \cdot 2^{-1} + 0 \dots + 1 \cdot 2^{-23}) \times 2^{-126} = 2^{-149} \approx 1.401298464324817 \times 10^{-45}$ . We can compare this number with the closest number to zero we can represent using a normalised mantissa. This number has all zeros mantissa and the smallest possible non-zero exponent (Recall; an all-zero exponent automatically implies a denormalised number),

If we now represent the number in denary system we obtain:  $(1 \cdot 2^0 + 0 \cdot 2^{-1} + 0 \dots + 0 \cdot 2^{-23}) \times 2^{-126} = 2^{-126} \approx 1.175494350822288 \times 10^{-38}$ .

We will calculate, as a final example, the largest number we can represent using denormalised numbers. In this case, the number should have the largest possible mantissa (all ones) and a all-zeros exponent,

In base-ten representation the number would be:  $(1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \dots + 1 \cdot 2^{-23}) \times 2^{-126} \approx 1.175494210692441 \times 10^{-38}$ . These last two represented number are really near one to another. In fact, we have already calculate their difference, when we got the denormalised number closest to zero,  $(1 \cdot 2^0 + 0 \cdot 2^{-1} + 0 \dots + 0 \cdot 2^{-23}) \times 2^{-126} = 2^{-126} \approx 1.175494350822288 \times 10^{-38}$ .

$$\begin{aligned} 2^{-23}) \times 2^{-126} - (1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdots + 1 \cdot 2^{-23}) \times \\ 2^{-126} = (0 \cdot 2^{-1} + 0 \cdots + 1 \cdot 2^{-23}) \times 2^{-126} = \\ 2^{-149} \approx 1.401298464324817 \times 10^{-45}. \end{aligned}$$

**Doble precisión** La diferencia entre los estándares de simple y doble precisión está en el tamaño de los registros empleados para representar los números. En doble precisión se emplea un registro de 64 bits. Es decir, el tamaño de los registros es el doble que el empleado en simple precisión. El resto del estándar se adapta al tamaño de la representación; se emplea 1 bit para el signo del número, 52 bits para la mantisa y 11 bits para el exponente.

En este caso el exponente puede almacenar  $2^{11} = 2048$  números (desde el 0 al 2047). Como en el caso de estándar de simple precisión se dividen los números por la mitad, con lo que los exponentes se representan ahora en exceso a 1023. Por tanto el exponente 0 representa el valor  $0 - 1023 = -1023$  y el exponente 2047 representa el valor  $2047 - 1023 = 1024$ .

De nuevo, el valor mayor del exponente, 1024 se emplea para representar el infinito  $\infty$ , si va acompañado de una mantisa formada exclusivamente por ceros. En caso de que acompañe a una mantisa no nula, el contenido del registro se considera que no representa un número NaN (error de desbordamiento). Podemos, como ejemplo, obtener para el estándar de doble precisión el número más grande representable,

$$\begin{aligned} 0 \cdot 2^{-1} + 0 \cdots + 0 \cdot 2^{-23}) \times 2^{-126} - (1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdots + 1 \cdot 2^{-23}) \times 2^{-126} = (0 \cdot 2^{-1} + 0 \cdots + 1 \cdot 2^{-23}) \times 2^{-126} = 2^{-149} \approx 1.401298464324817 \times 10^{-45}. \end{aligned}$$

**Double precision** The difference between the single and the double precision standards is in the size of the registers we use to represent the numbers. When working with double precision, we use 64-bits size registers. Therefore, the size of the register is double than the size we use for simple precision. The standard is just fit to the larger size of the registers; we reserve a bit for the number's sign, 52 bits for the mantissa and 11 bits for the exponent.

In this case the exponent can store  $2^{11} = 2048$  different numbers (from 0 to 2047). Very much alike the simple precision case, we split the exponent numbers into two halves, representing the values with a 1023 bias. Therefore, the an exponent equal to zero represent the value  $0 - 1023 = -1023$  and the exponent 2047 represent the value,  $2047 - 1023 = 1024$ .

Again, we use the largest exponent value 1024 to represent the infinite  $\infty$  with an all-zeros companying mantissa. Otherwise, if the companying mantissa has any non-zero bit, then the register does not represent a number NaN (overflow error). As an example, We can obtain the largest representable number using the double precision standard,

sig.	$\leftarrow$ mantissa, 52 bits $\rightarrow$	$\leftarrow$ exponent., 11 bits $\rightarrow$
0	1110	11111111110

El número representado toma en base 10 el valor:  $(1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdots + 1 \cdot 2^{-52}) \times 2^{1023} \approx 1.797693134862316 \times 10^{308}$

Por último, para exponente -1023 es decir, un exponente formado exclusivamente por ceros, el número representado se considera desnormalizado; se elimina el 1 entero implícito de la representación de la mantisa y se aumenta en una unidad el exponente, que pasa así a valer -1022. Como ejemplo, podemos calcular el número más próximo a cero representable en doble precisión,

The value in base-ten of the represented number is:  $(1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdots + 1 \cdot 2^{-52}) \times 2^{1023} \approx 1.797693134862316 \times 10^{308}$

Lastly, numbers with an exponent -1023, that is, numbers with an all-zero exponent, represent denormalised numbers. We remove the implicit one from the mantissa and increase the exponent on one unit, which takes the value -1022. For instance, we can compute the closest to zero number using the double precision representation,

sig.	$\leftarrow$ mantissa, 52 bits $\rightarrow$	$\leftarrow$ exponent., 11 bits $\rightarrow$
0	001	00000000000

El número representado toma el base 10 el valor:  $(0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdots + 1 \cdot 2^{-52}) \times 2^{-1022} \approx 4.940656458412465 \times 10^{-324}$

La tabla 5.3 resume y compara las características de los dos estándares vistos,

The number represented takes in base-ten the value:  $(0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdots + 1 \cdot 2^{-52}) \times 2^{-1022} \approx 4.940656458412465 \times 10^{-324}$

Table 5.3 summarises and compares the characteristics of both standards we have seen,

**Precisión simple. Registro de 32 bits. Exponente exceso a 127**  
**Simple precision. 32-bits Register. Biased exponent a 127**

Mantissa	exponent. (8 bits) (23 bits)	(-exceso)(-bias)	número representado/represented number
	0	$0 - 127 = -127$	$(-1)^{bs} \cdot (0 \cdot 2^{-1} + \cdots + 0 \cdot 2^{-23}) \times 2^{0-127} = 0$
	$\neq 0$	$0 - 127 \equiv -126$	$(-1)^{bs} \cdot (m_1 \cdot 2^{-1} + \cdots + m_{23} \cdot 2^{-23}) \times 2^{0-127} \equiv -126$
		$1 - 127 = -126$	
	$\forall$	hasta/until $254 - 127 = 127$	$(-1)^{bs} \cdot (1 + m_1 \cdot 2^{-1} + \cdots + m_{23} \cdot 2^{-23}) \times 2^{(e_8 \cdot 2^8 + \cdots + e_1 \cdot 2^0) - 127}$
	0	$255 - 127 = 128$	$(-1)^{bs} \cdot (0 \cdot 2^{-1} + \cdots + 0 \cdot 2^{-23}) \times 2^{255-127} \equiv \infty$
	$\neq 0$	$255 - 127 = 128$	$(-1)^{bs} \cdot (m_1 \cdot 2^{-1} + \cdots + m_{23} \cdot 2^{-23}) \times 2^{255-127} \equiv \text{NaN}$

**Precisión doble. Registro de 64 bits. Exponente exceso a 1023**  
**Double precision. 64-bits register. Biased exponent 1023**

Mantissa	exponent. (11 bits) (52 bits)	(-exceso)(-bias)	número representado/represented number
	0	$0 - 1023 = -1023$	$(-1)^{bs} \cdot (0 \cdot 2^{-1} + \cdots + 0 \cdot 2^{-52}) \times 2^{0-1023} = 0$
	$\neq 0$	$0 - 1023 \equiv -1022$	$(-1)^{bs} \cdot (m_1 \cdot 2^{-1} + \cdots + m_{52} \cdot 2^{-52}) \times 2^{0-1023} \equiv -1022$
		$1 - 1023 = -1022$	
	$\forall$	hasta $2046 - 1023 = 1023$	$(-1)^{bs} \cdot (1 + m_1 \cdot 2^{-1} + \cdots + m_{52} \cdot 2^{-52}) \times 2^{(e_{11} \cdot 2^{10} + \cdots + e_1 \cdot 2^0) - 1023}$
	0	$2047 - 1023 = 1024$	$(-1)^{bs} \cdot (0 \cdot 2^{-1} + \cdots + 0 \cdot 2^{-52}) \times 2^{2047-1023} \equiv \infty$
	$\neq 0$	$2047 - 1023 = 1024$	$(-1)^{bs} \cdot (m_1 \cdot 2^{-1} + \cdots + m_{52} \cdot 2^{-52}) \times 2^{2047-1023} \equiv \text{NaN}$

Tabla 5.3: Comparación entre los estándares del IEEE para la representación en punto flotante.  
 $(bs$  bit de signo,  $m_i$  bit de mantisa,  $e_i$  bit de exponente)

Table 5.3: Comparison between the IEEE standards for floating point representation. ( $bs$  sign bit,  $m_i$  mantissa bit,  $e_i$  exponent bit)

Para terminar veamos algunos ejemplos de representación de números en el estándar IEEE 754:

1. ¿Cuál es el valor decimal del siguiente número expresado en el estándar del IEEE 754 de simple precisión?

sig.	←mantissa, 23 bits →	← exponent., 8 bits →
1	11000000000000000000000000000000	01111100

- El bit de signo es 1, por lo tanto el número es negativo.
- El exponente sería,  $0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 - 127 =$

We will finish with some examples of number representation using the IEEE 754 standard:

1. Which is the base-ten representation of the following number represented in the simple precision IEEE 754 standard?

- The bit sign is 1 therefore, it is a negative number.
- The exponent is  $0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 - 127 = 124 - 127 = -3$

$$124 - 127 = -3$$

- A la vista del exponente, la mantisa está normalizada,  $1 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} = 1.75$
- Por tanto el número en representación decimal es:  $(-1)^1 \times (1.75) \times 2^{-3} = -0.21875$

2. ¿Cuál es el valor decimal del siguiente número expresado en el estándar del IEEE 754 de simple precisión?

sig.	←mantissa, 23 bits →	← exponent., 8 bits →
0	0100000000000000000000000	10000001

- El bit de signo es 0, por lo tanto el número es positivo.
- El exponente sería,  $1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 1 + 1 \cdot 2^0 - 127 = 129 - 127 = 2$
- A la vista del exponente, la mantisa está normalizada,  $1 \cdot 2^0 + 1 \cdot 2^{-2} = 0.25$
- Por tanto el número en representación decimal es:  $(-1)^0 \times (0.25) \times 2^2 = +5.0$

3. ¿Cuál es la representación en el estándar IEEE 754 de simple precisión del número: 347.625?

- Convertimos el número a binario,  $347.625 = 101011011.101$
- Representamos el número en formato de coma flotante,  $1.01011011101 \times 2^8$
- mantisa: 01011011101 (normalizada)
- exponente:  

$$8 \xrightarrow{\text{exceso } 127} 127 + 8 = 135 \xrightarrow{\text{binario 8 bits}} 10000111$$
- signo: 0 (positivo)

Con lo cual la representación de 347.625 es,

sig.	←mantissa, 23 bits →	← exponent., 8 bits →
0	01011011101000000000000	10000111

4. ¿Cuál es la representación en el estándar IEEE 754 de simple precisión del número:  $\frac{5}{3}$

- $\frac{5}{3} = 1.66666\cdots$
- Pasamos la parte entera a binario:  $1 = 1 \cdot 2^0$
- Pasamos la parte decimal a binario:

- looking at the exponent we realise that the number has a normalised mantissa,  $1 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} = 1.75$
- Thus, the number in denary representation is:  $(-1)^1 \times (1.75) \times 2^{-3} = -0.21875$

2. Which is the base-ten representation of the following number represented in the simple precision IEEE 754 standard?

- The sign bit is 0, therefore is a positive number.
- The exponent is,  $1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 1 + 1 \cdot 2^0 - 127 = 129 - 127 = 2$
- looking at the exponent we see that it is a normalised number,  $1 \cdot 2^0 + 1 \cdot 2^{-2} = 0.25$
- Then, the denary representation of the number is:  $(-1)^0 \times (0.25) \times 2^2 = +5.0$

3. Which is the simple precision standard IEEE 754 representation of the number 347.625?

- We change the number to binary representation,  $347.625 = 101011011.101$
- We get the floating point representation of the number,  $1.01011011101 \times 2^8$
- mantissa: 01011011101 (normalised)
- exponent:  

$$8 \xrightarrow{\text{bias } 127} 127+8 = 135 \xrightarrow{\text{binary 8 bits}} 10000111$$
- sign: 0 (positive)

Then the representation of 347.625 is,

4. What is the standard IEEE 754 simple precision representation of the number?:  $\frac{5}{3}$

- $\frac{5}{3} = 1.66666\cdots$
- We pass the integer part to binary:  $1 = 1 \cdot 2^0$
- We pass the decimal part to binary:

$$0.666666\cdots \times 2 = 1.333333\cdots$$

$$0.333333\cdots \times 2 = 0.666666\cdots$$

A partir de aquí se repite el periodo  $\widehat{10}$  indefinidamente:  $1.666666 \xrightarrow{\text{binario}} 1.101010\cdots$

- Mantisa: el número en binario quedaría:  $1.101010\cdots$ , con la misma representación en punto flotante,  $1.101010\cdots \times 2^0$ . La mantisa será,  $10101010101010101010101$
- Exponente:  
 $0 \xrightarrow{\text{exceso } 127} 127 + 0 = 127 \xrightarrow{\text{binario 8 bits}} 01111111$
- Signo: 0 (positivo)

Con lo cual la representación de  $\frac{5}{3}$  es,

sig.	$\leftarrow$ mantisa, 23 bits $\rightarrow$	$\leftarrow$ exponente, 8 bits $\rightarrow$
0	10101010101010101010101	01111111

### 5.3. Errores en la representación numérica.

Como se ha indicado anteriormente, cualquier representación numérica que empleemos con el ordenador, está sometida a errores derivados del tamaño finito de los registros empleados. Vamos a centrarnos en el estudio de los errores cometidos cuando representamos números empleando el formato de punto flotante del estándar del IEEE754.

En primer lugar, solo hay una cantidad finita de números que admiten una representación exacta, son los que se obtienen directamente de los valores –unos y ceros– contenidos en un registro al interpretarlos de acuerdo con las especificaciones del estándar. Estos números reciben el nombre de números máquina.

Un número real no será un número máquina si,

1. Una vez representado en formato de punto flotante su exponente está fuera del rango admitido para los exponentes: es demasiado grande o demasiado pequeño.
2. Una vez representado en formato de punto flotante su mantisa contiene más dígitos de los bits que puede almacenar la mantisa del estándar.

$$0.666666\cdots \times 2 = 1.333333\cdots$$

$$0.333333\cdots \times 2 = 0.666666\cdots$$

from this point on the period  $\widehat{10}$  repeats indefinitely:  $1.666666 \xrightarrow{\text{binario}} 1.101010\cdots$

- Mantissa: the number representation in binary is:  $1.101010\cdots$ , and the same floating-point representation,  $1.101010\cdots \times 2^0$ . The mantissa is,  $10101010101010101010101$
- Exponent:  
 $0 \xrightarrow{\text{exceso } 127} 127 + 0 = 127 \xrightarrow{\text{binario 8 bits}} 01111111$
- Sign: 0 (positive)

therefore,  $\frac{5}{3}$  binary representation is,

sig.	$\leftarrow$ mantisa, 23 bits $\rightarrow$	$\leftarrow$ exponente, 8 bits $\rightarrow$
0	10101010101010101010101	01111111

### 5.3. Numerical representations Errors.

As we have already stated before, whatever numerical representation we use with a computer is subject to errors due to the computer register's finite size. We are going to focus on the mistakes we make when using the IEEE754 standard for floating point representation to represent real numbers.

First, there is only a finite set of numbers with an exact representation. These numbers are those directly obtained from the values —ones and zeros— contained in the register and interpreted according to the standard specifications. These numbers are called machine numbers.

A real number will not be a machine number if,

1. Once we have represented it in floating point format, the number exponent is out of the admitted range for exponents: it is too large or too small.
2. Once we have represented it in floating point format, the number mantissa has got more digits than bits can store the IEEE 754 standard mantissa.

Si el exponente se sale del rango admitido, se produce un error de desbordamiento. Si se trata de un valor demasiado grande, el error de desbordamiento se produce por exceso (*overflow*). El ordenador asignará al número el valor  $\pm\infty$ . Si el exponente es demasiado pequeño, entonces el desbordamiento se produce por defecto (*underflow*) y el ordenador asignará al número el valor cero.

Si el tamaño de la mantisa del número excede el de las mantisas representables, la mantisa se trunca al tamaño adecuado para que sea representable. Es decir, se sustituye el número por un número máquina cercano, este proceso se conoce como redondeo y el error cometido en la representación como error de redondeo.

### 5.3.1. Error de redondeo unitario

Supongamos que tenemos un número no máquina  $x = (1.a_1a_2 \dots a_{23}a_{24} \dots) \times 2^{\exp}$ .

**Aproximación por truncamiento.** Si queremos representarlo empleando el estándar de simple precisión, solo podremos representar 23 bits de los que componen su mantisa. Una solución es truncar el número, eliminando directamente todos los bits de la mantisa más allá del 23  $x \approx x_T = (1.a_1a_2 \dots a_{23}) \times 2^{\exp}$ . Como hemos eliminado algunos bits, el número máquina  $x_T$ , por el que hemos aproximado nuestro número, es menor que él.

**Aproximación por exceso.** Otra opción sería aproximar el número máquina inmediatamente superior. Esto sería equivalente a eliminar todos los bits de la mantisa más allá del 23 y sumar un bit en la posición 23 de la mantisa  $x \approx x_E = (1.a_1a_2 \dots a_{23} + 2^{-23}) \times 2^{\exp}$ . En este caso, estamos aproximando por un número máquina mayor que el número aproximado.

**Redondeo** Es evidente que cualquier número real que admite una representación aproximada en el formato del estándar estará comprendido entre dos números máquina:  $x_T \leq x \leq x_E$ .

If the exponent is beyond the admissible range, it leads to overflow or underflow. Overflow happens when the exponent is too large, while underflow occurs when the exponent is too small. In the first case (overflow) the computer will assign the number the value  $pm\infty$ . In the second case, (underflow) the computer will assign the number the value zero.

If a number mantissa size exceeds the size of a representable mantissa, it is cut off to a suitable size to make it representable. In other words, we replace the number with a nearby machine number. This process is known as rounding-off and the error we make as rounding error.

#### 5.3.1. round-off error

Suppose we have a non-machine number  $x = (1.a_1a_2 \dots a_{23}a_{24} \dots) \times 2^{\exp}$ .

**Approximation by Truncation.** If we want to represent the number using the simple precision standard, we can only represent the first 23 bits of its mantissa. A possible solution is to truncate the number, eliminating all mantissa bits beyond the 23rd,  $x \approx x_T = (1.a_1a_2 \dots a_{23} + 2^{-23}) \times 2^{\exp}$ . Because we have eliminated some bits, the machine number  $x_T$  we have taken to approximate our number is less than it.

**Approximation by Excess.** Another option is to approximate the number using the next upper machine number. This is the same that eliminates all mantissa bits beyond the 23rd and then adds one bit to the 23rd mantissa position value,  $x \approx x_E = (1.a_1a_2 \dots a_{23} + 2^{-23}) \times 2^{\exp}$ . Notice that now, we are using a machine number greater than the number we are approximating.

**Rounding.** Whatever number admits an approximate representation using the standard format will be comprised of two machine numbers:  $x_T \leq x \leq x_E$ .

We will follow, as a general criterion, to approximate each real number, either by truncation or excess, always using the nearest number. Whenever we round a number, we are get-

En general, el criterio que se sigue es aproximar cada número real, por truncamiento o exceso, empleando en cada caso el número máquina más cercano. Siempre que redondeamos un número cometemos un error, que podemos definir como el valor absoluto de la diferencia entre el valor real del número y su aproximación. Este error recibe el nombre de error absoluto,

$$\text{Error absoluto/absolute error} = |x - x_r|$$

Donde  $x_r = x_T$  si se redondeó por truncamiento o  $x_r = x_E$  si se redondeó por exceso.

El intervalo entre dos números máquina consecutivos puede obtenerse restando el menor del mayor,

ting an error that we can define as the absolute value of the difference between the actual valor of the number and the machine number chosen to represent it. This error is called absolute error.

$$x_E - x_T = (1.a_1a_2 \cdots a_{23} + 2^{-23}) \times 2^{\exp} - (1.a_1a_2 \cdots a_{23}) \times 2^{\exp} = 2^{-23} \times 2^{\exp}$$

Si aproximamos ahora cualquier número real comprendido en el intervalo  $x_T$  y  $x_E$  por el más cercano de estos dos, el error absoluto que cometemos, sera siempre menor o como mucho igual que la mitad del intervalo,

$$|x - x_r| \leq \frac{1}{2}|x_E - x_T| = \frac{1}{2} \cdot 2^{-23} \cdot 2^{\exp}$$

Este resultado se ilustra gráficamente en la figura 5.1

Where  $x_r = x_T$ , if we rounded the number by truncation or  $x_r = x_E$  if we rounded the number by excess.

We can obtain the interval between two consecutive machine numbers by subtracting the lower from the larger.

If we approximate any real number inside the interval  $(x_T, x_E)$  by the nearest of this last two, the absolute error we get is always less or as much equal to half the interval,

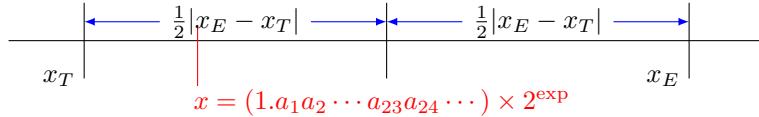


Figura 5.1: Posición relativa de un número no máquina  $x$  y su redondeo a número máquina por truncamiento  $x_T$  y por exceso  $x_E$ . Si redondeamos al más próximo de los dos, el error es siempre menor o igual a la mitad del intervalo  $x_E - x_T$ .

Figure 5.1: Location of a non-machine number in relation to its rounding to a machine number through truncation ( $x_T$ ) and excess ( $x_E$ ). When we round the number to the nearest of the two, the error is always less than half the interval  $x_E - x_T$ .

Examinando con un poco de detalle el resultado anterior, vemos que consta de tres términos. El término  $\frac{1}{2}$  surge de aproximar un número real por su número máquina más cercano, El termino  $2^{\exp}$  depende del tamaño del número. Para números grandes este factor será grande y para números pequeños será un factor

If we look at the previous result in more detail, we can see that is the product of three factors: the term  $\frac{1}{2}$  comes from approximating a real number by its nearest machine number. The term  $2^{\exp}$  depends on the size of the number. For large numbers, this factor will be large, for small numbers, small. The

pequeño. Por último queda el factor  $2^{-23}$ ; este factor está directamente relacionado con la mantisa empleada en la representación. Efectivamente, si hubiéramos representado el número en el estándar de doble precisión, es fácil demostrar que el error absoluto cometido habría quedado acotado como,

$$|x - x_r| \leq \frac{1}{2} |x_E - x_T| = \frac{1}{2} \cdot 2^{-52} \cdot 2^{\exp}$$

Es decir, el único factor que cambia en el error es precisamente el término relacionado con el tamaño de la mantisa. Este término recibe el nombre de precisión del computador o *epsilon del computador* (*eps*). Y vale siempre  $2$  elevado a menos ( $-$ ) el número de bits de la mantisa. Por tanto, podemos generalizar la expresión para la cota del error absoluto como,

$$|x - x_r| \leq \frac{1}{2} |x_E - x_T| = \frac{1}{2} \cdot \text{eps} \cdot 2^{\exp}$$

El significado del *epsilon* del computador queda aún más claro si definimos el error relativo,

$$\text{Error relativo/Relative error} = \frac{|x - x_r|}{|x|} \leq \frac{1}{2} \frac{|x_E - x_T|}{|x|} = \frac{1}{2} \cdot \frac{\text{eps} \cdot 2^{\exp}}{(1.a_1a_2 \cdots a_{23}a_{24} \cdots) \times 2^{\exp}} \leq \frac{1}{2} \text{eps}$$

El error relativo pondera el valor del error cometido con la magnitud del número representado. Un ejemplo sencillo ayudará a entender mejor su significado. Imaginemos que tuviéramos un sistema de representación que solo nos permitiera representar números enteros. Si queremos representar los números  $1.5$  y  $1000000.5$  su representación sería  $1$  y  $1000000$  en ambos casos hemos cometido un error absoluto de  $0.5$ . Sin embargo si comparamos con los números representados, en el primer caso el error vale la mitad del número mientras que en el segundo no llega a una millonésima parte.

En el caso de la representación que estamos estudiando, el error relativo cometido para cualquier número representable es siempre más pequeño que la mitad del *epsilon* del computador,

$$x_r = x \cdot (1 + \delta); |\delta| \leq \frac{1}{2} \cdot \text{eps}$$

last term is a fixed factor  $2^{-23}$  and it is directly related with the mantissa size used in the representation. Indeed, if we had used the double precision standard, the absolute error we would have gotten would be limited as,

That is, the only factor which changes is precisely the term related with the mantissa size. This term is known as the machine precision or the *machine epsilon* (*eps*) and it takes always the value  $2$  raised to minus ( $-$ ) the mantissa number-of-bits. Therefore, we can generalise the upper limit of the absolute error as,

The machine *epsilon* meaning is still easy to understand if we define relative error,

The relative error weighs the value of the gotten error with the magnitude of the represented number. A simple example may help to better understand its meaning. Consider a representation system that only allows integers to represent integer numbers. If we want to represent the numbers  $1.5$  and  $1000000.5$  their representations would be  $1$  and  $1000000$ . In both cases we get an absolute error equal to  $0.5$ . But if we compare this results with the represented numbers, in the first case the error is half the number represented while in the second case this error does not reach the millionth part of the number.

For the representations we are studying, the relative error we get for whatever representable number is always less than half the machine *epsilon*,

Un último comentario sobre el *epsilon* del computador, entendido como precisión. La diferencia entre dos números máquina consecutivos está estrechamente relacionada con el *epsilon*. Si tenemos un número máquina y queremos incrementarlo en la cantidad más pequeña posible, dicha cantidad es precisamente el *epsilon*, multiplicado por 2 elevado al exponente del número. La razón de que esto sea así está relacionada con el modo en que se suman dos números en la representación en punto flotante. Supongamos que tenemos un número cualquiera representado en el estándar de precisión simple,

sig.	$\leftarrow$ mantissa, 23 bits $\rightarrow$	$\leftarrow$ exponent., 8 bits $\rightarrow$
0	11110000000000000000000000000	10000000

El número representado tiene de exponente  $2^7 - 127 = 1$ . Supongamos ahora que quisieramos sumar a este número la cantidad  $2^{-22}$  su representación en el estándar emplearía una mantisa de ceros (recordar el 1 implícito) y un exponente  $-22 + 127 = 105$ . Sería por tanto,

sig.	$\leftarrow$ mantissa, 23 bits $\rightarrow$	$\leftarrow$ exponent., 8 bits $\rightarrow$
0	00000000000000000000000000000	01101001

Para sumar dos números en notación científica es imprescindible que los dos estén representados con el mismo exponente, para entonces poder sumar directamente las mantisas. Disminuir el exponente del mayor de ellos hasta que coincida con el del menor no es posible, ya que eso supondría añadir dígitos a la parte entera de la mantisa, pero no hay bits disponibles para ello entre los asignados a la mantisa. Por tanto, la solución es aumentar el exponente del menor de los números, hasta que alcance el valor del exponente del mayor, y disminuir el valor de la mantisa desnormalizándola, es decir sin considerar el 1 implícito. Por tanto en nuestro ejemplo, debemos representar  $2^{-22}$  empleando un exponente 1,  $2^{-22} \rightarrow 2^{-23} \cdot 2^1$ ,

sig.	$\leftarrow$ mantisa (desnorm.), 23 bits $\rightarrow$	$\leftarrow$ exponente, 8 bits $\rightarrow$
0	00000000000000000000000000001	10000000

La suma de ambos números se obtiene sumando directamente las mantisas,

sig.	$\leftarrow$ mantisa, 23 bits $\rightarrow$	$\leftarrow$ exponente, 8 bits $\rightarrow$
0	11110000000000000000000000001	10000000

¿Qué pasa si tratamos de sumar un número más pequeño, por ejemplo  $2^{-23}$ ? Al repre-

A last comment on the machine *epsilon* considering it from the machine precision point of view. The difference between two consecutive machine numbers is tightly related to the *epsilon*. If we have a machine number and we want to increase it by the least possible quantity, such quantity is precisely the *epsilon* times two rises to the number exponent. The reason for this is the way we add numbers when using the numbers floating point representation. Suppose we have a number whatsoever represented using the simple precision standard,

The represented number has exponent  $2^7 - 127 = 1$ . Suppose now that we want to add to this number the quantity  $2^{-22}$ . Its representation in the standard should use an all-zeros mantissa (recall the implicit 1) and an exponent  $-22 - 127 = 105$ . So it should be,

To add two numbers using scientific notation, we have to represent both numbers using the same exponent and then directly add their mantissa. We cannot decrease the exponent of the larger number to reach the exponent of the lower one because this means adding numbers to the mantissa integer part. However, this is impossible because there are no more bits available to enlarge the mantissa integer part. Therefore, the only solution available is to increase the exponent of the lower number till it reaches the value of the larger number exponent and to diminish the value of the mantissa, de-normalising it; that is, no longer considering the implicit 1. So, in our example, we must represent  $2^{-22}$  using a exponent equal to 1,  $2^{-22} \rightarrow 2^{-23} \cdot 2^1$ ,

We can now get the sum the numbers, just adding their mantissas.

What happen if we sum a smaller number, for example  $2^{-23}$ ? When we try to represent

sentarlo con exponente 1 para poder sumarlo el número tomaría la forma,  $2^{-23} \rightarrow 2^{-24} \cdot 2^1$ . Es fácil ver el problema, con una mantisa de 23 bits nos se puede representar el número  $2^{-24}$  porque ya no hay *hueco* para él. La mantisa sería cero y –dado que se trata de una representación desnormalizada–, el número resultante sería cero. Por tanto, al sumarlo con el número inicial nos daría este mismo número.

Esto nos lleva a que la precisión del computador no es igual para todos los números representables, sino que depende de su magnitud. La precisión, tomada en función de la distancia entre dos números consecutivos, es: precisión =  $\text{eps} \cdot 2^{exp}$  y su valor se duplica cada vez que aumentamos el exponente en una unidad. La figura 5.2 muestra esquemáticamente este fenómeno.

the number with exponent 1, the number will takes the form  $2^{-23} \rightarrow 2^{-24} \cdot 2^1$ . It is easy to see the problem, with a 23-bits mantissa it is not possible to represent the number  $2^{-24}$  because there is not *place* for it. the mantissa would be equal to zero and, due to we are using a denormalised representation, the resulting number is zero. Therefore, when we try to add it to the other number we obtain this same number again.

This lead us to realise that the precision is not equal for every representable number but it depends on the number size. If we associate the precision with the distance between two consecutive machine numbers we may write: precision =  $\text{eps} \cdot 2^{exp}$  and its values duplicates each time we increase the exponent in one unit. Figure 5.2 depicts this phenomenon.

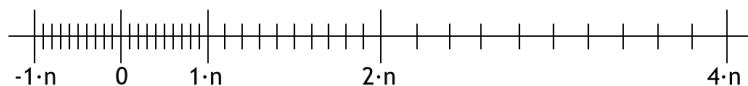


Figura 5.2: Ilustración del cambio de precisión con la magnitud de los números representados.

Figure 5.2: Graphic illustration of precision change with the magnitude of represented numbers

### 5.3.2. Errores de desbordamiento

En el apartado anterior hemos visto una de las limitaciones de la representación numérica de los ordenadores; el hecho de usar una mantisa finita hace que solo unos pocos números tengan representación exacta, el resto hay que aproximarlos cometiendo errores de redondeo. En este apartado nos centraremos en estudiar las limitaciones debidas al hecho de usar un exponente finito; solo podemos representar un rango limitado de valores.

La figura 5.3 muestra esquemáticamente el rango de números representables. El número negativo más pequeño representable, viene definido, por un bit de signo 1, para indicar que se trata de un número negativo y la mantisa y el exponente más grandes que, dentro de las especificaciones del estándar, todavía representan un número finito. Cualquier número negativo menor que éste, produce un error de desbordamiento conocido en la literatura

### 5.3.2. Underflow and Overflow errors

In previous sections we have seen one of the computer limitations to represent numbers; a finite mantissa makes that only a few numbers has an exact representation. We need to approximate the remaining numbers, getting in the process round-off errors. In this section, we focus on the limitations derived from the use of finite exponents, which the range of number we can represent using a computer.

Figure 5.3 shows a schematic view of the representable numbers range. The lowest negative representable number is defined using a sign bit equal to 1, to indicate that it is a negative number, an the largest mantissa and exponent which, according to the standard specification, still represent a finite number. Any lowest negative number will produce an error technically known as negative overflow error. The negative number closest to zero that we can represent will be that which has a sign bit 1, the (denormalised) smallest possible man-

técnica con el nombre de *overflow* negativo. El número negativo más próximo a cero, que se puede representar será aquel que tenga bit de signo 1, la mantisa (desnormalizada) más pequeña posible y el exponente más pequeño posible. Cualquier número más próximo a cero que este, será representado por el ordenador como cero. Se ha producido en este caso un error de desbordamiento conocido como *underflow* negativo.

De modo análogo a como hemos definido el número negativo más pequeño representable, podemos definir el número positivo más grande representable. La única diferencia será que en este caso el bit de signo toma valor cero para indicar que es un número positivo. Cualquier número mayor que éste que queramos representar, provocará un error de desbordamiento (*overflow* positivo.) Por último, el número positivo más próximo a cero representable coincide con el correspondiente negativo, de nuevo salvo en el bit de signo, que ahora deberá ser cero. En el caso de tratar de representar un número más pequeño el ordenador no lo distinguirá de cero, produciéndose un desbordamiento denominado *underflow* positivo.

tissa and the lowest possible exponent. Whatever number closer than it to zero will be represented as zero by the computer. In this case we get an error known as negative underflow error.

Similar to how we have defined the lower negative number we can represent, we can also define the largest positive representable number. The only difference with the lower number case will be the sign bit, which now is equal to zero, to indicate that the represented number is positive. If we try to represent a number greater than largest representable, we will get a positive overflow error. Lastly, the positive number closest to zero we can represent is equal to the negative one except for the sign bit, which must now be equal to zero. If we try to represent a smaller number than the smallest representable, the computer cannot tell one from the other, producing a positive underflow and taking zero as the representation of the number.

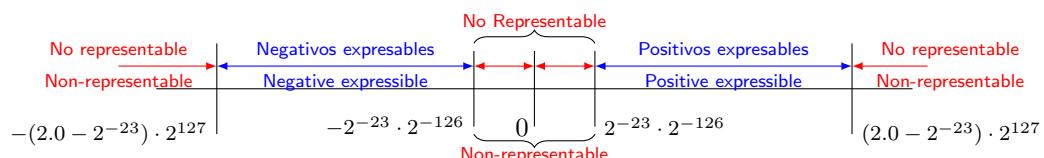


Figura 5.3: Números representables y desbordamientos en el estándar IEEE 754 de precisión simple.  
Figure 5.3: Representable numbers and over/under-flows in the IEEE 754 simple precision standard.

## 5.4. Errores derivados de las operaciones aritméticas

Hasta ahora, nos hemos centrado siempre en el problema de la representación numérica del computador. En esta sección vamos a introducir un nuevo tipo de errores que tienen si cabe todavía más importancia desde el punto de vista del cálculo científico. Se trata de los

## 5.4. Errors derived from arithmetic operation

So far, we have focused on the numerical representation problem. In this section we will introduce a new kind of errors we are still more critical from the point of view of scientific computing; these are errors derived from the arithmetic operations.

errores derivados de las operaciones aritméticas.

Indirectamente ya se introdujo el problema en la sección anterior al hablar de la precisión del computador y de la necesidad de igualar los exponentes de los sumandos al mayor de ellos antes de poder sumar en formato de punto flotante. Veamos en más detalle algunas consecuencias de la aritmética en punto flotante.

#### 5.4.1. Acumulación de errores de redondeo

Para empezar, se ilustrará el proceso de la suma de dos números representados en base 10 en formato de punto flotante. Supongamos que queremos sumar los números 99.99 y 0.161. Supongamos además que seguimos una representación en punto flotante con las siguientes limitaciones la mantisa es de cuatro dígitos, el exponente es de dos dígitos.

Si sumamos los números, tal y como los hemos escrito más arriba, sin seguir formato de punto flotante, el resultado sería,

$$99.99 + 0.161 = 100.151$$

Supongamos que los representamos ahora en el formato de punto flotante descrito más arriba,  $99.99 = 9.999 \times 10^1$ ,  $0.161 = 1.610 \times 10^{-1}$ . Vamos a descomponer el proceso de sumar estos dos números en cuatro pasos, que reflejan, esquemáticamente, el proceso que seguiría un computador.

**1. Alineamiento.** Consiste en representar el número más pequeño empleando el exponente del mayor. Para ellos se desplazan hacia la derecha los dígitos del número más pequeño tantas posiciones como indique la diferencia de los exponentes de los dos números y se cambiar el exponente del número mas pequeño por el del más grande.

$$1.610 \times 10^{-1} \rightarrow 0.016 \times 10^1$$

Como cabía esperar, el desplazamiento hacia la derecha de la mantisa produce la pérdida de los últimos dígitos del número. Tenemos aquí un primer error de redondeo en el proceso

We partially introduced the problem in the previous section when we spoke about computer precision and how we need to match the exponents of the addends, making them equal to the largest one, before performing the addition in floating-point representation. Let see in more detail some consequences of performing floating point arithmetic operations.

##### 5.4.1. Accumulation of round-off errors.

We start looking at the addition process of two base-ten numbers represented on floating-point format. Suppose we want to add number 99.99 and 0.161. Besides, suppose we use the floating-point representation with the following two limitation: the mantissa is four digit size and the exponent has only two digits.

If we add the number as they are written above, without use the floating-point format the result would be,

Suppose now that we represent the number following the floating-point format previously described,  $99.99 = 9,999 \times 10^1$ ,  $0.161 = 1.610 \times 10^{-1}$ . We will divided the addition process of this two numbers in four steps which, schematically, shows the procedure the computer will carry out.

**1. Alignment.** This first step involves representing the smaller number using the exponent of the larger one: We shift the smaller number digits rightwards so many positions as the difference between the exponents of the numbers, and we equal the exponent of the smaller number to the larger one.

As it can be expected, the rightwards shift of the mantissa digits causes the lost of the last digits of the number. So we find here a first rounding error in the alignment procedu-

de alineamiento.

**2. Operación.** Una vez que los operandos están alineados, se puede realizar la operación. Si la operación es suma y los signos son iguales o si la operación es resta y los signos son diferentes, se suman las mantisa. En otro caso se restan.

Es importante comprobar tras realizar la operación si se ha producido desbordamiento de la mantisa; en el ejemplo propuesto:

$$\begin{array}{r} 9.999 \cdot 10^1 \\ + 0.016 \cdot 10^1 \\ \hline 10.015 \cdot 10^1 \end{array}$$

No es posible emplear directamente el resultado de la suma de las mantisas de los sumandos como mantisa de la suma ya que requeriría emplear un dígito más. El resultado obtenido desborda el tamaño de la mantisa.

**3. Normalización.** Si se ha producido desbordamiento de la mantisa, es preciso volver a normalizarla,

re.

**2. Operation.** Once the addends have been aligned, we can carry out the operation. Whether the operation is an addition and the number signs are equal or whether the operation is a subtraction and the signs are different, it is enough to add the mantissas. Otherwise, we subtract them.

We must check after perform the computation if the mantissa has overflowed. In our example,

$$\begin{array}{r} 9.999 \cdot 10^1 \\ + 0.016 \cdot 10^1 \\ \hline 10.015 \cdot 10^1 \end{array}$$

We cannot use directly the result of the addends mantissas sum as the mantissa of the result because it need an extra digit, i.e., this addition yields a result which overflows the size of the mantissa.

**3. Normalisation.** If the mantissa has overflowed then, it is necessary to normalise it again,

$$10.015 \times 10^1 = 1.0015 \times 10^2$$

**4. Redondeo.** El desplazamiento de la mantisa hace que no quepan todo los dígitos, por lo que es necesario redondearla (por truncamiento o por exceso),

**4. Rounding.** The shifting of the digits makes that not all the digits can be accommodate inside the mantissa. Therefore, it is necessary to round it (by truncation or excess).

$$1.0015 \times 10^2 = 1.002 \times 10^2$$

Por último, se comprueba si las operaciones realizadas han producido desbordamiento del exponente, en cuyo caso el resultado sería un número no representable:  $\infty$  ó 0.

**5. Renormalización.** Para números en representación binaria es preciso a veces volver a normalizar la mantisa después del redondeo. Supongamos, como en el ejemplo anterior, una mantisa de cuatro bits, —ahora con números representados en binario— y que tras realizar una operación suma el resultado que se obtiene es  $11.111 \times 2^1$ . La mantisa ha desbordado y hay que normalizarla  $11.111 \times 2^1 \rightarrow 1.1111 \times 2^2$ . Tenemos que redondear la mantisa y lo normal en este caso, dado que el número estaría exactamente en la mitad del intervalo entre dos números representables, es hacerlo

Eventually, we check if the operations we have carried out have produce an exponent overflow, because in this case the number would be not representable:  $\infty$  or 0.

**5. Renormalisation.** For number in binary representation is sometimes necessary to normalise again the mantissa after carried out the rounding. Suppose as in the previous example that we have a four-bit size mantissa, but now we numbers in binary representation, and that after performing an addition operation it yield the result  $11.111 \times 2^1$ . The mantissa has overflowed and then we have to normalised it  $11.111 \times 2^1 \rightarrow 1.1111 \times 2^2$ . We have to round the mantissa and, due to the number is just on the interval half between two machine number, we round it by excess:

por exceso:  $1.1111 \times 2^2 \rightarrow 10.000 \times 2^2$ . La operación de redondeo por exceso ha vuelto a desbordar la mantisa y, por tanto, hay que volver a normalizarla:  $10.000 \times 2^2 \rightarrow 1.000 \times 2^3$ .

Analicemos el error cometido para la suma empleada en los ejemplos anteriores. En aritmética exacta, el número obtenido tras la suma es 100.152. Empleando la representación en punto flotante, con una mantisa de cuatro dígitos el resultado es  $1.002 \times 10^2 = 100.2$ . por tanto, el error absoluto cometido es,

$1.1111 \times 2^2 \rightarrow 10.000 \times 2^2$ . The round-by-excess process has overflowed the mantissa one more time and, thus, we have to normalize it again  $10.000 \times 2^2 \rightarrow 1.000 \times 2^3$ .

If we analyse the error we get after perform the addition in our previous example, In exact arithmetic, the sum of the numbers yields 100.152. When we use floating-point representation with a four-bit mantissa the result is  $1.002 \times 10^2 = 100.2$ . Then, the absolute error we get is,

$$|100.151 - 100.2| = 0.049$$

y el error relativo,

$$\left| \frac{100.151 - 100.2}{100.152} \right| \approx 0.000489$$

En general puede comprobarse que para cualquier operación aritmética básica  $\odot$  (suma, resta, multiplicación, división) y dos números máquina  $x, y$  se cumple,

and the relative error,

In general, it is possible to demonstrate that for any basic arithmetic operation  $\odot$  (addition, subtraction, multiplication, division) and two machine numbers  $x, y$ , it fulfills that,

$$\text{flotante}(x \odot y) = (x \odot y) \cdot (1 + \delta) \quad |\delta| \leq \text{eps}$$

Este enunciado se conoce como el axioma fundamental de la aritmética en punto flotante: *El eps del computador es la cota superior del error relativo en cualquier operación aritmética básica realizada en punto flotante entre números máquina.*

El axioma fundamental de la aritmética en punto flotante establece una cota superior para el error. En la práctica, es frecuente que se encadenen un gran número de operaciones aritméticas elementales. Al encadenar operaciones, los errores cometidos en cada una de ellas se acumulan.

Supongamos por ejemplo que queremos realizar la operación  $x \cdot (y + z)$  en aritmética flotante. La operación podríamos describirla como,

This assertion is known as the fundamental axiom of floating-point arithmetic: *the machine eps is the upper bound of the relative error in any basic floating-point arithmetic operation performed between two machine numbers.*

The fundamental axiom of floating-point arithmetic sets an error upper bound. In practice, when computing, we link many basic arithmetical operations. This operation linking will lead to an accumulation of the errors we get in each basic operation.

for instance, suppose we want to carry out the operation,  $x \cdot (y + z)$  in floating-point arithmetic. We may describe the operation as follows,

$$\begin{aligned} \text{flotante}(x \cdot (y + z)) &= (x \cdot \text{flotante}(y + z)) \cdot (1 + \delta_1) \\ &= (x \cdot (y + z)) \cdot (1 + \delta_2) \cdot (1 + \delta_1) \\ &\approx (x \cdot (y + z)) \cdot (1 + 2\delta) \end{aligned}$$

Donde  $\delta_1$  representa el error relativo cometido en el producto y  $\delta_2$  el error relativo come-

Where  $\delta_1$  represents the relative error after performing the product and  $\delta_2$  the error deri-

tido en la suma. Ambos errores están acotados por el  $\text{eps}$  del ordenador ( $\delta_1, \delta_2 \leq \text{eps}$ ). El valor  $\delta$  se puede obtener como,

$$(1 + \delta_1) \cdot (1 + \delta_2) = 1 + \delta_1\delta_2 + \delta_1 + \delta_2 \approx 1 + 2\delta, \delta = \max(\delta_1, \delta_2)$$

Podemos concluir que, en este caso, el error de redondeo relativo duplica al de una operación aritmética sencilla. En general, el error tenderá a multiplicarse con el número de operaciones aritméticas encadenadas.

Hay situaciones en las cuales los errores de redondeo que se producen durante una operación aritmética son considerables. Por ejemplo, cuando se suman cantidades grandes con cantidades pequeñas,

$$\text{flotante}(1.5 \cdot 10^{38} + 1.0 \cdot 10^0) = 1.5 \cdot 10^{38} + 0$$

En este caso, durante el proceso de alineamiento, es preciso desplazar la mantisa del número pequeño 38 posiciones decimales para poder sumarlo. Con cualquier mantisa que tenga menos de 38 dígitos el resultado es equivalente a convertir el segundo sumando en cero.

Otro ejemplo es la pérdida de la propiedad asociativa,

$$\left. \begin{array}{l} x = 1.5 \cdot 10^{38} \\ y = -1.5 \cdot 10^{38} \end{array} \right\} \Rightarrow (x + y) + 1 \neq x + (y + 1) \left\{ \begin{array}{l} (x + y) + 1 = 1 \\ x + (y + 1) = 0 \end{array} \right.$$

Los resultados pueden estar sometidos a errores muy grandes cuando la operación aritmética es la sustracción de cantidades muy parecidas. Si, por ejemplo, queremos realizar la operación  $100.1 - 99.35 = 0.75$  y suponemos que estamos empleando una representación en punto flotante con una mantisa de cuatro dígitos y los números representados en base 10 ( $100.1 = 1.001 \cdot 10^2$ ,  $99.35 = 9.935 \cdot 10^1$ )

### 1. Alineamiento

$$9.935 \cdot 10^1 \rightarrow 0.994 \cdot 10^2$$

### 2. Operación

$$\begin{array}{r} | \\ 1.001 \cdot 10^2 \\ - 0.994 \cdot 10^2 \\ \hline 0.007 \cdot 10^2 \end{array}$$

### 3. Normalización

$$0.007 \cdot 10^2 \rightarrow 7.000 \cdot 10^{-1}$$

ved from the addition. Both errors are bounded by the machine  $\text{eps}$ . ( $\delta_1, \delta_2 \leq \text{eps}$ ). Notice that  $\delta$  can be approximated as,

We may conclude that, in this case, the rounding error doubles the error of a simple arithmetical operation. An, in general, the error tends to multiply with the number of linked operations.

In some situations, the rounding errors derived from an arithmetical operation are notable. For example, when we add small quantities with large ones,

In this case, on the alignment process, it is necessary to shift the small number mantissa 38 decimal places to be able to sum it to the large one. Using any mantissa with less than 38 digits is as much as converting the second addend into zero.

Another example is the lost of the associative property,

$$\left. \begin{array}{l} (x + y) + 1 = 1 \\ x + (y + 1) = 0 \end{array} \right.$$

The results may be subject to very large errors when the arithmetical operation is the subtraction of two very similar quantities. Suppose we want to carry out the operation  $100.1 - 99.35 = 0.75$  using a floating-point representation with 4-digits mantissa and the numbers represented in base-10. ( $100.1 = 1.001 \cdot 10^2$ ,  $99.35 = 9.935 \cdot 10^1$ )

### 1. Alignment

Los pasos 4 y 5 no son necesarios en este ejemplo. Si calculamos ahora el error absoluto de redondeo cometido,

$$|0.75 - 0.7| = 0.05$$

Y el error relativo,

$$\left| \frac{0.75 - 0.7}{0.75} \right| \approx 0.0666$$

Es decir, se comete un error de un 6,7 %. El problema en este caso surge porque en el proceso de alineamiento perdemos un dígito significativo.

En un sistema de representación en punto flotante en que los números se representan en base  $\beta$  y el tamaño de la mantisa es  $p$ , Si las sustracciones se realizan empleando  $p$  dígitos, el error de redondeo relativo puede llegar a ser  $\beta - 1$ .

Veamos un ejemplo para números en base 10 ( $\beta = 10$ ). Supongamos que empleamos una mantisa de cuatro dígitos y que queremos realizar la operación  $1.000 \cdot 10^0 - 9.999 \cdot 10^{-1}$ . El resultado exacto sería,  $1 - 0.9999 = 0.0001$ . Sin embargo, en el proceso de alineamiento,

$$9.999 \cdot 10^{-1} \rightarrow 0.9999 \cdot 10^0$$

Si redondeamos el número por exceso, el resultado de la sustracción sería cero. Si lo redondeamos por defecto,

$$\begin{array}{r} 1.000 \cdot 10^0 \\ - 0.999 \cdot 10^0 \\ \hline 0.001 \cdot 10^0 \end{array}$$

y el error relativo cometido sería,

$$\left| \frac{1.000 \cdot 10^{-4} - 1.000 \cdot 10^{-3}}{1.000 \cdot 10^{-4}} \right| = \frac{10^{-4}(10 - 1)}{10^{-4}} = 10 - 1 \Rightarrow \beta - 1$$

Para paliar estos problemas, el estándar IEEE 754 establece que las operaciones aritméticas deben realizarse siempre empleando dos dígitos extra para guardar los resultados intermedios. Estos dos dígitos extra reciben el nombre de dígitos de protección o dígitos de guarda (*guard digits*). Si repetimos la sustracción,  $100.1 - 99.35$ , empleando los dos dígitos de guarda,

### 1. Alineamiento

$$9.935\textcolor{red}{00} \cdot 10^1 \rightarrow 0.993\textcolor{red}{50} \cdot 10^2$$

Steps 4 and 5 are not necessary in this example. If we now calculate the absolute rounding error we get,

$$|0.75 - 0.7| = 0.05$$

And the relative error,

Hence, we have approximately a 6.7 % error. In this case the problem arises during the alignment process, because we lost a significant digit.

In a floating-point representation system, where we represent the number in base- $\beta$  and the mantissa size is  $p$ , if we perform differentiation using  $p$  digits, we can expect a relative rounding error that can reach a maximum value of  $\beta - 1$ .

Let's see an example for numbers written in base-10 ( $\beta = 10$ ). Suppose we are using a four-digits mantissa and we want to carry out the operation  $1.000 \cdot 10^0 - 9.999 \cdot 10^{-1}$ . The exact result would be,  $1 - 0.9999 = 0.0001$ . Nevertheless, in the alignment process,

If we round the number by excess the result of the subtraction will be zero. If we round the number by truncation,

$$\begin{array}{r} 1.000 \cdot 10^0 \\ - 0.999 \cdot 10^0 \\ \hline 0.001 \cdot 10^0 \end{array}$$

and the relative error we get would be,

To alleviate these problems, the IEEE 754 standard establishes that arithmetical operation should be performed always using two extra digits to save the intermediate results. These two extra bits are called protection digit or guard digits. If we repeat the subtraction,  $100.1 - 99.35$ , but using now the two guard digits,

### 1. Alignment.

2. Operación	2. Operation
	$1.00100 \cdot 10^2$
–	$0.99350 \cdot 10^2$
	$0.00750 \cdot 10^2$

3. normalización

$$0.00750 \cdot 10^2 \rightarrow 7.500 \cdot 10^{-1}$$

En este caso, obtenemos el resultado exacto. En general, empleando dos bits de guarda para realizar las sustracciones el error de redondeo es siempre menor que el *eps* del computador.

#### 5.4.2. Anulación catastrófica

La anulación catastrófica se produce cuando en una operación aritmética, típicamente la sustracción, los dígitos más significativos de los operandos, que no están afectados por el redondeo, se cancelan entre sí. El resultado contendrá fundamentalmente dígitos que sí están afectados por errores de redondeo. Veamos un ejemplo.

Supongamos que queremos realizar la operación  $b^2 - 4 \cdot a \cdot c$  empleando números en base 10, y una mantisa de 5 dígitos. Supongamos que los números empleados son:  $b = 3.3357 \cdot 10^0$ ,  $a = 1.2200 \cdot 10^0$ ,  $c = 2.2800 \cdot 10^0$ . El resultado exacto de la operación es,

In this case we obtain the exact result. In general, using two guards bit to perform subtractions, the rounding error is always less than the machine *epsilon*.

#### 5.4.2. Catastrophic cancellation.

Catastrophic cancellation takes place when performing an arithmetical operation –typically the subtraction– the more significant digits that are not affected by the rounding cancel between them. The result will then contain digits that are indeed affected by the rounding. Let's see an example,

Suppose we can carry out the operation  $b^2 - 4 \cdot a \cdot c$  using base-10 numbers and a five-digit mantissa. Suppose also that we employ the numbers:  $b = 3.3357 \cdot 10^0$ ,  $a = 1.2200 \cdot 10^0$ ,  $c = 2.2800 \cdot 10^0$ . The exact result of the operation is,

$$b^2 - 4 \cdot a \cdot c = 4.944 \cdot 10^{-4} \quad (5.1)$$

Si realizamos las operaciones en la representación pedida,

If we perform the operations in the requested representation,

$$\begin{aligned} b^2 &= 1.112689 \cdot 10^1 \\ 4 \cdot a \cdot c &= 1.112640 \cdot 10^1 \\ b^2 - 4 \cdot a \cdot c &= 5.0000 \cdot 10^{-4} \end{aligned}$$

El error relativo cometido es aproximadamente un 1 %. Como se han utilizado dos bits de guarda en las operaciones intermedias, la sustracción no comete en este caso error alguno. El error se debe a los redondeos de los resultados anteriores. Una vez realizada la sustracción, el resultado solo contiene los dígitos sometidos a redondeo ya que los anteriores se han anulado en la operación.

We get a relative error of around a 1 %. We used two guard bits in the intermediate operations, so the subtraction makes no mistake. The error is due to the rounding of previous results. Once the subtraction is performed, the result only holds digits affected by the previous rounding because the first (more significant) digits have been cancelled during the operation.

Como regla práctica se debe evitar al realizar operaciones aritméticas aquellas situaciones en las que se sustraen cantidades casi iguales. Veamos otro ejemplo, para ilustrar esta idea. Supongamos que queremos realizar la operación,

$$y = \sqrt{x^2 + 1} - 1$$

Para valores pequeños de  $x$ , esta operación implica una anulación con pérdida de dígitos significativos. Una posible solución es utilizar una forma alternativa de construir la ecuación. Si multiplicamos y dividimos por el conjugado,

$$y = (\sqrt{x^2 + 1} - 1) \cdot \left( \frac{\sqrt{x^2 + 1} + 1}{\sqrt{x^2 + 1} + 1} \right) = \frac{x^2}{\sqrt{x^2 + 1} + 1}$$

#### 5.4.3. Errores de desbordamiento

Al hablar del rango finito de los números representables, se indicó cómo trata el estándar IEEE 754 los números que desbordan los límites de representación. Al realizar operaciones aritméticas, es posibles llegar a situaciones en las que el resultado sea un número no representable bien por ser demasiado grande en magnitud, (*Overflow* positivo o negativo) o por ser demasiado pequeño, (*underflow* positivo o negativo). Un ejemplo que nos permite ilustrar este fenómeno; es el del cálculo del módulo de un vector,

El siguiente módulo de python contiene dos funciones para calcular la norma de un vector. Vamos a fijarnos en primer lugar en la función `norma_naive`.

As a thumb rule, we have to avoid, when performing arithmetical operation, the subtraction of almost equal quantities. Let's see another example, to clarify this idea. Suppose we want to carry out the operation,

For small values of  $x$ , this operation implies a cancellation with the loss of significant digits. A possible solution is to write the equation using an alternative form. If we multiply and divide by the conjugate,

#### 5.4.3. Overflow and underflow errors

When we spoke above the representable numbers finite rank, we showed how the IEEE 754 standard deals with the numbers that overflows the representation boundaries. When we perform arithmetic operations it is possible to arrive to situations where the operation result be a non-representable number because its magnitude is too large (positive or negative overflow) or because its magnitude is too small (positive or negative underflow). We will present an example that allows us to show this phenomenon; the computing of a vector module,

$$\|\vec{v}\| = \|(v_1, v_2 \dots v_n)\| = \sqrt{\sum_{i=1}^n v_i^2}$$

The following Python module includes two functions to calculate a vector norm. Let's first examine the function `norma_naive`.

---

#### norma\_naive.py

---

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3 """
4 Created on Wed Jul 17 14:43:50 2024
5 This

```

```

6  @author: juan
7  """
8
9  import numpy as np
10
11 def norma_naive(x):
12     """
13         This program (naive) calculates the norm of a vector
14         Parameters
15         -----
16         x : TYPE Real Vector
17             DESCRIPTION. input vector whose norm we wanna get
18
19         Returns
20         -----
21         m: Type Real number
22             Description. The vector norm
23         """
24
25
26     m = 0.
27     for e in x:
28         m = m +e**2
29
30     m =np.sqrt(m)
31     return m
32
33 def norma_robust(x):
34     """
35         This program calculates the norm of a vector
36         Parameters
37         -----
38         x : TYPE Real Vector
39             DESCRIPTION. input vector whose norm we wanna get
40
41         Returns
42         -----
43         m: Type Real number
44             Description. The vector norm
45         """
46     l =x.shape[0]
47     if l==1:
48         m = np.abs(x[0])
49     else:
50         mayor = 0
51         mscalado = -1
52         for e in x:
53             if e == 0:
54                 continue
55             modxi = np.abs(e)
56             if mayor<modxi:
57                 mscalado = 1 + mscalado*(mayor/modxi)**2

```

```

58     mayor = modxi
59
60     else:
61         mscalado = mscalado+(modxi/mayor)**2
62     m = mayor*np.sqrt(mscalado)
63
64     return m

```

---

`norm_naive` provocará un error de desbordamiento incluso para  $n=1$ , si introducimos un número cuyo cuadrado sea mayor que el mayor número representable. Si introducimos el número,  $2^{1024/2} = 1.340780792994260e + 154$  en nuestro programa, Python devolverá como solución `inf`.

In [77]: `x= np.array([2**((1024/2))])`

In [78]: `m = norma(x)`  
`/home/juan/LCC_Python/codigos/aritmetica_del_computador/norm_naive.py:28:`  
`RuntimeWarning: overflow encountered in scalar power`  
`m = m +e**2`

In [79]: `m`  
Out[79]: `inf`

Es decir, el resultado produce un error de desbordamiento. El problema se produce en el bucle, al calcular el cuadrado del número,

$$\left(2^{1024/2}\right)^2 = 2^{1024} > (2 - 2^{-52}) \cdot 2^{1023}$$

Una vez producido el desbordamiento el resto de las operaciones quedan invalidadas. Como en casos anteriores, la solución a este tipo de problemas exige modificar la forma en que se realizan los cálculos. Un primer paso sería igualar el módulo al valor absoluto del número cuando  $n=1$ . De este modo, el modulo del número propuesto en el ejemplo anterior se podría calcular correctamente.

Todavía es posible mejorar el programa, y ampliar el rango de vectores para los que es posible calcular el módulo. Para ello es suficiente recurrir a un pequeño artificio: dividir los elementos del vector por el elemento de mayor tamaño en valor absoluto, calcular el módulo del vector resultante, y multiplicar el resultado de nuevo por el elemento de mayor tamaño,

`norm_naive` will generate an overflow error even for  $n=1$ , provide we introduce a number whose square is greater than the largest representable number. If we introduce the number  $2^{1024/2} = 1.340780792994260e + 154$ , our function will return `inf` and we also get a warning from Python.

So, the result we get is an overflow error. The problem takes place in the loop when it calculate the square of the number,

once the programs produces the overflow, the remaining operation are invalid. Like in previous cases, we can solve this kind of problems modifying the method we follow to compute the result. A first step would be to make the norm equal to the absolute value of the number in case  $n=1$ . This way we can correctly compute the norm of the number proposed in the previous example.

However, if we can widen the rank of the vector for which we want to compute the norm, more than this fix is needed. We can improve our code using a small trick: we divide the elements of the vector for the largest element in absolute value, then calculate the module of the vector so obtained, and eventually multiply again the result for the largest number.

$$\sqrt{\sum_{i=1}^n v_i^2} = |\max v_i| \cdot \sqrt{\sum_{i=1}^n \left(\frac{v_i}{|\max v_i|}\right)^2}$$

La segunda función incluida en el anterior script the python, `norm_robust`, calcula la norma de un vector empleando el procedimiento que acabamos de describir.

The second function included in the python script above, `norm_robust`, computes the norm of a vector using the procedure just described,

```
In [109]: x = np.array([0,0,0,0,0,2**(1024/2),2**(1024/2)])  
  
In [110]: x  
Out[110]:  
array([0.00000000e+000, 0.00000000e+000, 0.00000000e+000, 0.00000000e+000,  
0.00000000e+000, 1.34078079e+154, 1.34078079e+154])  
  
In [111]: m = norma_naive(x)  
/home/juan/LCC_Python/codigos/aritmetica_del_computador/norm_naive.py:28:  
RuntimeWarning: overflow encountered in scalar power  
m = m +e**2  
  
In [112]: m  
Out[112]: inf  
  
In [113]: m = norma_robust(x)  
  
In [114]: m  
Out[114]: 1.8961503816218355e+154
```

Si aplicamos este programa a obtener la norma del vector,

If we apply this program to obtain the norm of the vector,

$$x = [2^{1024/2} \ 2^{1024/2}]$$

obtenemos como resultado,

We get the following result,

$$m = 1.8961503816218355 \cdot 10^{154}$$

En lugar de un error de desbordamiento (*inf*).

Instead an overflow error (*inf*)



# Capítulo/Chapter 6

## Cálculo de raíces de una función Methods for calculating the roots of a function

Sólo el tiempo muestra al hombre  
justo mientras que podrías  
conocer al perverso en un sólo día  
(Sófocles. Edipo rey)

### 6.1. Raíces de una función

Se entiende por raíces de una función real  $f(x) : \mathbb{R} \rightarrow \mathbb{R}$ , los valores  $x = r$  que satisfacen,  $f(r) = 0$ .

El cálculo de las raíces de una función, tiene una gran importancia en la ciencia, donde un número significativo de problemas pueden reducirse a obtener la raíz o raíces de una ecuación.

La obtención de la raíz de una ecuación es inmediata en aquellos casos en que se conoce la forma analítica de su función inversa  $f^{-1}$ ,  $(f(x) = y \Rightarrow f^{-1}(y) = x)$ . En este caso,  $r = f^{-1}(0)$ . Por ejemplo,

$$\begin{aligned}f(x) &= x^2 - 4 \\f^{-1}(y) &= \pm\sqrt{y + 4} \Rightarrow r = f^{-1}(0) = \pm 2\end{aligned}$$

Sin embargo, en muchos casos de interés las funciones no pueden invertirse. Un ejem-

### 6.1. Roots of a function

The roots of a real function  $f(x) : \mathbb{R} \rightarrow \mathbb{R}$  are those values  $x = r$  for which  $f(r) = 0$ .

Obtaining the roots of a function is of great interest because many scientific problems could be reduced to the calculus of the root or roots of a function.

Obtaining the root of a function is trivial for those cases in which we know the inverse function  $f^{-1}$ ,  $(f(x) = y \Rightarrow f^{-1}(y) = x)$ . In this case,  $r = f^{-1}(0)$ . For instance,

Nevertheless, there are many cases where functions cannot be inverted. An example

plo, extraído de la física es la ecuación de Kepler para el cálculo de las órbitas planetarias,

$$x - a \sin(x) = b$$

Donde  $a$  y  $b$  son parámetros conocidos y se desea conocer el valor de  $x$ . La solución de la ecuación de Kepler es equivalente a obtener las raíces de la función  $f(x) = x - a \sin(x) - b$ . (La figura 6.1 muestra un ejemplo de dicha función.) En este caso, no se conoce la función inversa, y solo es posible conocer el valor de la raíz, aproximadamente, empleando métodos numéricos.

from physics is Kepler's equation for planetary orbits.

We are given two known parameters,  $a$  and  $b$ , and we need to find the value of  $x$ . To solve Kepler's equation, we must calculate the roots of the function  $f(x) = x - a \sin(x) - b$ . Figure 6.1 provides an example of this function. Since the inverse function is unknown, we can only obtain an approximate root value using numerical methods.

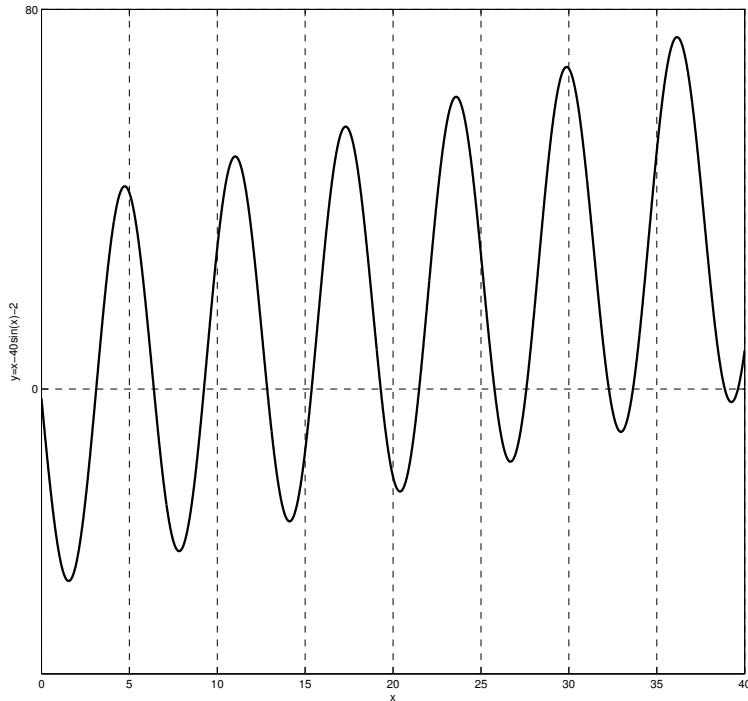


Figura 6.1: Ejemplo de ecuación de Kepler para  $a = 40$  y  $b = 2$ .  
Figure 6.1: An Example of Kepler's equation is taking  $a = 40$  and  $b = 2$ .

**Métodos iterativos** Todos los métodos que se describen en este capítulo, se basan en procedimientos iterativos. La idea es estimar un valor inicial para la raíz  $r_0$ , y a partir de él ir

**Iterative methods** This chapter solely focuses on iterative techniques. These methods begin with an initial guess for the root, denoted as  $r_0$ , and then gradually refine the solu-

refinando paso a paso la solución, de modo que el resultado se acerque cada vez más al valor real de la raíz. Cada nueva aproximación a la raíz se obtiene a partir de las aproximaciones anteriores.

$$\begin{array}{ccccccccc} r_0 & \rightarrow & r_1 & \rightarrow & r_2 & \rightarrow & \cdots & \rightarrow & r_k & \rightarrow \cdots \\ |f(r_0)| & \geq & |f(r_1)| & \geq & |f(r_2)| & \geq & \cdots & \geq & |f(r_k)| & \geq \cdots \end{array}$$

El proceso que lleva de una solución aproximada a la siguiente se conoce con el nombre de *iteración*. Lo habitual es que en cada iteración se realicen las mismas operaciones matemáticas una y otra vez.

El proceso se detiene cuando la solución alcanzada se estima lo suficientemente próxima a la solución real como para darla por buena. Para ello, se suele establecer un valor (*tolerancia*) que actúa como criterio de convergencia. De este modo, las iteraciones se repiten hasta que se llega a un valor  $r_n$  que cumple,

$$|f(r_n)| \leq (tol)$$

Se dice entonces que el algoritmo empleado para obtener la raíz ha convergido en  $n$  iteraciones. Por otro lado, es importante señalar que los algoritmos para el cálculo de raíces de una función no siempre convergen. Hay veces en que no es posible aproximarse cada vez más al valor de la raíz bien por la naturaleza de la función o bien por que el algoritmo no es adecuado para obtenerla.

**Búsqueda local.** Una función puede tener cualquier número de raíces, incluso infinitas, basta pensar por ejemplo en funciones trigonométricas como  $\cos(x)$ . Una característica importante de los métodos descritos en este capítulo es que solo son capaces de aproximar una raíz. La raíz de la función a la que el método converge depende de el valor inicial  $r_0$  con el que se comienza la búsqueda iterativa<sup>1</sup>. Por ello reciben el nombre de métodos locales. Si queremos encontrar varias (o todas) las raíces de una determinada función, es preciso em-

tion through a series of iterative steps. With each step, the solution gets closer and closer to the actual value of the root. The successive approximations of the root are carried out based on the previous ones.

Going from one approximate solution to the next is called *iteration*. Usually, the same mathematical operations are repeated in each iteration.

The iterative process comes to a halt when the obtained solution is considered to be close enough to the actual solution to be considered a good approximation. Typically, a value is set as a convergence criterion, known as tolerance. The iterations continue until a value  $r_n$  is obtained that satisfies the criterion.

$$|f(r_n)| \leq (tol)$$

The algorithm used to obtain the root is then said to have converged in  $n$  iterations. On the other hand, it is important to note that algorithms for calculating the root of a function do not always converge. Sometimes it is not possible to get closer and closer to the value of the root either because of the nature of the function or because the algorithm is not suitable for obtaining it.

**Local search** . A function can have any number of roots, even infinite ones, just think for example of trigonometric functions such as  $\cos(x)$ . An important feature of the methods described in this chapter is that they are only able to approximate one root. The root of the function to which the method converges depends on the initial value  $r_0$  with which the iterative search is started<sup>1</sup>. This is why they are called local methods. If we want to find several (or all) the roots of a given function, it is necessary to use the method on each of the

<sup>1</sup>En ocasiones, como veremos más adelante no se suministra al algoritmo un valor inicial, sino un intervalo en el que buscar la raíz

<sup>1</sup>Sometimes, as we will see later on, the algorithm is not given an initial value, but an interval in which to search for the root

plear el método para cada una de las raíces por separado, cambiando cada vez el punto de partida.

## 6.2. Metodos iterativos locales

### 6.2.1. Método de la bisección

#### Teorema de Bolzano.

Si una función  $f(x)$ , continua en el intervalo  $[a, b]$ , cambia de signo en los extremos del intervalo:  $f(a) \cdot f(b) \leq 0$ , debe tener una raíz en el intervalo  $[a, b]$ . (figura: 6.2)

roots separately, changing the starting point each time.

## 6.2. Local iterative methods

### 6.2.1. Bisection Method

#### Bolzano's theorem

Let  $f(x)$  be a continuous function defined in an interval  $[a, b]$ . Then, if  $f(a) \cdot f(b) < 0$  (therefore,  $f(a) < 0$  and  $f(b) > 0$  or  $f(a) > 0$  and  $f(b) < 0$ ), there exists at least a point inside the interval  $c \in [a, b]$  such that  $f(c) = 0$ . (figure: 6.2)

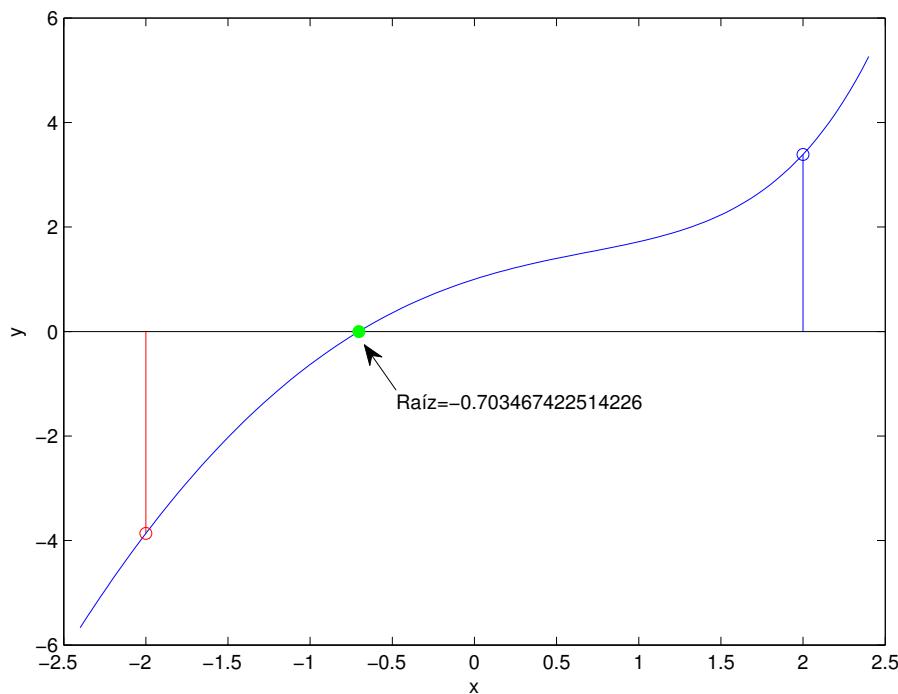


Figura 6.2: Ilustración del teorema de Bolzano  
Figure 6.2: Bolzano's theorem.

El conocido teorema de Bolzano, suministra el método más sencillo de aproximar la raíz de una función: Se parte de un intervalo inicial

The well-known Bolzano's theorem provides the simplest method of approximating the root of a function: We start from an initial in-

en el que se cumpla el teorema; y se va acotando sucesivamente el intervalo que contiene la raíz, reduciéndolo a la mitad en cada iteración, de forma que en cada nuevo intervalo se cumpla siempre el teorema de Bolzano.

terval in which the theorem is satisfied; and we successively limit the interval containing the root, reducing it by half in each iteration, so that in each new interval Bolzano's theorem is always satisfied.

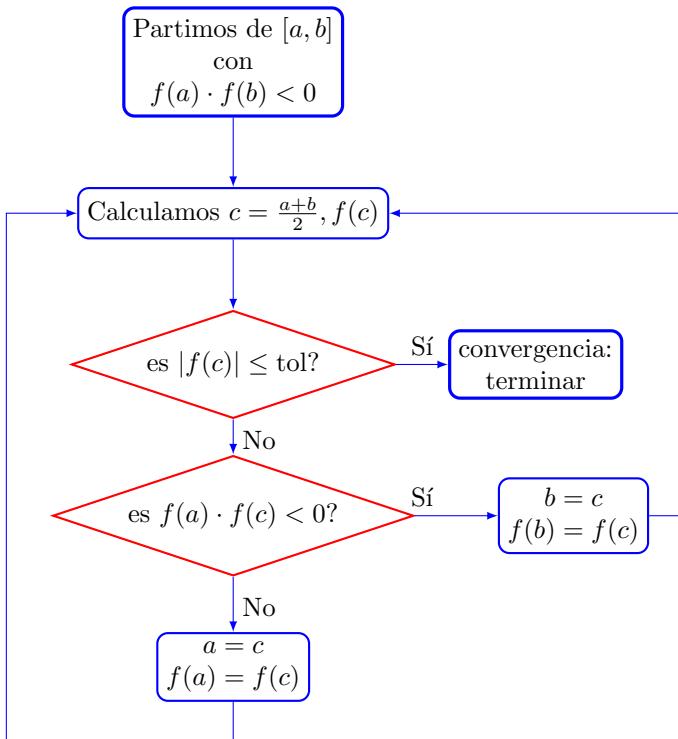


Figura 6.3: Diagrama de flujo del método de la bisección

En la figura 6.3 se muestra un diagrama de flujo correspondiente al método de la biseción. El punto de partida es un intervalo  $[a, b]$  en el que se cumple el teorema de Bolzano, y que contiene por tanto al menos una raíz. Es interesante hacer notar que el teorema de Bolzano se cumple siempre que la función sea continua en el intervalo  $[a, b]$  y existan un número impar de raíces. Por esto es importante realizar cuidadosamente la elección del intervalo  $[a, b]$ , si hay más de una raíz, el algoritmo puede no converger.

Una vez que se tiene el intervalo se calcula el punto medio  $c$ . A continuación se compara el valor que toma la función en  $c$ , es decir  $f(c)$  con la tolerancia. Si el valor es menor que ésta, el algoritmo ha encontrado un valor aproximado de la raíz con la tolerancia requerida, con lo

Figure 6.4 shows a flowchart corresponding to the bisection method. The starting point is an interval  $[a, b]$  in which Bolzano's theorem is satisfied, and which therefore contains at least one root. It is interesting to note that Bolzano's theorem is satisfied whenever the function is continuous on the interval  $[a, b]$  and there are an odd number of roots. This is why it is important to choose the interval  $[a, b]$  carefully. If there is more than one root, the algorithm may not converge.

Once we have the interval we calculate the midpoint  $c$ . The value taken by the function at  $c$ , i.e.  $f(c)$ , is then compared with the tolerance. If the value is less than the tolerance, the algorithm has found an approximate value of the root with the required tolerance, so  $c$  is the root and no further search is neces-

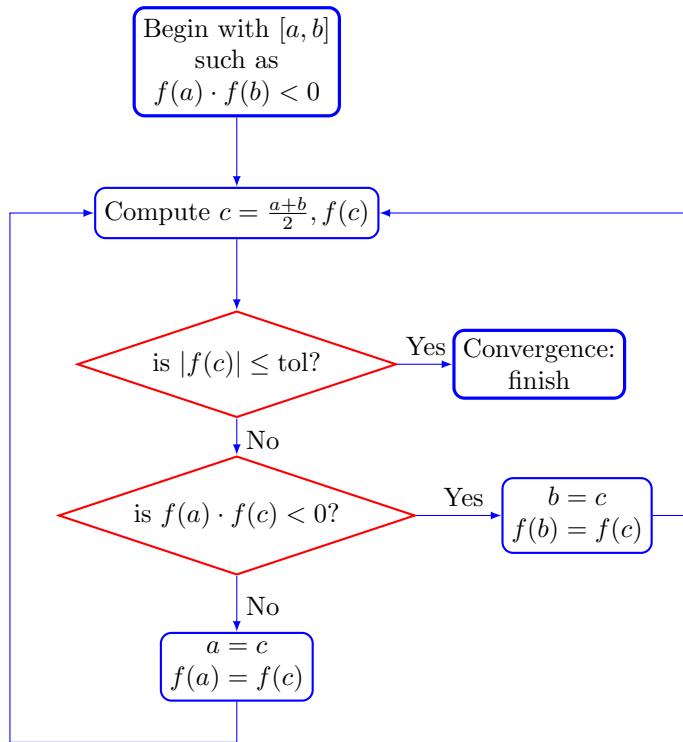


Figura 6.4: Flowchart of the bisection method

que  $c$  es la raíz y no hace falta seguir buscando. Si por el contrario,  $f(c)$  está por encima de la tolerancia requerida, comparamos su signo con el que toma la función en uno cualquiera de los extremos del intervalo. En el diagrama de flujo se ha elegido el extremo  $a$ , pero el algoritmo funcionaría igualmente si eligiéramos  $b$ . Si el signo de  $f(c)$  coincide con el que toma la función en el extremo del intervalo elegido,  $c$  sustituye al extremo, (hacemos  $a = c$  y  $f(a) = f(c)$ ) si por el contrario el signo es distinto, hacemos que  $c$  sustituya al otro extremo del intervalo. (hacemos  $b = c$  y  $f(b) = f(c)$ ). Este proceso se repetirá hasta que se cumpla que  $|f(c)| \leq tol$

El proceso se muestra gráficamente en la figura 6.5, para un caso particular. Se trata de obtener la raíz de la función mostrada en la figura 6.2,  $f(x) = e^x - x^2$ . Esta función tiene una única raíz:  $r \approx -0.7035$ . Para iniciar el algoritmo se ha elegido un intervalo  $[a = -2, b = 2]$ . La figura 6.5, muestra tres iteraciones sucesivas, y la solución final,

sary. If, on the other hand,  $f(c)$  is above the required tolerance, we compare its sign with the sign of the function at either end of the interval. In the flowchart we have chosen the end  $a$ , but the algorithm would work equally well if we chose  $b$ . If the sign of  $f(c)$  coincides with the sign that the function takes at the end of the chosen interval,  $c$  replaces the end, (we make  $a = c$  and  $f(a) = f(c)$ ) if on the contrary the sign is different, we make  $c$  replace the other end of the interval (we make  $b = c$  and  $f(b) = f(c)$ ). This process will be repeated until  $|f(c)| \leq tol$ .

The process is shown graphically in figure 6.5, for a particular case. This is to obtain the root of the function shown in figure 6.2,  $f(x) = e^x - x^2$ . This function has a single root:  $r \approx -0.7035$ . To start the algorithm, an interval  $[a = -2, b = 2]$  has been chosen. Figure 6.5, shows three successive iterations, and the final solution, which is obtained after

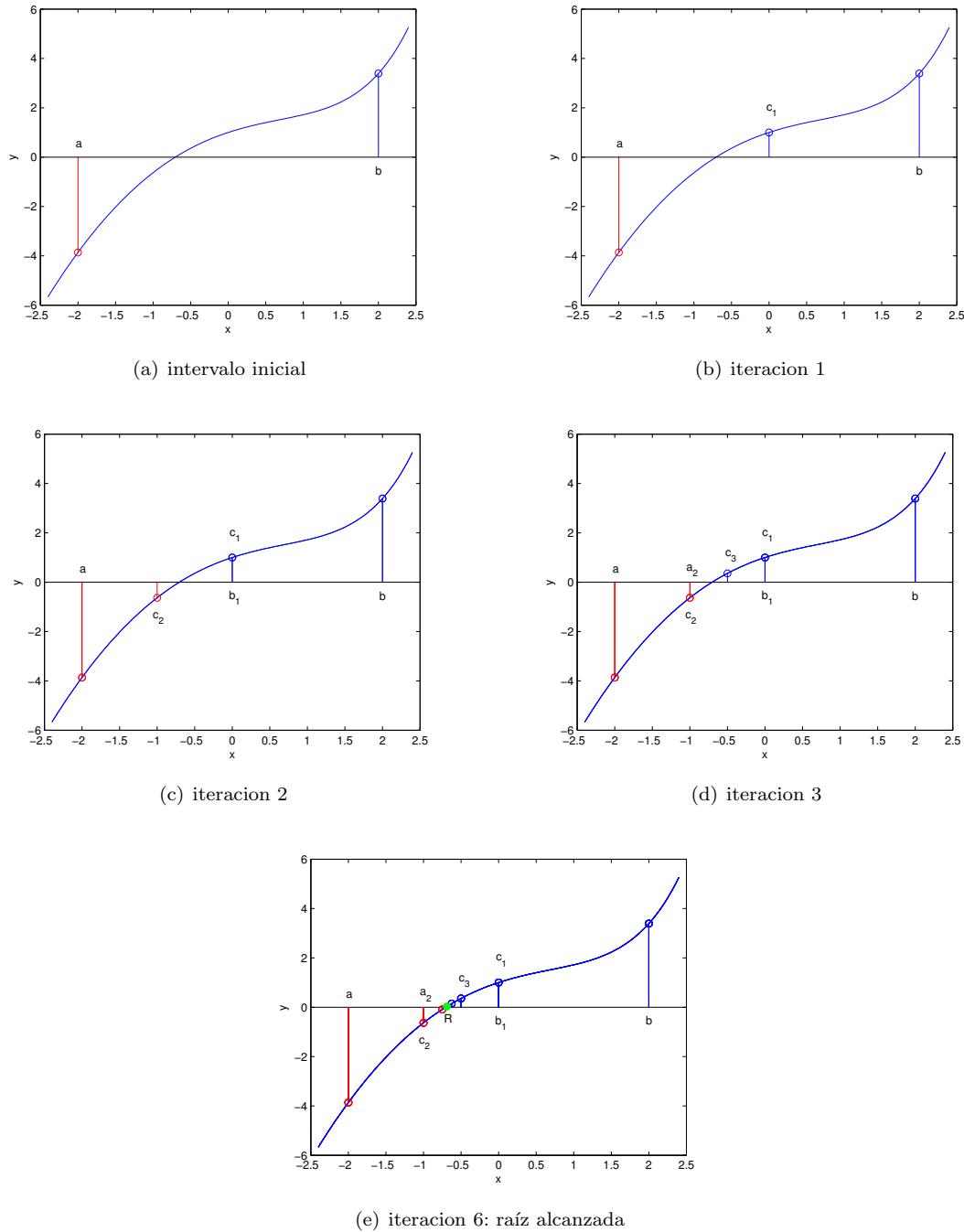


Figura 6.5: proceso de obtención de la raíz de una función por el método de la bisección.

Figure 6.5: process of obtaining the root of a function by the bisection method.

que se obtiene al cabo de ocho iteraciones en éste ejemplo, para el que se a empleado una tolerancia  $tol = 0.01$ . En la secuencia de gráficas se puede observar también la evolución del intervalo de búsqueda,  $[-2, 2] \rightarrow [-2, 0] \rightarrow [-1, 0] \rightarrow [-1, -0.5] \dots$ ; así como el cambio alternativo del límite derecho o izquierdo, para asegurar que la raíz queda siempre dentro de los sucesivos intervalos de búsqueda obtenidos.

### 6.2.2. Método de interpolación lineal o (*Regula falsi*)

Este método supone una mejora del anterior ya que, en general, converge más rápidamente. La idea es modificar el modo en que calculamos el punto  $c$ . En el caso del método de la bisección el criterio consistía en ir tomando en cada iteración el punto medio del intervalo que contiene la raíz. El método de interpolación lineal, elige como punto  $c$  el punto de corte con el eje  $x$ , de la recta que pasa por los puntos  $(a, f(a))$  y  $(b, f(b))$ . Es decir la recta que corta a la función  $f(x)$  en ambos límites del intervalo que contiene a la raíz buscada. La recta que pasa por ambos puntos puede construirse a partir de ellos como,

$$y = \frac{f(a) - f(b)}{a - b} \cdot (x - b) + f(b)$$

el punto de corte con el eje  $x$ , que será el valor que tomaremos para  $c$ , se obtiene cuando  $y = 0$ ,

$$0 = \frac{f(a) - f(b)}{a - b} \cdot (x - b) + f(b)$$

y despejando  $c \equiv x$  en la ecuación anterior obtenemos,

$$c = b - \frac{f(b)}{f(b) - f(a)} \cdot (b - a)$$

La figura 6.6 muestra gráficamente la posición del punto  $c$  obtenido mediante el método de interpolación.

Por lo demás, el procedimiento es el mismo que en el caso del método de la bisección. Se empieza con un intervalo  $[a, b]$  que contienen

eight iterations in this example, for which a tolerance  $tol = 0.01$  has been used. The sequence of graphs also shows the evolution of the search interval,  $[-2, 2] \rightarrow [-2, 0] \rightarrow [-1, 0] \rightarrow [-1, -0.5] \dots$ ; as well as the alternative change of the right or left limit, to ensure that the root is always within the successive search intervals obtained.

### 6.2.2. False position method or (*Regula falsi*)

This method is an improvement on the previous one as it converges more quickly in general. The idea is to modify the way in which we calculate the point  $c$ . In the case of the bisection method, the criterion consisted of taking the midpoint of the interval containing the root at each iteration. The linear interpolation method chooses as point  $c$  the point of intersection with the x-axis of the straight line that passes through the points  $(a, f(a))$  and  $(b, f(b))$ . That is, the line that cuts the function  $f(x)$  at both limits of the interval containing the root we are looking for. The line through both points can be constructed from them as,

$$y = \frac{f(a) - f(b)}{a - b} \cdot (x - b) + f(b)$$

The intersection point with the x-axis, will be the value for  $c$ , is obtained when  $y = 0$ ,

$$0 = \frac{f(a) - f(b)}{a - b} \cdot (x - b) + f(b)$$

and replacing *cequivx* in the above equation, we get

$$c = b - \frac{f(b)}{f(b) - f(a)} \cdot (b - a)$$

Figure 6.6 shows the position of point  $c$  computed with this method.

Otherwise, the procedure is the same as in the case of the bisection method. We start with an interval  $[a, b]$  containing a root, obtain

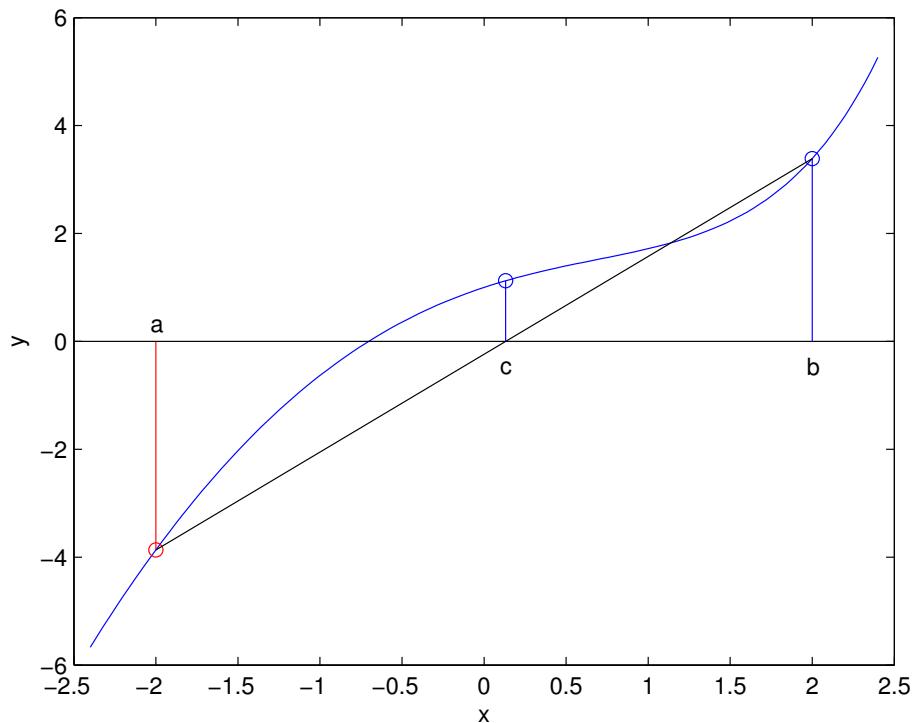


Figura 6.6: Obtención de la recta que une los extremos de un intervalo  $[a, b]$  que contiene una raíz de la función

Figure 6.6: Obtaining the line joining the extremes of an interval  $[a, b]$  containing a root of the function

ga una raíz, se obtiene el punto  $c$  por el procedimiento descrito y se intercambia  $c$  con el extremo del intervalo cuya imagen  $f(a)$  o  $f(b)$  tenga el mismo signo que  $f(c)$  el procedimiento se repite iterativamente hasta que  $f(c)$  sea menor que el valor de tolerancia pre establecido.

En la figura 6.7 se muestra el diagrama de flujo para el método de interpolación lineal. Como puede verse, es idéntico al de la biseción excepto en el paso en que se obtiene el valor de  $c$ , donde se ha sustituido el cálculo del punto medio del intervalo de búsqueda, por el cálculo del punto de corte con el eje de abscisas de la recta que une los extremos del intervalo.

La figura 6.9 Muestra gráficamente el proceso iterativo seguido para obtener la raíz de

the point  $c$  by the procedure described above and exchange  $c$  with the end of the interval whose image  $f(a)$  or  $f(b)$  has the same sign as  $f(c)$ . The procedure is repeated iteratively until  $f(c)$  is less than the preset tolerance value.

The flowchart for the false position method is shown in figure 6.8. As can be seen, it is identical to the bisection method except in the step where the value of  $c$  is obtained, where the calculation of the midpoint of the search interval has been replaced by the calculation of the point of intersection with the abscissa axis of the straight line joining the ends of the interval.

Figure 6.9 shows graphically the iterative process followed to obtain the root of a fun-

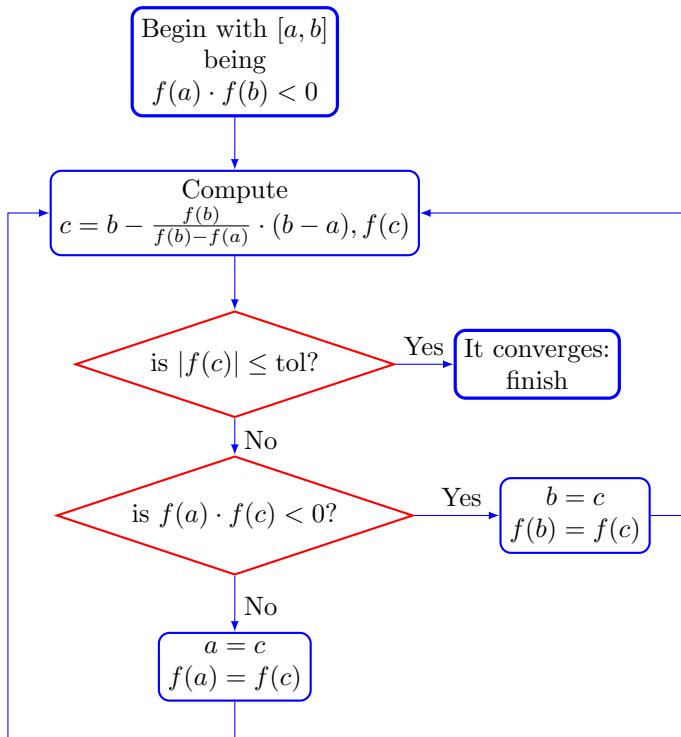


Figura 6.8: Flowchart of false position method

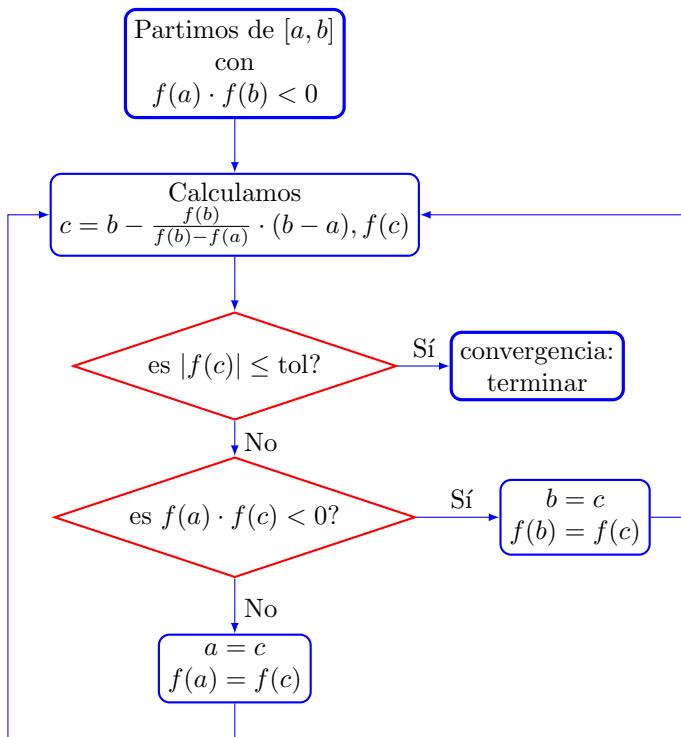


Figura 6.7: Diagrama de flujo del método de interpolación lineal

una función en un intervalo mediante el método de interpolación lineal. Se ha empleado la misma función y el mismo intervalo inicial que en el caso de la biseción.

Es fácil ver, sin embargo, que los puntos intermedios que obtiene el algoritmo hasta converger a la raíz son distintos. De hecho, el al-

gorithm converges in an interval by means of the false position method. The same function and the same initial interval have been used as in the case of bisection.

It is easy to see, however, that the intermediate points that the algorithm obtains until converging to the root are different. In fact,

Una observación final, se ha dicho al principio que éste método supone una mejora al método anterior de la bisección. Esto no siempre es cierto. El método de la bisección tiene una tasa de convergencia constante, cada iteración divide el espacio de búsqueda por la mitad. Sin embargo la convergencia del método de interpolación lineal depende de la función  $f(x)$  y de la posición relativa de los puntos iniciales  $(a, f(a))$  y  $(b, f(b))$  con respecto al la raíz. Por esto no es siempre cierto que converja más rápido que el método de la bisección. Por otro lado, el cálculo de los sucesivos valores del punto  $c$ , requiere más operaciones aritméticas en el método de interpolación, con lo que cada iteración resulta más lenta que en el caso de la bisección.

### 6.2.3. Método de Newton-Raphson

El método de Newton se basa en la expansión de una función  $f(x)$  en serie de Taylor en el entorno de un punto  $x_0$ ,

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2}f''(x_0)(x - x_0)^2 + \cdots + \frac{1}{n!}f^{(n)}(x_0)(x - x_0)^n + \cdots$$

Pertenece a una familia de métodos ampliamente empleados en cálculo numérico. La idea en el caso del método de Newton es aproximar la función para la que se desea obtener la raíz, mediante el primer término de la serie de Taylor. Es decir aproximar localmente  $f(x)$ , en el entorno de  $x_0$  por la recta,

$$f(x_0) + f'(x_0)(x - x_0)$$

Esta recta, es precisamente la recta tangente a la curva  $f(x)$  en el punto  $x_0$  (figura 6.10)

El método consiste en obtener el corte de esta recta tangente con el eje de abscisas,

$$0 = f(x_0) + f'(x_0)(x - x_0)$$

y despejando  $x$ ,

$$x = x_0 - \frac{f(x_0)}{f'(x_0)}$$

A continuación se evalúa la función en el punto obtenido  $x \rightarrow f(x)$ . Como en los méto-

One final remark, it was said at the beginning that this method is an improvement on the previous method of bisection. This is not always true. The bisection method has a constant convergence rate, each iteration halving the search space. However, the convergence of the false position method depends on the function  $f(x)$  and the relative position of the initial points  $(a, f(a))$  and  $(b, f(b))$  with respect to the root. This is why it is not always true that it converges faster than the bisection method. On the other hand, the calculation of the successive values of the point  $c$  requires more arithmetic operations in the false position method, making each iteration slower than in the case of bisection.

### 6.2.3. Newton-Raphson method

Newton's method is based on the expansion of a Taylor series function  $f(x)$  around a point  $x_0$ ,

It belongs to a family of methods widely used in numerical calculus. The idea in the case of Newton's method is to approximate the function for which one wishes to obtain the root, by means of the first term of Taylor's series. That is to say, to approximate locally  $f(x)$ , in the environment of  $x_0$  by the straight line,

$$f(x_0) + f'(x_0)(x - x_0)$$

This line is precisely the tangent line to the curve  $f(x)$  at the point  $x_0$  (figure 6.10).

The method consists of obtaining the intersection of this tangent line with the abscissa axis,

$$0 = f(x_0) + f'(x_0)(x - x_0)$$

and obtaining  $x$ ,

$$x = x_0 - \frac{f(x_0)}{f'(x_0)}$$

Next, the function is evaluated at the point obtained  $x \rightarrow f(x)$ . As in the previous

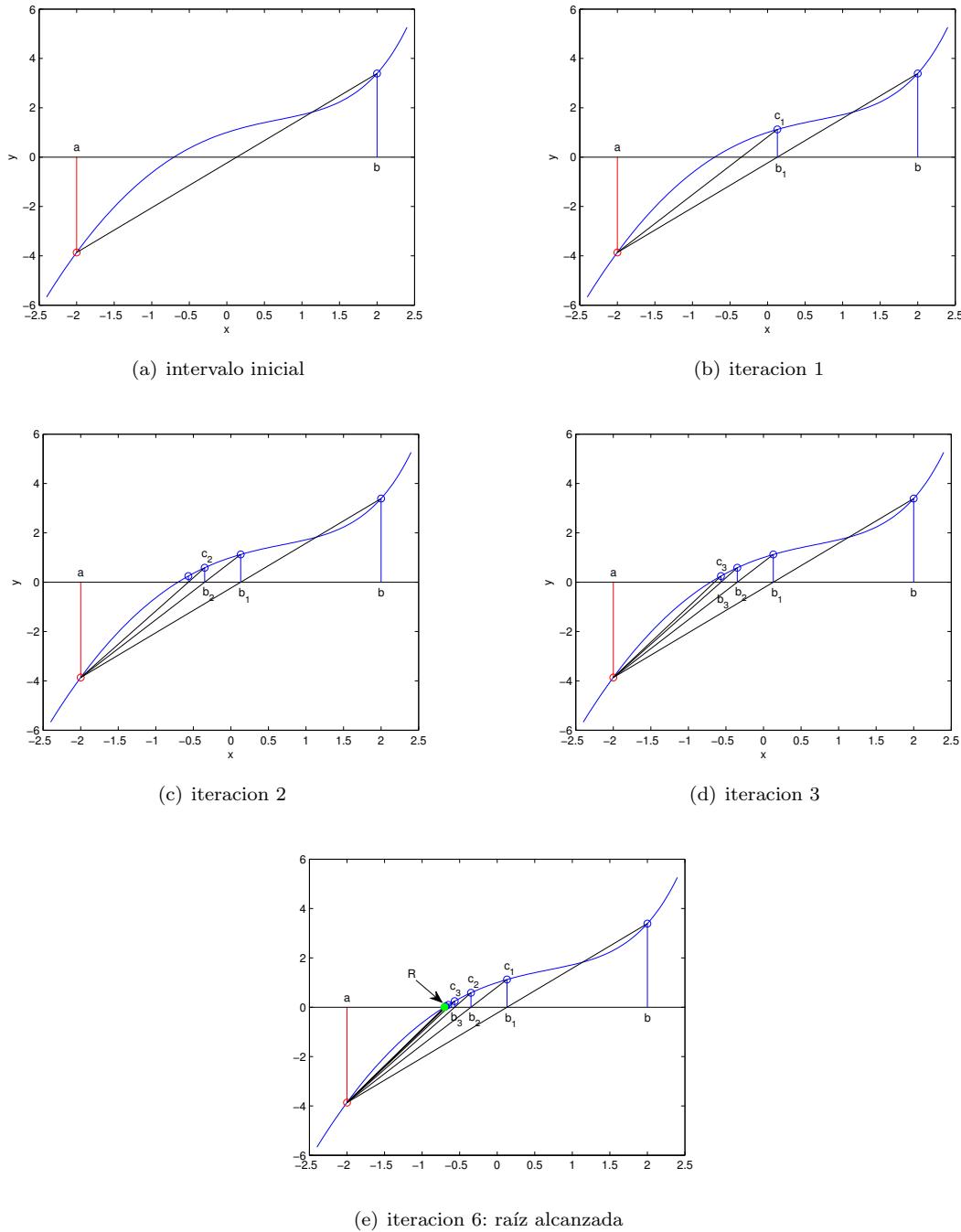


Figura 6.9: Proceso de obtención de la raíz de una función por el método de interpolación lineal  
 Figure 6.9: Process of obtaining the root of a function by the method of linear interpolation

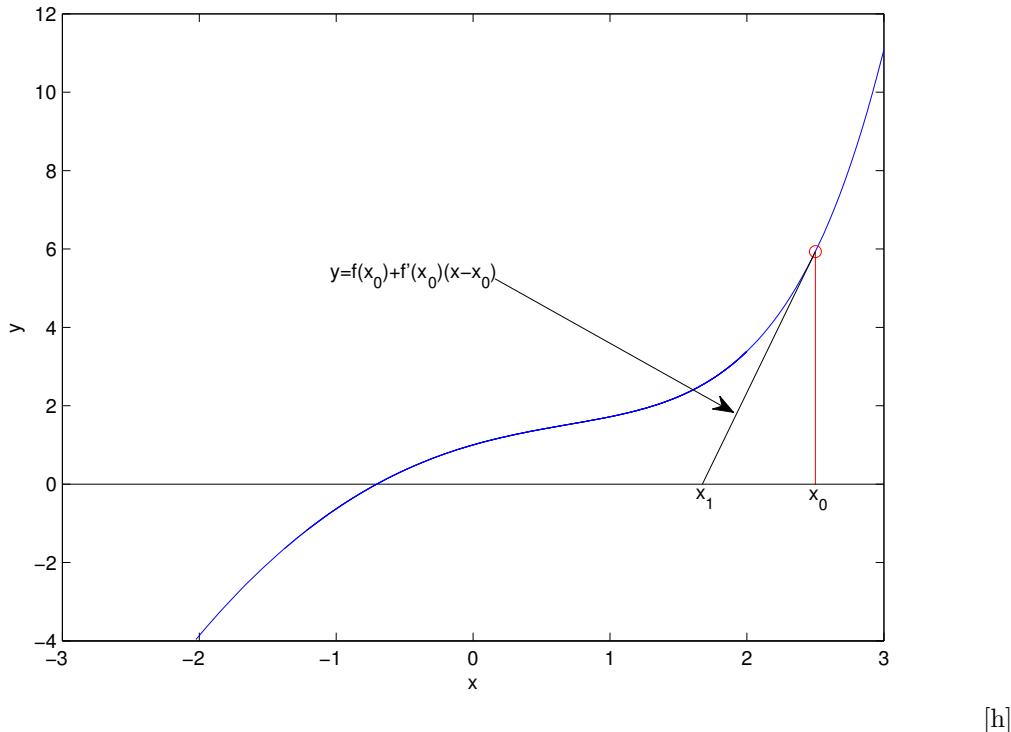


Figura 6.10: Recta tangente a la función  $f(x)$  en el punto  $x_0$   
 Figure 6.10: Tangent line to the function  $f(x)$  at the point  $x_0$ .

dos anteriores, se compara el valor de  $f(x)$  con una cierta tolerancia preestablecida. Si es menor, el valor de  $x$  se toma como raíz de la función. Si no, se vuelve aplicar el algoritmo, empleando ahora el valor de  $x$  que acabamos de obtener como punto de partida. Cada cálculo constituye una nueva iteración y los sucesivos valores obtenidos para  $x$ , convergen a la raíz,

methods, the value of  $f(x)$  is compared with a certain pre-set tolerance. If it is smaller, the value of  $x$  is taken as the root of the function. If not, the algorithm is applied again, now using the value of  $x$  just obtained as the starting point. Each calculation constitutes a new iteration and the successive values obtained for  $x$  converge to the root,

$$x_0 \rightarrow x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \rightarrow x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} \rightarrow \dots \rightarrow x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})} \rightarrow \dots$$

La figura 6.11 muestra un diagrama de flujo correspondiente al método de Newton. Si se compara con los diagramas de flujo de los algoritmos anteriores, el algoritmo de Newton resulta algo más simple de implementar. Sin embargo es preciso evaluar en cada iteración el valor de la función y el de su derivada.

El cálculo de la derivada, es el punto débil

Figure 6.12 shows a flowchart for Newton's method. Compared to the flowcharts of the previous algorithms, Newton's algorithm is somewhat simpler to implement. However, the value of the function and its derivative must be evaluated at each iteration.

The calculation of the derivative is the weak point of this algorithm, since for values  $x_0$  clo-

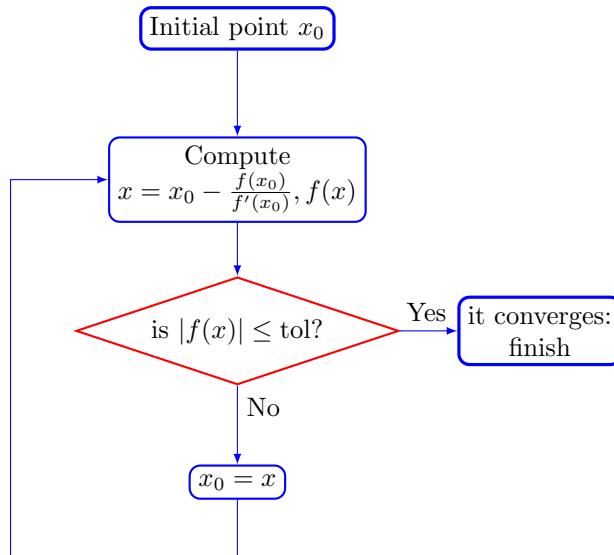


Figura 6.12: Flowchart of Newton-Raphson method

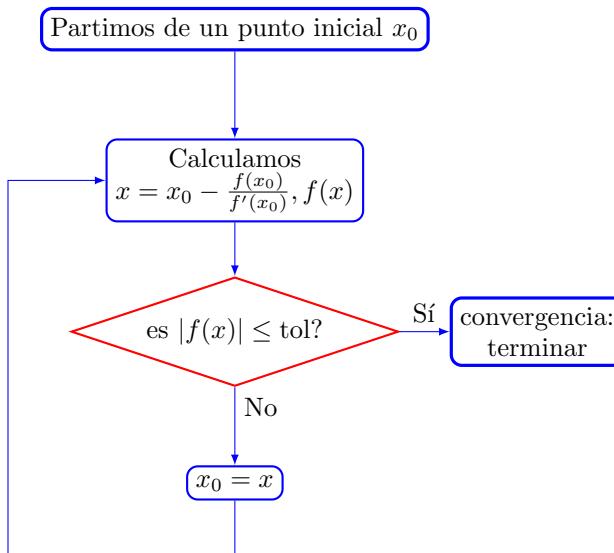


Figura 6.11: Diagrama de flujo del método de Newton-Raphson

de este algoritmo, ya que para valores  $x_0$  próximos a un mínimo o máximo local obtendremos valores de la derivada próximos a cero, lo que puede causar un error de desbordamiento al calcular el punto de corte de la recta tangente con el eje de abscisas o hacer que el algoritmo converja a una raíz alejada del punto inicial de búsqueda.

se to a local minimum or maximum we will obtain values of the derivative close to zero, which may cause an overflow error when calculating the point of intersection of the tangent line with the abscissa axis or cause the algorithm to converge to a root far from the initial search point.

La figura 6.13 muestra un ejemplo de obtención de la raíz de una función mediante el método de Newton. El método es más rápido que los dos anteriores, es decir, partiendo de una distancia comparable a la raíz, es el que converge en menos iteraciones.

En el ejemplo de la figura se ha obtenido la raíz para la misma función que en los ejemplos del método de la bisección e interpolación lineal. Se ha empezado sin embargo en un punto más alejado de la raíz, para que pueda observarse mejor en la figura la evolución del algoritmo. En cada uno de los gráficos que componen la figura pueden observarse los pasos del algoritmo: dado el punto  $x_i$ , se calcula la recta tangente a la función  $f(x)$  en el punto y se obtiene un nuevo punto  $x_{i+1}$ , como el corte de dicha recta tangente con el eje de abscisas.

En este ejemplo el algoritmo converge en las cinco iteraciones que se muestran en la figura, para la misma tolerancia empleada en los métodos anteriores,  $tol = 0.01$ . El punto de inicio empleado fue  $x_0 = 2.5$ , por tanto está fuera del intervalo  $[-2, 2]$  y más alejado de la raíz que en el caso de los métodos anteriores.

#### 6.2.4. Método de la secante

El método de la secante podría considerarse una variante del método de newton en el que se sustituye la recta tangente al punto  $x_0$  por la recta secante que une dos puntos obtenidos en iteraciones sucesivas. La idea es *aproximar* la derivada a la función  $f$  en el punto  $x_n$  por la pendiente de una recta secante, es decir de una recta que corta a la función en dos puntos,

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$$

Las sucesivas aproximaciones a la raíz de la función se obtienen de modo similar a las del método de Newton, simplemente sustituyendo la derivada de la función por su valor aproximado,

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \approx x_n - \frac{(x_n - x_{n-1}) \cdot f(x_n)}{f(x_n) - f(x_{n-1})}$$

Figure 6.13 shows an example of obtaining the root of a function using Newton's method. The method is faster than the previous two, i.e., starting from a comparable distance to the root, it is the one that converges in the fewest iterations.

In the example in the figure, the root has been obtained for the same function as in the examples of the bisection and false position method. However, we have started at a point further away from the root, so that the evolution of the algorithm can be better observed in the figure. In each of the graphs that make up the figure, the steps of the algorithm can be observed: given the point  $x_i$ , the tangent line to the function  $f(x)$  at the point is calculated and a new point  $x_{i+1}$  is obtained, as the intersection of this tangent line with the abscissa axis.

In this example the algorithm converges in the five iterations shown in the figure, for the same tolerance used in the previous methods,  $tol = 0.01$ . The starting point used was  $x_0 = 2.5$ , so it is outside the interval  $[-2, 2]$  and further away from the root than in the case of the previous methods.

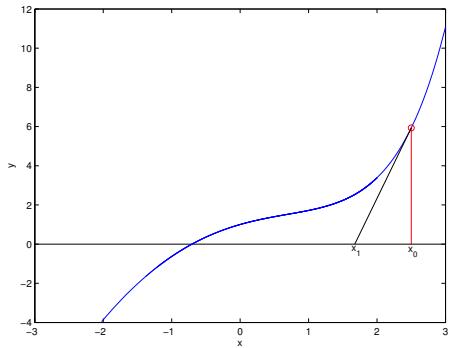
#### 6.2.4. Secant method

The secant method could be considered a variant of Newton's method in which the tangent line to the point  $x_0$  is replaced by the secant line joining two points obtained in successive iterations. The idea is to approximate the derivative of the function  $f$  at the point  $x_n$  by the slope of a secant line, i.e. a line that cuts the function at two points,

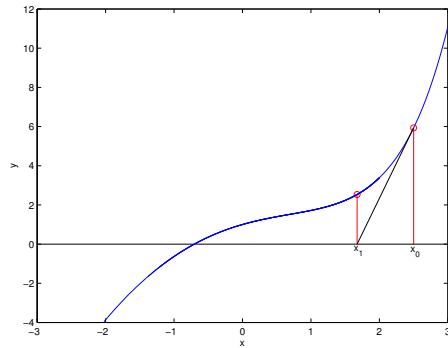
$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$$

The successive approximations to the root of the function are obtained in a similar way to Newton's method, by simply substituting the derivative of the function by its approximate value,

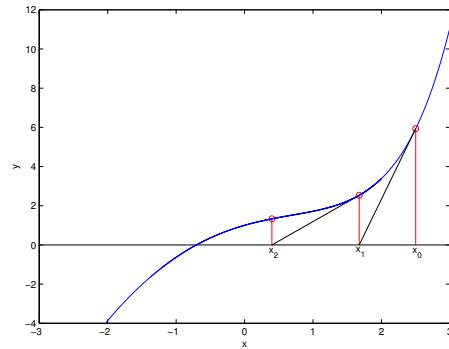
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \approx x_n - \frac{(x_n - x_{n-1}) \cdot f(x_n)}{f(x_n) - f(x_{n-1})}$$



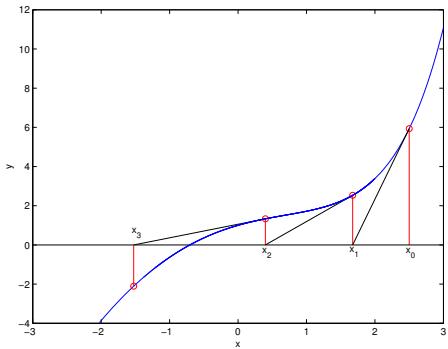
(a) intervalo inicial



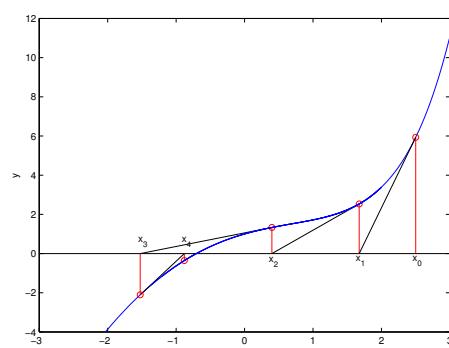
(b) iteracion 1



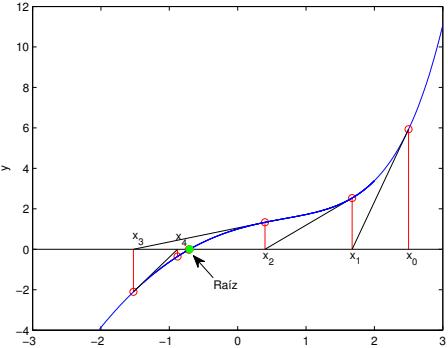
(c) iteracion 2



(d) iteracion 3



(e) iteracion 4



(f) iteracion 5: raíz de la función

Figura 6.13: Proceso de obtención de la raíz de una función por el método de Newton  
 Figure 6.13: Process of obtaining the root of a function by Newton's method

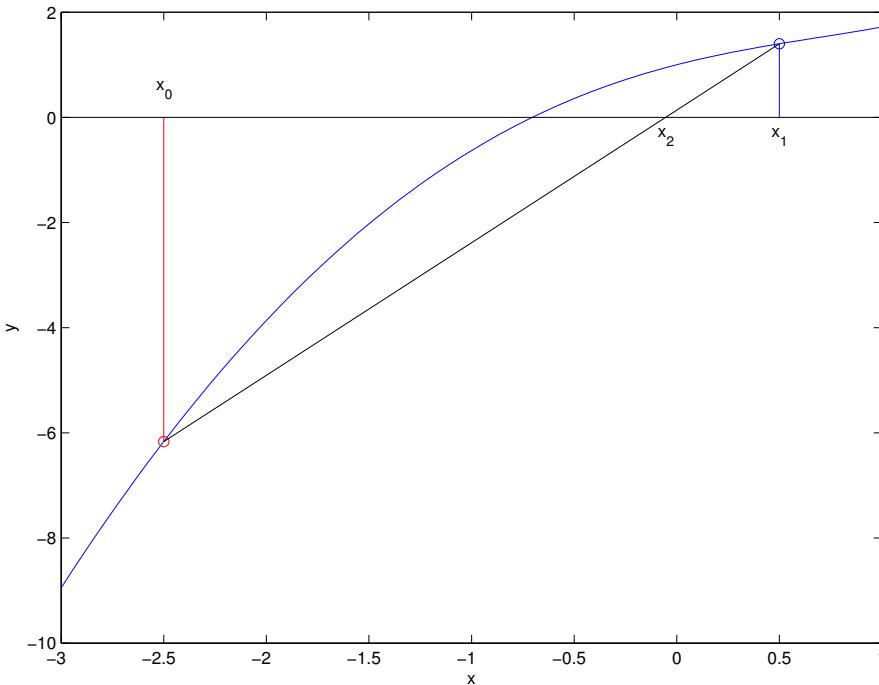


Figura 6.14: Recta secante a la función  $f(x)$  en los puntos  $x_0$  y  $x_1$   
 Figure 6.14: Secant line to the function  $f(x)$  at points  $x_0$  and  $x_1$ .

Para iniciar el algoritmo, es preciso emplear en este caso dos puntos iniciales. La figura 6.14 muestra un ejemplo.

El método podría en este punto confundirse con el de interpolación, sin embargo tiene dos diferencias importantes: En primer lugar, la elección de los dos puntos iniciales  $x_0$  e  $x_1$ , no tienen por qué formar un intervalo que contenga a la raíz. Es decir, podrían estar ambos situados al mismo lado de la raíz. En segundo lugar, los puntos obtenidos se van sustituyendo por orden, de manera que la nueva recta secante se construye siempre a partir de los dos últimos puntos obtenidos, sin prestar atención a que el valor de la raíz esté contenido entre ellos. (No se comparan los signos de la función en los puntos para ver cual se sustituye, como en el caso del método de interpolación).

La figura 6.15 muestra un diagrama de flujo para el método de la secante. El diagrama es básicamente el mismo que el empleado para el método de Newton. Las dos diferencias fun-

To start the algorithm, two starting points must be used in this case. Figure 6.14 shows an example.

The method could at this point be confused with false position method, but it has two important differences: Firstly, the choice of the two initial points  $x_0$  and  $x_1$ , need not form an interval containing the root. That is, they could both be located on the same side of the root. Secondly, the points obtained are substituted in order, so that the new secant line is always constructed from the last two points obtained, without paying attention to whether the value of the root is contained between them. (The signs of the function at the points are not compared to see which one is substituted, as in the case of the interpolation method).

Figure 6.16 shows a flow diagram for the secant method. The diagram is basically the same as the one used for Newton's method. The two fundamental differences are that now,

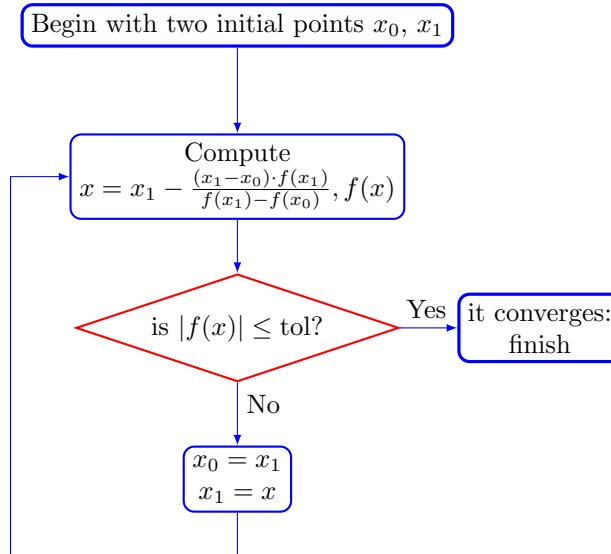


Figura 6.16: Flowchart of the secant method

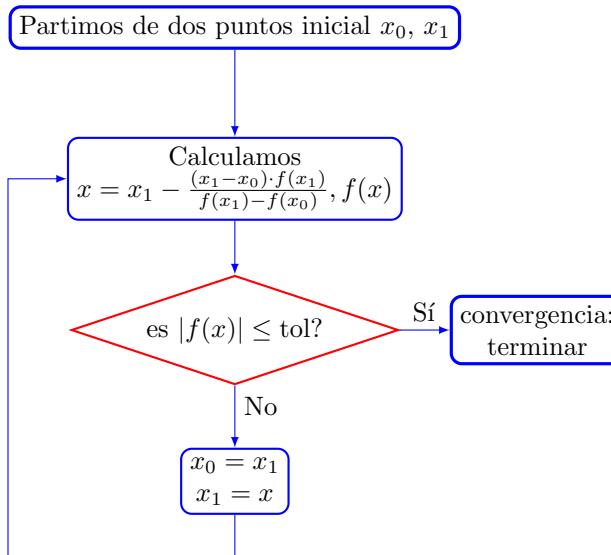


Figura 6.15: Diagrama de flujo del método de la secante

damentales son, que ahora en lugar de evaluar la función y la derivada en cada iteración, se calcula el valor del punto de corte de la recta que pasa por los dos últimos puntos obtenidos (es decir, empleamos una recta secante, que corta a la curva en dos puntos, en lugar de emplear una recta tangente).

Además es preciso actualizar, en cada iteración, el valor de los dos últimos puntos obtenidos: el más antiguo se desecha, el punto recién obtenido sustituye al anterior y éste al obtenido dos iteraciones antes.

instead of evaluating the function and the derivative at each iteration, we calculate the value of the cut-off point of the line passing through the last two points obtained (i.e. we use a secant line, which cuts the curve at two points, instead of using a tangent line).

In addition, the value of the last two points obtained must be updated in each iteration: the oldest point is discarded, the newly obtained point replaces the previous one and this one replaces the one obtained two iterations before.

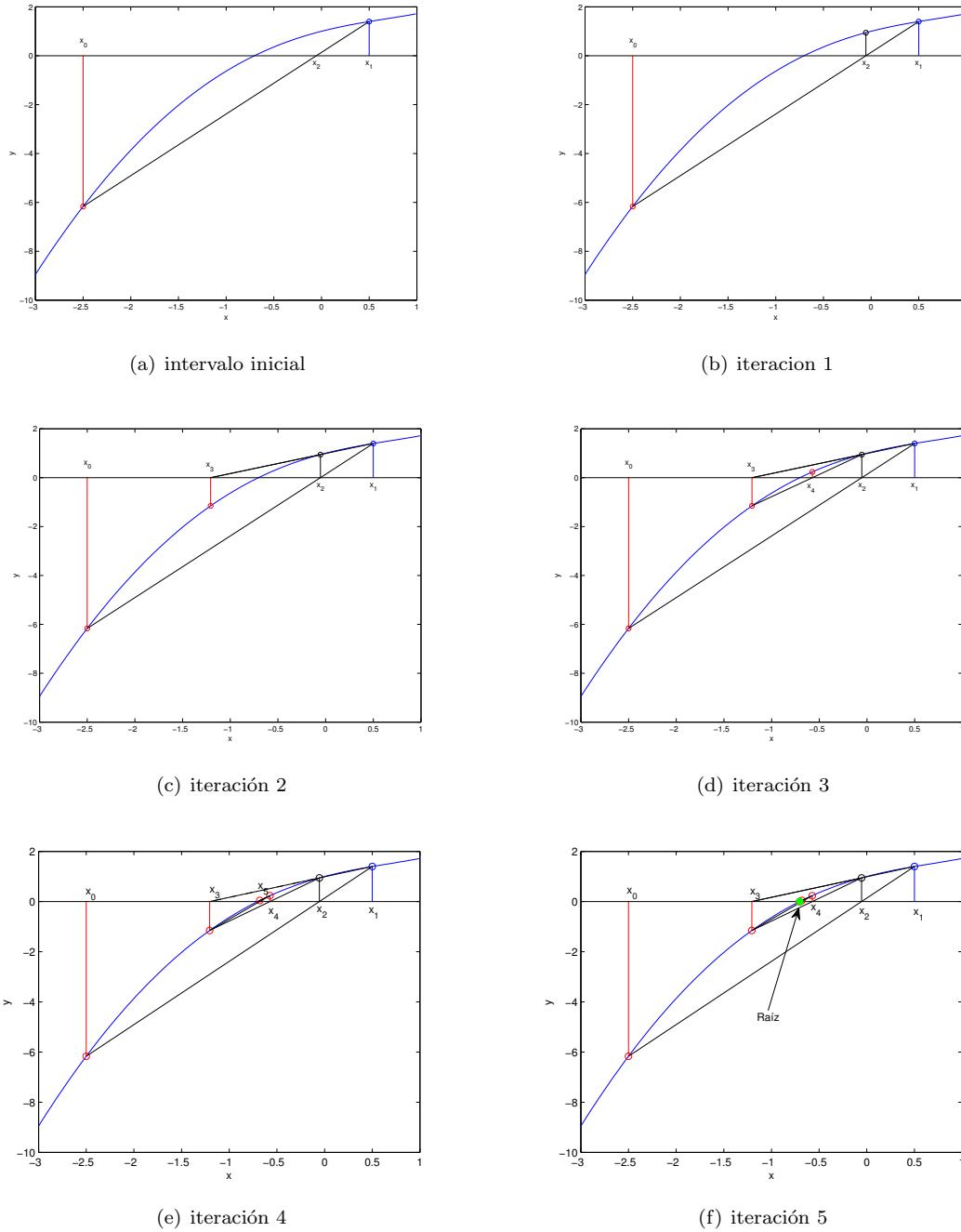


Figura 6.17: proceso de obtención de la raíz de una función por el método de la secante  
process of obtaining the root of a function by the secant method

La figura 6.17 muestra un ejemplo de la obtención de una raíz por el método de la secante. Se ha empleado de nuevo la misma función que en los ejemplos anteriores, tomando como valores iniciales,  $x_0 = -2.5$  y  $x_1 = 0.5$ . La tolerancia se ha fijado en  $tol = 0.01$  también como en los anteriores algoritmos descritos. En este caso, el algoritmo encuentra la raíz en cinco iteraciones. Cada uno de los gráficos que compone la figura 6.17, muestra la obtención de un nuevo punto a partir de los dos anteriores.

En la iteración 2, puede observarse como el nuevo punto se obtiene a partir de dos puntos que están ambos situados a la derecha de la raíz, es decir, no forman un intervalo que contenga a la raíz. Aquí se pone claramente de manifiesto la diferencia con el método de interpolación lineal. De hecho, como ya se ha dicho, el método de la secante puede iniciarse tomando los dos primeros puntos a uno de los lados de la raíz.

El método es, en principio, más eficiente que el de la bisección y el de interpolación lineal, y menos eficiente que el de Newton.

La ventaja de este método respecto al de Newton es que evita tener que calcular explícitamente la derivada de la función para la que se quiere calcular la raíz. El algunos casos, la obtención de la forma analítica de dicha derivada puede ser compleja.

### 6.2.5. Método de las aproximaciones sucesivas o del punto fijo

El método del punto fijo es, como se verá a lo largo de esta sección, el más sencillo de programar de todos. Desafortunadamente, presenta el problema de que no podemos aplicarlo a todas las funciones. Hay casos en los que el método no converge, con lo que no es posible emplearlo para encontrar la raíz o raíces de una función.

**Punto fijo de una función.** Se dice que un punto  $x_f$  es un punto fijo de una función  $g(x)$  si se cumple,

$$g(x_f) = x_f$$

Figure 6.17 shows an example of obtaining a root by the secant method. The same function has been used again as in the previous examples, taking as initial values,  $x_0 = -2.5$  and  $x_1 = 0.5$ . As in the previous algorithms described, the tolerance has been set to  $tol = 0.01$ . In this case, the algorithm finds the root in five iterations. Each of the graphs that make up the figure 6.17, shows the obtaining of a new point from the previous two.

In iteration 2, it can be seen how the new point is obtained from two points that are both located to the right of the root, i.e. they do not form an interval containing the root. Here the difference with the linear interpolation method is clearly shown. In fact, as already mentioned, the secant method can be started by taking the first two points on either side of the root.

The method is, in principle, more efficient than bisection and false position, and less efficient than Newton's method.

The advantage of this method over Newton's is that it avoids having to explicitly calculate the derivative of the function for which the root is to be calculated. In some cases, obtaining the analytical form of this derivative can be complex.

### 6.2.5. Method of successive approximations or fixed point iteration

The fixed point iteration is, as will be seen throughout this section, the simplest of all to program. Unfortunately, it has the problem that we cannot apply it to all functions. There are cases in which the method does not converge, so it is not possible to use it to find the root or roots of a function.

**Fixed point.**  $x_f$  is a fixed point of  $g(x)$  if it holds,

$$g(x_f) = x_f$$

That is, the image of the fixed point  $x_f$  is again the fixed point. So for example the

Es decir, la imagen del punto fijo  $x_f$  es de nuevo el punto fijo. Así por ejemplo la función,

$$g(x) = -\sqrt{e^x}$$

Tiene un punto fijo en  $x_f = -0.703467$ , porque  $g(-0.703467) = -0.703467$ . La existencia de un punto fijo puede obtenerse gráficamente, representando en un mismo gráfico la función  $y = g(x)$  y la recta  $y = x$ . Si existe un punto de corte entre ambas gráficas, se trata de un punto fijo. La figura 6.18, muestra gráficamente el punto fijo de la función  $g(x) = -\sqrt{e^x}$  del ejemplo anterior.

Una función puede tener uno o más puntos fijos o no tener ninguno. Por ejemplo, la función  $y = \sqrt{e^x}$  no tiene ningún punto fijo.

function,

$$g(x) = -\sqrt{e^x}$$

Has a fixed point  $x_f = -0.703467$ , because  $g(-0.703467) = -0.703467$ . The existence of a fixed point can be obtained graphically, representing in the same graph the function  $y = g(x)$  and the straight line  $y = x$ . If there is a point of intersection between both graphs, it is a fixed point. The figure 6.18, shows graphically the fixed point of the function  $g(x) = -\sqrt{e^x}$  of the previous example.

A function can have one or more fixed points or none at all. For example, the function  $y = \sqrt{e^x}$  does not have any fixed point.

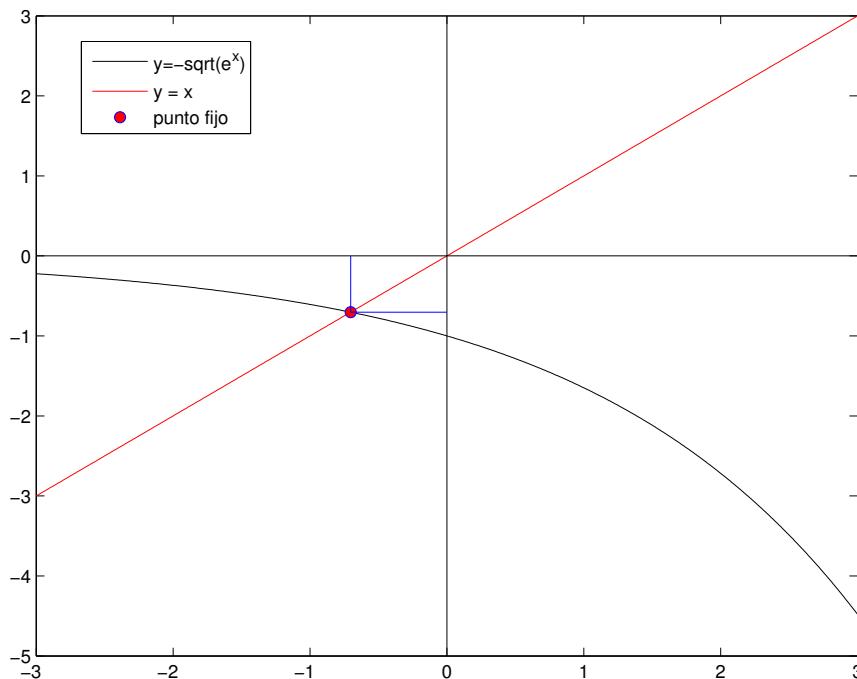


Figura 6.18: Obtención gráfica del punto fijo de la función,  $g(x) = -\sqrt{e^x}$   
 Figure 6.18: Obtaining the fixed point of the function graphically,  $g(x) = -\sqrt{e^x}$

**Punto fijo atractivo.** Supongamos ahora que, a partir de la función  $g(x)$  creamos la

**Attracting fixed point.** Suppose now that from the function  $g(x)$  we create the following

siguiente sucesión,

$$x_{n+1} = g(x_n)$$

Es decir, empezamos tomando un punto inicial  $x_0$  y a partir de él vamos obteniendo los siguientes valores de la sucesión como,

$$x_0 \rightarrow x_1 = g(x_0) \rightarrow x_2 = g(x_1) = g(g(x_0)) \rightarrow \cdots \rightarrow x_{n+1} = g(x_n) = g(g(\cdots(g(x_0)))) \rightarrow \cdots$$

Decimos que un punto fijo  $x_f$  de la función  $g(x)$  es un punto fijo atractivo si la sucesión  $x_{n+1} = g(x_n)$  converge al valor  $x_f$ , siempre que  $x_0$  se tome *suficientemente* cercano a  $x_f$ . Cómo de cerca tienen que estar  $x_0$  y  $x_f$  para que la serie converja, es una cuestión delicada. De entrada, es importante descartar que hay funciones que tienen puntos fijos no atractivos, por ejemplo, la función  $g(x) = x^2$  tiene dos puntos fijos  $x = 0$  y  $x = 1$ . El primero es el límite de la sucesión  $x_{n+1} = g(x_n)$  para cualquier valor inicial  $x_0$  contenido en el intervalo abierto  $(-1, 1)$ . El punto  $x = 1$  resulta inalcanzable para cualquier sucesión excepto que el punto de inicio sea él mismo  $x_0 = x_f = 1$ .

Hay algunos casos en los que es posible, para determinadas funciones, saber cuando uno de sus puntos fijos es atractivo,

**Teorema de existencia y unicidad del punto fijo.** Dada una función  $g(x)$ , continua y diferenciable en un intervalo  $[a, b]$ , si se cumple que,  $\forall x \in [a, b] \Rightarrow g(x) \in [a, b]$ , entonces  $g(x)$  tiene un punto fijo en el intervalo  $[a, b]$ .

Si además existe una constante positiva  $k < 1$  y se cumple que la derivada  $|g'(x)| \leq k$ ,  $\forall x \in (a, b)$ , entonces el punto fijo contenido en  $[a, b]$  es único.

Para demostrar la primera parte del teorema, se puede emplear el teorema de Bolzano. Si se cumple que  $g(a) = a$  o que  $g(b) = b$ , entonces  $a$  o  $b$  serían el punto fijo. Supongamos que no es así; entonces tiene que cumplirse que  $g(a) > a$  y que  $g(b) < b$ . Si construimos una función,  $f(x) = g(x) - x$  esta función, que es continua por construcción, cumple que  $f(a) = g(a) - a > 0$  y  $f(b) = g(b) - b < 0$ . Pero entonces, debe existir un punto,  $x_f \in [a, b]$

sequence,

$$x_{n+1} = g(x_n)$$

That is, we start by taking an initial point  $x_0$  and from it we obtain the following values of the sequence as,

We can say that fixed point  $x_f$  of function  $g(x)$  is an attracting fixed point if the succession  $x_{n+1} = g(x_n)$  converges to  $x_f$ , if  $x_0$  is close enough to  $x_f$ . How close  $x_0$  and  $x_f$  have to be for the series to converge is a delicate question. For example, the function  $g(x) = x^2$  has two fixed points  $x = 0$  and  $x = 1$ . The first is the limit of the sequence  $x_{n+1} = g(x_n)$  for any initial value  $x_0$  contained in the open interval  $(-1, 1)$ . The point  $x = 1$  is unreachable for any sequence unless the starting point is itself  $x_0 = x_f = 1$ .

There are some cases where it is possible, for certain functions, to know when one of your fixed points is attractive,

**Fixed point theorem.** Let  $g(x)$ , be a continuous and derivable function in  $[a, b]$ , if,  $\forall x \in [a, b] \Rightarrow g(x) \in [a, b]$ , then  $g(x)$  has at least a fixed point in  $[a, b]$ .

If, in addition, there is a positive constant  $k < 1$  and it holds that  $|g'(x)| \leq k$ ,  $\forall x \in (a, b)$ , then  $g(x)$  has a unique fixed point in  $[a, b]$ .

To prove the first part of the theorem, Bolzano's theorem can be used. If it is satisfied that  $g(a) = a$  or  $g(b) = b$ , then  $a$  or  $b$  would be the fixed point. Suppose this is not the case; then it must be true that  $g(a) > a$  and that  $g(b) < b$ . Define  $f(x) = g(x) - x$ . This function, which is continuous by construction, satisfies that  $f(a) = g(a) - a > 0$  and  $f(b) = g(b) - b < 0$ . Then, there is a  $x_f \in [a, b]$  that holds  $f(x_f) = 0$  and therefore  $f(x_f) = g(x_f) - x_f = 0 \Rightarrow g(x_f) = x_f$ . That is,  $x_f$  is a fixed point of  $g(x)$ .

The second part of the theorem can be proved using the mean value theorem. If we assume that there are two distinct fixed points

para el cual  $f(x_f) = 0$  y, por tanto,  $f(x_f) = g(x_f) - x_f = 0 \Rightarrow g(x_f) = x_f$ . Es decir,  $x_f$  es un punto fijo de  $g(x)$ .

La segunda parte del teorema puede demostrarse empleando el teorema de valor medio. Si suponemos que existen dos puntos fijos distintos  $x_{f1} \neq x_{f2}$  en el intervalo  $[a, b]$ , según el teorema del valor medio, existe un punto  $\xi$  comprendido entre  $x_{f1}$  y  $x_{f2}$  para el que se cumple,

$$\frac{g(x_{f1}) - g(x_{f2})}{x_{f1} - x_{f2}} = g'(\xi)$$

$$|g(x_{f1}) - g(x_{f2})| = |x_{f1} - x_{f2}| \cdot |g'(\xi)| \leq |x_{f1} - x_{f2}| \cdot k < |x_{f1} - x_{f2}|$$

Pero como se trata de puntos fijos  $|g(x_{f1}) - g(x_{f2})| = |x_{f1} - x_{f2}|$ , con lo que llegaríamos al resultado contradictorio,

$$|x_{f1} - x_{f2}| = |g(x_{f1}) - g(x_{f2})| \leq |x_{f1} - x_{f2}| \cdot k < |x_{f1} - x_{f2}|$$

Salvo que, en contra de la hipótesis inicial, se cumpla que  $x_{f1} = x_{f2}$ . En cuyo caso, solo puede existir un único punto fijo en el intervalo  $[a, b]$  bajo las condiciones impuestas por el teorema.

**Teorema de punto fijo (atractivo).** <sup>2</sup> Dada una función  $g(x)$ , continua y diferenciable en un intervalo  $[a, b]$ , que cumple que,  $\forall x \in [a, b] \Rightarrow g(x) \in [a, b]$  y que  $|g'(x)| \leq k$ ,  $\forall x \in (a, b)$ , con  $0 < k < 1$ , entonces se cumple que, para cualquier punto inicial  $x_0$ , contenido en el intervalo  $[a, b]$ , la sucesión  $x_{n+1} = g(x_n)$  converge al único punto fijo del intervalo  $[a, b]$ . La demostración puede obtenerse de nuevo a partir del teorema del valor medio. Si lo aplicamos al valor inicial  $x_0$  y al punto fijo  $x_f$ , obtenemos,

Para la siguiente iteración tendremos,

$$|g(x_1) - g(x_f)| \leq |x_1 - x_f| \cdot k \leq |x_0 - x_f| \cdot k^2$$

puesto que,  $x_1 = g(x_0)$  y  $x_f = g(x_f)$ , puesto que  $x_f$  es el punto fijo.

Por simple inducción tendremos que para el término enésimo de la sucesión,

<sup>2</sup>Hay varios teoremas de punto fijo definidos en distintos contextos matemáticos. Aquí se da una versión reducida a funciones  $f(x) : \mathbb{R} \rightarrow \mathbb{R}$

$x_{f1} \neq x_{f2}$  in  $[a, b]$ , according to the mean value theorem, there exists a point  $\xi$  between  $x_{f1}$  and  $x_{f2}$  for which it is satisfied,

$$\frac{g(x_{f1}) - g(x_{f2})}{x_{f1} - x_{f2}} = g'(\xi)$$

Therefore,

But since we are dealing with fixed points  $|g(x_{f1}) - g(x_{f2})| = |x_{f1} - x_{f2}|$ , this would lead to the contradictory result

Unless, contrary to the initial hypothesis, it is satisfied that  $x_{f1} = x_{f2}$ . In which case, there can only be a single fixed point in the interval  $[a, b]$  under the conditions imposed by the theorem.

**Theorem of the attracting fixed point.**

<sup>2</sup> Let  $g(x)$ , continuous and differentiable on an interval  $[a, b]$ , that holds,  $\forall x \in [a, b] \Rightarrow g(x) \in [a, b]$  and that  $|g'(x)| \leq k$ ,  $\forall x \in (a, b)$ , with  $0 < k < 1$ , then it holds that for any  $x_0$ , in  $[a, b]$ , the succession  $x_{n+1} = g(x_n)$  converges to the only fixed point of the interval  $[a, b]$ . The proof can again be obtained from the mean value theorem. If we apply it to the initial value  $x_0$  and the fixed point  $x_f$ , we obtain,

$$|g(x_0) - g(x_f)| = |x_0 - x_f| \cdot |g'(\xi)| \leq |x_0 - x_f| \cdot k$$

For the next iteration we will have,

$$|g(x_1) - g(x_f)| \leq |x_1 - x_f| \cdot k \leq |x_0 - x_f| \cdot k^2$$

since  $x_1 = g(x_0)$  y  $x_f = g(x_f)$ , because  $x_f$  is a fixed point.

<sup>2</sup>There are several fixed point theorems defined in different mathematical contexts. A version reduced to functions  $f(x) : \mathbb{R} \rightarrow \mathbb{R}$  is given here.

Pero

$$|g(x_n) - g(x_f)| \leq |x_{n-1} - x_f| \cdot k \leq |x_{n-2} - x_f| \cdot k^2 \leq \cdots \leq |x_0 - x_f| \cdot k^n$$

| But,

$$\lim_{n \rightarrow \infty} k^n = 0 \Rightarrow \lim_{n \rightarrow \infty} |x_n - x_f| \leq \lim_{n \rightarrow \infty} |x_0 - x_f| k^n = 0$$

Es decir, la sucesión  $x_{n+1} = g(x_n)$  converge al punto fijo  $x_f$ .

**El método del punto fijo.** Como ya hemos visto, obtener una raíz de una función  $f(x)$ , consiste en resolver la ecuación  $f(x) = 0$ . Supongamos que podemos descomponer la función  $f(x)$  como la diferencia de dos términos, una función auxiliar,  $g(x)$ , y la propia variable  $x$

$$f(x) = g(x) - x$$

Encontrar una raíz de  $f(x)$  resulta entonces equivalente a buscar un punto fijo de  $g(x)$ .

$$f(x) = 0 \rightarrow g(x) - x = 0 \rightarrow g(x) = x$$

En general, a partir de una función dada  $f(x)$ , es posible encontrar distintas funciones  $g(x)$  que cumplan que  $f(x) = g(x) - x$ . No podemos garantizar que cualquiera de las descomposiciones que hagamos nos genere una función  $g(x)$  que tenga un punto fijo. Además, para funciones que tengan más de una raíz, puede suceder que distintas descomposiciones de la función converjan a distintas raíces. Si podemos encontrar una que cumpla las condiciones del teorema de punto fijo que acabamos de enunciar, en un entorno de una raíz de  $f(x)$ , podemos desarrollar un método que obtenga iterativamente los valores de la sucesión  $x_{n+1} = g(x_n)$ , a partir de un valor inicial  $x_0$ . El resultado se aproximará al punto fijo de  $g(x)$ , y por tanto a la raíz de  $f(x)$  tanto como queramos. Bastará, como en los métodos anteriores, definir un valor (tolerancia), por debajo del cual consideraremos que el valor obtenido es suficientemente próximo a la raíz como para darlo por válido.

La figura 6.19 muestra un diagrama de flujo del método del punto fijo.

Using induction we can obtain the ntn term of the sequence,

$$| \quad \text{But,}$$

That is, the sequence  $x_{n+1} = g(x_n)$  converges to the fixed point  $x_f$ .

**Fixed point iteration.** As we have already seen, obtaining a root of a function  $f(x)$ , consists of solving the equation  $f(x) = 0$ . Suppose we can decompose the function  $f(x)$  as the difference of two terms, an auxiliary function,  $g(x)$ , and the variable  $x$  itself.

$$f(x) = g(x) - x$$

Finding a root of  $f(x)$  is then equivalent to finding a fixed point of  $g(x)$ .

$$f(x) = 0 \rightarrow g(x) - x = 0 \rightarrow g(x) = x$$

In general, starting from a given function  $f(x)$ , it is possible to find different functions  $g(x)$  that satisfy that  $f(x) = g(x) - x$ . We cannot guarantee that any of the decompositions we do will generate a function  $g(x)$  that has a fixed point. Moreover, for functions that have more than one root, it can happen that different decompositions of the function converge to different roots. If we can find one that satisfies the conditions of the fixed point theorem just stated, in an environment of a root of  $f(x)$ , we can develop a method that iteratively obtains the values of the sequence  $x_{n+1} = g(x_n)$ , starting from an initial value  $x_0$ . The result will approximate the fixed point of  $g(x)$ , and therefore the root of  $f(x)$  as much as we want. It will suffice, as in the previous methods, to define a value (tolerance), below which we consider that the value obtained is sufficiently close to the root to be valid.

Figure 6.19 shows a flowchart of the fixed point iteration.

The idea is to carefully choose the initial point  $x_0$ , to ensure that it lies within the interval of convergence of the fixed point. Then,

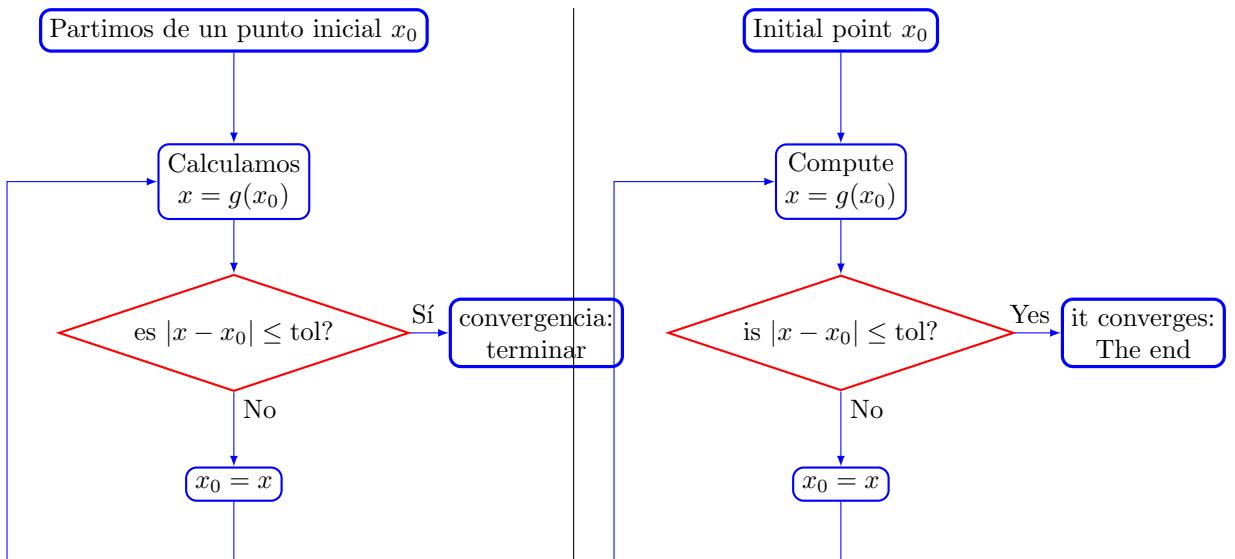


Figura 6.19: Diagrama de flujo del método del punto fijo. Nótese que la raíz obtenida corresponde a la función  $f(x) = g(x) - x$

La idea es elegir cuidadosamente el punto inicial  $x_0$ , para asegurar que se encuentra dentro del intervalo de convergencia del punto fijo. A continuación, calculamos el valor de  $g(x_0)$ , el resultado será un nuevo valor  $x$ . Comprobamos la diferencia entre el punto obtenido y el anterior y si es menor que una cierta tolerancia, consideramos que el método ha convergido, dejamos de iterar, y devolvemos el valor de  $x$  obtenido como resultado. Si no, copiamos  $x$  en  $x_0$  y volvemos a empezar todo el proceso. Es interesante hacer notar que el algoritmo converge cuando la diferencia entre dos puntos consecutivos es menor que un cierto valor. De acuerdo con la condición de punto fijo  $g(x_0) = x_0$ , dicha distancia, sería equivalente a la que media entre  $f(x_0) = g(x_0) - x_0$ , la función para la que queremos obtener la raíz, y 0.

Veamos un ejemplo. Supongamos que queremos calcular por el método del punto fijo la raíz de la función  $y = e^x - x^2$ , que hemos empleado en los ejemplos de los métodos anteriores.

En primer lugar, debemos obtener a partir de ella una nueva función que cumpla que  $f(x) = g(x) - x$ . Podemos hacerlo de varias

Figura 6.19: Flowchart of fixed point iteration. Root corresponds to  $f(x) = g(x) - x$

we calculate the value of  $g(x_0)$ , the result will be a new value  $x$ . We check the difference between the obtained point and the previous one and if it is less than a certain tolerance, we consider that the method has converged, we stop iterating, and we return the value of  $x$  obtained as a result. If not, we copy  $x$  into  $x_0$  and start the whole process again. It is interesting to note that the algorithm converges when the difference between two consecutive points is less than a certain value. According to the fixed point condition  $g(x_0) = x_0$ , this distance would be equivalent to that between  $f(x_0) = g(x_0) - x_0$ , the function for which we want to obtain the root, and 0.

Let us look at an example. Suppose we want to calculate by the fixed point method the root of the function  $y = e^x - x^2$ . function  $y = e^x - x^2$ , which we have used in the examples of the previous methods.

First, we must obtain from it a new function that satisfies that  $f(x) = g(x) - x$ . We can do this in several ways by clearing an ' $x$ ' from the equation  $e^x - x^2 = 0$ . To illustrate the different cases of convergence, we will clear  $x$  in three different ways.

maneras despejando una ' $x$ ', de la ecuación  $e^x - x^2 = 0$ . Para ilustrar los distintos casos de convergencia, despejaremos  $x$  de tres maneras distintas .

$$e^x - x^2 = 0 \Rightarrow \begin{cases} x = \pm\sqrt{e^x} \\ x = \ln(x^2) = 2 \cdot \ln(|x|) \\ x = \frac{e^x}{x} \end{cases}$$

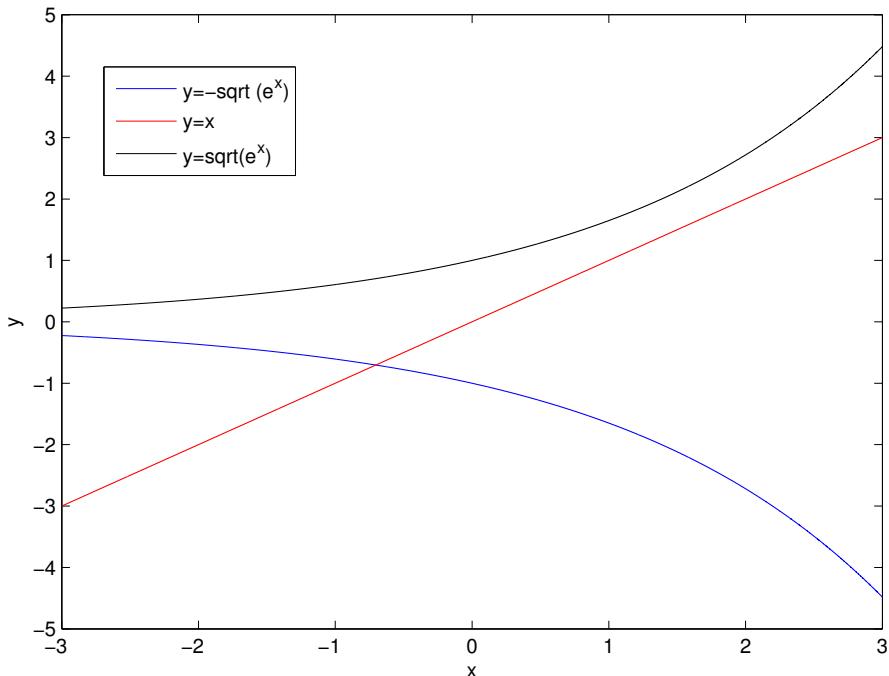


Figura 6.20:  $g(x) = \pm\sqrt{e^x}$ , Solo la rama negativa tiene un punto fijo.

Figure 6.20:  $g(x) = \pm\sqrt{e^x}$ , only negative branch has a fixed point.

En nuestro ejemplo hemos obtenido tres formas distintas de *despejar* la variable  $x$ . La cuestión que surge inmediatamente es, si todas las funciones obtenidas, tienen un punto fijo y, en caso de tenerlo, si es posible alcanzarlo iterativamente.

En el primer caso,  $x = \pm\sqrt{e^x}$ , obtenemos las dos ramas de la raíz cuadrada. Cada una de ellas constituye a los efectos de nuestro cálculo una función distinta. Si las dibujamos junto a la recta  $y = x$  (figura 6.20), observamos que

In our example we have obtained three different ways of *clearing* the variable  $x$ . The question that immediately arises is whether all the functions obtained have a fixed point and, if so, whether it is possible to reach it iteratively.

In the first case,  $x = \pm\sqrt{e^x}$ , we obtain the two branches of the square root. For the purposes of our calculation, each of them constitutes a different function. If we draw them next to the straight line  $y = x$  (figure 6.20), we

solo la rama negativa la corta. Luego será esta rama  $g(x) = -\sqrt{e^x}$ , la que podremos utilizar para obtener la raíz de la función original por el método del punto fijo. La rama positiva, al no cortar a la recta  $y = x$  en ningún punto, es una función que carece de punto fijo.

No es difícil demostrar, que la función  $g(x) = -\sqrt{e^x}$  cumple las condiciones del teorema de punto fijo descrito más arriba para el intervalo  $(-\infty, 0]$ . Luego el algoritmo del punto fijo debería converger para cualquier punto de inicio  $x_0$  contenido en dicho intervalo. De hecho, para esta función, el algoritmo converge desde cualquier punto de inicio (Si empezamos en punto positivo, el siguiente punto,  $x_1$  será negativo, y por tanto estará dentro del intervalo de convergencia). Esta función es un ejemplo de que el teorema suministra una condición suficiente, pero no necesaria para que un punto fijo sea atractivo.

La figura 6.21 muestra un ejemplo del cálculo de la raíz de la función  $f(x) = e^x - x^2$  empleando la función  $g(x) = -\sqrt{e^x}$ , para obtener el punto fijo. Se ha tomado como punto de partida  $x_0 = 2.5$ , un valor fuera del intervalo en el que se cumple el teorema. Como puede observarse en 6.21(a). A pesar de ello el algoritmo converge rápidamente, y tras 5 iteraciones, 6.21(f), ha alcanzado el punto fijo —y por tanto la raíz buscada—, con la tolerancia impuesta.

Si tratamos de emplear la función  $g(x) = \ln(x^2)$  para obtener la raíz, observamos que la función no cumple el teorema para ningún intervalo que contenga la raíz.

La figura 6.22 muestra la función  $g(x)$ , la recta  $y = x$  y la evolución del algoritmo tras cuatro evaluaciones. Es fácil deducir que el algoritmo saltará de la rama positiva a la negativa y de ésta volverá a saltar de nuevo a la positiva.

La función presenta una asíntota vertical en el 0. Si se empieza desde  $x_0 = 0$ ,  $x_0 = 1$  o  $x_0 = -1$  el algoritmo no converge, puesto que la función diverge hacia  $-\infty$ . Para el resto de los valores, la función oscila entre una rama y otra. Si en alguna de las oscilaciones acierta a pasar suficientemente cerca del punto fijo,  $x_n - x_{n-1} \leq tol$ , el algoritmo habrá aproximado la raíz, aunque propiamente no se puede decir

observe that only the negative branch cuts it. Then it will be this branch  $g(x) = -\sqrt{e^x}$ , which we will be able to use to obtain the root of the original function by the fixed point method. The positive branch, as it does not cut the straight line  $y = x$  at any point, is a function without a fixed point.

It is not difficult to show that the function  $g(x) = -\sqrt{e^x}$  satisfies the conditions of the fixed point theorem described above for the interval  $(-\infty, 0]$ . Then the fixed point iteration should converge for any starting point  $x_0$  contained in that interval. In fact, for this function, the algorithm converges from any starting point. (If we start at a positive point, the next point,  $x_1$  will be negative, and therefore within the interval of convergence). This function is an example of the theorem providing a sufficient, but not necessary condition for a fixed point to be attractive.

Figure 6.21 shows an example of the calculation of the root of the function  $f(x) = e^x - x^2$  using the function  $g(x) = -\sqrt{e^x}$ , to obtain the fixed point. We have taken as a starting point  $x_0 = 2.5$ , a value outside the interval in which the theorem is satisfied. As can be seen in 6.21(a). In spite of this, the algorithm converges quickly, and after 5 iterations, 6.21(f), has reached the fixed point —and therefore the root sought—, with the imposed tolerance.

If we try to use the function  $g(x) = \ln(x^2)$  to obtain the root, we observe that the function does not satisfy the theorem for any interval containing the root.

Figure 6.22 shows the function  $g(x)$ , the line  $y = x$  and the evolution of the algorithm after four evaluations. It is easy to deduce that the algorithm will jump from the positive branch to the negative branch and from the negative branch back to the positive branch.

The function has a vertical asymptote at 0. If one starts from  $x_0 = 0$ ,  $x_0 = 1$  or  $x_0 = -1$  the algorithm does not converge, since the function diverges towards  $-\infty$ . For the rest of the values, the function oscillates between one branch and another. If in any of the oscillations it manages to pass close enough to the fixed point,  $x_n - x_{n-1} \leq tol$ , the algorithm will have approximated the root, although it can-

que converja.

La figura 6.23(a), muestra la evolución del algoritmo, tomando como punto inicial  $x_0 = -0.2$ . Tras 211 iteraciones el algoritmo 'atrapa la raíz'. En este caso la tolerancia se fijó en  $tol = 0.01$ .

La gráfica 6.23(b) muestra una ampliación de 6.23(a) en la que pueden observarse en detalles los valores obtenidos para las dos últimas iteraciones. Las dos líneas horizontales de puntos marcan los límites  $\text{raíz} \pm tol$ .

El algoritmo se detiene porque la diferencia entre los valores obtenidos en las dos últimas iteraciones caen dentro de la tolerancia. El valor obtenido en la penúltima iteración, que proviene de la rama positiva de la función  $g(x)$  cae muy cerca del punto fijo. El último valor obtenido, se aleja de hecho del valor de la raíz, respecto al obtenido en la iteración anterior, pero no lo suficiente como para salirse de los límites de la banda marcada por la tolerancia. Como resultado, se cumple la condición de terminación y el algoritmo se detiene.

Si disminuimos el valor de la tolerancia, no podemos garantizar que el algoritmo converja. De hecho, si trazamos cuales habrían sido los valores siguientes que habría tomado la solución del algoritmo, caso de no haberse detenido, es fácil ver que se alejan cada vez más de la raíz. De nuevo habrá que esperar a que cambie de rama y vuelva a pasar otra vez cerca del punto fijo para que haya otra oportunidad de que el algoritmo *atrapa* la solución.

La gráfica 6.23(c) muestra la evolución del error en función del número de iteración. Como puede observarse, el error oscila de forma caótica de una iteración a la siguiente. De hecho, el estudio de las sucesiones de la forma  $x_{n+1} = g(x_n)$  constituyen uno de los puntos de partida para la descripción y el análisis de los llamados sistemas caóticos.

Uno sencillo, pero muy interesante es el de la ecuación logística discreta,  $x_{n+1} = R \cdot (1 - x_n) \cdot x_n$ . Esta ecuación muestra un comportamiento muy distinto, según cual sea el valor de  $R$  y el valor inicial  $x_0$  con el que empecemos a iterar.

Por último, si empleamos la función  $g(x) = \frac{e^x}{x}$ , no se cumple el teorema de punto fijo en

not properly be said to converge.

Figure 6.23(a), shows the evolution of the algorithm, taking  $x_0 = -0.2$  as the starting point. After 211 iterations the algorithm 'catches the root'. In this case the tolerance was set to  $tol = 0.01$ .

The graph 6.23(b) shows an enlargement of 6.23(a) in which the values obtained for the last two iterations can be seen in detail. The two horizontal dotted lines mark the  $\text{root} \pm tol$ .

The iteration stops because the difference between the values obtained in the last two iterations is lower than the tolerance. The value obtained in the penultimate iteration, which comes from the positive branch of the function  $g(x)$ , falls very close to the fixed point. The last value obtained, in fact, moves away from the value of the root, with respect to that obtained in the previous iteration, but not enough to fall outside the limits of the band marked by the tolerance. As a result, the termination condition is met and the algorithm stops.

If we decrease the value of the tolerance, we cannot guarantee that the algorithm will converge. In fact, if we plot what would have been the next values that the solution of the algorithm would have taken, had it not stopped, it is easy to see that they move further and further away from the root. Again, we will have to wait for it to change branches and pass close to the fixed point again for there to be another chance for the algorithm to catch the solution.

The graph 6.23(c) shows the evolution of the error as a function of the iteration number. As can be seen, the error oscillates chaotically from one iteration to the next. In fact, the study of sequences of the form  $x_{n+1} = g(x_n)$  is one of the starting points for the description and analysis of so-called chaotic systems.

A simple, but very interesting one is that of the discrete logistic equation,  $x_{n+1} = R \cdot (1 - x_n) \cdot x_n$ . This equation shows very different behaviour, depending on the value of  $R$  and the initial value  $x_0$  with which we start iterating.

Finally, if we use the function  $g(x) = \frac{e^x}{x}$ , the fixed point theorem is not satisfied at any

ningún punt. En este caso, el algoritmo diverge siempre. La figura 6.24 muestra la evolución del algoritmo del punto fijo para esta función. Se ha elegido un punto de inicio  $x_0 = -0.745$ , muy próximo al valor de la raíz, para poder observar la divergencia de las soluciones obtenidas con respecto al punto fijo. Como puede verse, el valor de  $x_n$  cada vez se aleja más de la raíz. LA solución oscila entre un valor que cada vez se aproxima más a cero y otro que tiende hacia  $-\infty$ . Si se deja aumentar suficientemente el número de iteraciones, llegará un momento en que se producirá un error de desbordamiento.

A diferencia de lo que sucedía en la elección de  $g(x) = \ln(x^2)$ , en este caso, el algoritmo no oscila entre las dos ramas. Si empezamos en la rama de la derecha, eligiendo un valor positivo para  $x_0$ , el algoritmo diverge llevando las soluciones hacia  $+\infty$ . Es un resultado esperable, ya que dicha rama no tiene ningún punto fijo.

### 6.3. Cálculo de raíces de funciones con Python.

Python tiene sus propios métodos para calcular raíces de funciones. Vamos a ver dos de esos métodos. Primero la función `fsolve` del paquete `scipy.optimize` que calcula raíces de todo tipo de funciones. Después veremos las funciones específicas para usar polinomios del paquete `numpy` y entre ellas la función `roots` para hallar las raíces de un polinomio.

#### 6.3.1. La función `fsolve`

La función `fsolve` se encuentra en el paquete `optimize` de `scipy`. Esta función devuelve las raíces de un sistema de ecuaciones definidas por  $func(x) = 0$  dado un punto de inicio.

Si el algoritmo no llega a converger `fsolve` devuelve el resultado de la última iteración.

La sintaxis del método `fsolve` en su forma más sencilla es la siguiente:

```
import scipy.optimize as opt
```

point. In this case, the algorithm always diverges. Figure 6.24 shows the evolution of the fixed-point iteration for this function. A starting point  $x_0 = -0.745$ , very close to the value of the root, has been chosen in order to observe the divergence of the solutions obtained with respect to the fixed point. As can be seen, the value of  $x_n$  moves further and further away from the root. The solution oscillates between a value that gets closer and closer to zero and another that tends towards  $-\infty$ . If the number of iterations is allowed to increase sufficiently, there will come a time when an overflow error will occur.

Unlike in the choice of  $g(x) = \ln(x^2)$ , in this case, the algorithm does not oscillate between the two branches. If we start on the right-hand branch, choosing a positive value for  $x_0$ , the algorithm diverges, taking the solutions towards  $+\infty$ . This is to be expected, since this branch has no fixed point.

### 6.3. Python root finding

Python has its own methods for calculating roots of functions. Let's take a look at two of those methods. First, the `fsolve` function from the `scipy.optimize` package calculates roots of all kinds of functions. Then we will look at the specific functions for using polynomials from the `numpy` package, including the `roots` function for finding the roots of a polynomial.

#### 6.3.1. Function `fsolve`

`fsolve` returns the roots of the (non-linear) equations defined by  $func(x) = 0$  given a starting estimate.

If algorithm fails `fsolve` returns the last iteration result.

The syntax of the `fsolve` method in its simplest form is as follows:

```
import scipy.optimize as opt
roots=opt.fsolve(fun,[x0])
```

```
roots=opt.fsolve(fun,[x0])
```

donde `fun` representa el nombre de la función que se desea evaluar, `[x0]`, son los valores de partida de las raíces y `roots` las raíces calculadas oir `fsolve`.

Un ejemplo de uso de `fsolve` para calcular la raíz de  $f(x) = e^x - x^2$  partiendo de una aproximación inicial de  $x_0 = 2$  sería el siguiente:

```
import numpy as np
import scipy.optimize as opt

def fun(x):
    y=np.exp(x)-x**2
    return y

roots=opt.fsolve(fun,-2)
```

El método `fsolve`, tiene muchas posibilidades de ajuste de la precisión, el número máximo de iteraciones, etc. Para obtener una visión mas completa de su uso, consultar la ayuda de `scipy.optimize.fsolve`.

### 6.3.2. Cálculo de raíces de polinomios.

Podemos crear y manipular polinomios en NumPy usando las clases del paquete `numpy.polynomial package` incorporado en Numpy 1.4. La clase `Polynomial` proporciona los métodos numéricos habituales en Python `+, -, *, /, %` y otros específicos de polinomios.

Para crear un polinomio tendremos que importar el módulo `numpy.polynomial` y usar la clase `Polynomial`. A esta clase, como primer argumento le proporcionaremos un vector cuyos elementos, son los coeficientes del polinomio ordenados de menor grado a mayor grado. Así por ejemplo, el polinomio  $y = 1 + 4x + 3x^2 + 2x^3$  se representa mediante el vector, `[1 4 3 2]` de la siguiente manera:

```
from numpy.polynomial import Polynomial as P
p = P([1,4,3,2])
```

where `fun` represents the name of the function to be evaluated, `[x0]` are the starting values of the roots and `roots` are the calculated roots or `fsolve`.

An example of using `fsolve` to calculate the root of  $f(x) = e^x - x^2$  starting from an initial approximation of  $x_0 = 2$  would be as follows:

```
import numpy as np
import scipy.optimize as opt

def fun(x):
    y=np.exp(x)-x**2
    return y

roots=opt.fsolve(fun,-2)
```

The `fsolve` method has many possibilities to adjust the precision, the maximum number of iterations, etc. For a more complete overview of its use, see the `scipy.optimize.fsolve` help.

### 6.3.2. Polynomials roots

Polynomials in NumPy can be created, manipulated, and even fitted using the convenience classes of the `numpy.polynomial package`, introduced in NumPy 1.4. The `Polynomial` class provides the standard Python numerical methods `+, -, *, //, %, **` and others.

To create a polynomial we have to import the `polynomial` package and use the `Polynomial` class. Coefficients are given in an array-like form in order of increasing degree, i.e., `p1 = [1 4 3 2]` give  $y = 1 + 4x + 3x^2 + 2x^3$ .

El polinomio  $y = 3x^4 + 2x^2 + 6x$  se representa mediante el vector, [0 6 2 0 3] y, en general, el polinomio  $y(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1} + a_nx^n$  se representa mediante el vector  $[a_0 \ a_1 \ a_2 \ \cdots \ a_{n-1} \ a_n]$ . Si al polinomio le falta algún o algunos términos, el elemento correspondiente toma el valor 0 en el vector que representa el polinomio.

```
from numpy.polynomial import Polynomial as P
p = P([0,6,2,0,3])
```

**El método roots de la clase Polynomial.** Este método de la clase `Polynomial` calcula las raíces de un polinomio de grado  $n$  definido como un como los que acabamos de describir. La sintaxis es: `p.roots()` donde `p` es un polinomio definido como hemos visto antes. Veamos un ejemplo. Dado el polinomio  $y(x) = x^3 - 6^2 + 11x - 6$  lo expresaríamos y calcularíamos sus raíces en python como,

```
from numpy.polynomial import Polynomial as P
p=P([1,-6,11,-6])
print(p)
print("The roots are:\t",p.roots())
1.0 - 6.0·x + 11.0·x² - 6.0·x³
The roots are: [0.33333333 0.5
```

El método `roots` devuelve las raíces del polinomio en un único vector. Si el polinomio tiene raíces complejas devuelve su parte real como en el caso del polinomio  $y(x) = x^2 + 2x + 1$

```
from numpy.polynomial import Polynomial as P
p=P([1,2,1])
print(p)
print("The roots are:\t",p.roots())
1.0 + 2.0·x + 1.0·x²
The roots are: [-1.00000001 -0.99999999]
```

**La función fromroots.** Esta función podría considerarse la opuesta a la anterior; dado un vector que contiene las raíces de un polinomio, nos devuelve el polinomio correspondien-

Polynomial  $y = 3x^4 + 2x^2 + 6x$  is represented by array, [0 6 2 0 3]. In general, polynomial  $y(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1} + a_nx^n$  is represented by array  $[a_0 \ a_1 \ a_2 \ \cdots \ a_{n-1} \ a_n]$ . if the polynomial does not have any of the monomials, the corresponding coefficient is array is 0.

**Method roots.** This method returns the roots of a polynomial defined using the class `Polynomial`. The syntax is: `p.roots()` where `p` is a polynomial defined as above. Let's see an example. Given the polynomial  $y(x) = x^3 - 6^2 + 11x - 6$  we define it and compute the roots in python as:

1. ]

The method `roots` returns the polynomial roots in a single vector. If the polynomial has complex roots it returns its real part as in the case of the polynomial  $y(x) = x^2 + 2x + 1$

**The method fromroots.** This method could be considered the opposite of the previous one; given a vector containing the roots of a polynomial, it returns the corresponding polyno-

te, Por ejemplo si definimos el vector de raíces,  $[3, 2, 1]$  podemos obtener el polinomio que posee esas raíces. En este caso el polinomio es  $y(x) = x^3 - 6x^2 + 11x - 6$

```
from numpy.polynomial import Polynomial as P
p=P.fromroots([3,2,1])
print(p)

-6.0 + 11.0·x - 6.0·x² + 1.0·x³
```

**El método linspace.** El método `linspace` devuelve valores equiespaciados  $x$  y sus correspondientes  $y$  en el dominio definido del polinomio. Si dicho dominio no se ha definido al crear el polinomio, se establece por defecto como  $[-1, 1]$ . Este método es muy útil para representar un polinomio gráficamente, como se ve en el siguiente ejemplo y en la figura 6.25.

```
from numpy.polynomial import Polynomial as P
#Polynomial p created and domain defined as [-2,2]
p=P([0,6,2,0,3],[-2,2])
dat=p.linspace()
plt.figure()
plt.grid()
plt.plot(dat[0],dat[1])
plt.xlabel('x')
plt.ylabel('y')
p_string=str(p)
plt.title(p_string)
```

**El método deriv.** El método `deriv` devuelve un polinomio que es el resultado de calcular la derivada del polinomio actual.

**El método integ.** Devuelve un polinomio que es la integral del polinomio actual.

**El método copy.** Devuelve una copia del polinomio actual.

```
from numpy.polynomial import Polynomial as P

#Create a polynomial with roots 3, -2, 2
p3=P.fromroots([3,-2,2])
print(p3)
#Compute the polynomial that is its derivative
```

mial. For example, if we define the vector of roots,  $[3, 2, 1]$  we can obtain the polynomial that has those roots. In this case the polynomial is  $y(x) = x^3 - 6x^2 + 11x - 6$ .

**Method linspace.** Method `linspace` returns  $x, y$  values at equally spaced points in domain of polynomial. If domain has been not defined  $[-1, 1]$  is set. This method is usefull to plot the polynomial, as in the next example and figure 6.25.

**Method deriv.** Returns a polynomial instance of that is the derivative of the current polynomial.

**Method integ.** Returns a polynomial that is the integral of the current polynomial.

**Method copy.** Returns a copy of the current polynomial.

```

pdot=p3.deriv()
print("Derivative:", pdot)
#Compute the polynomial that is the integral
pint=p3.integ()
print("Integral:", pint)
#Create polynomial p4 as a copy of pint
p4=pint.copy()
print(p4)

12.0 - 4.0·x - 3.0·x2 + 1.0·x3
Derivative: -4.0 - 6.0·x + 3.0·x2
Integral: 0.0 + 12.0·x - 2.0·x2 - 1.0·x3 + 0.25·x4
0.0 + 12.0·x - 2.0·x2 - 1.0·x3 + 0.25·x4

```

## 6.4. Cálculo de raices con la precisión máxima

Como ya se vio en la sección 5.3 todos los números que empleamos en un ordenador están sometidos a errores a causa del tamaño finito de los registros empleados para guardarlos. Cualquier número que queramos representar ha de aproximarse por un número máquina. Por lo tanto, en vez de buscar la aproximación a la raiz de una función con una tolerancia dada, podemos hallarla con la máxima precisión que nos permita la representación numérica empleada. Esta precisión máxima está relacionada con el `epsilon` del computador ya que la distancia entre dos números consecutivos es el épsilon multiplicado por dos elevado al exponente del número. De esta manera, por muchas iteraciones que hagamos la mejor aproximación que vamos a poder obtener de una raiz la obtendremos cuando el intervalo en el que se busca la raiz queda reducido a `eps` y por lo tanto no es posible reducirlo aún más.

En Numpy el método `spacing(a)` nos devuelve la distancia entre  $a$  y el siguiente número adyacente más próximo. De esta manera si a un número  $x$  le sumamos un valor menor que `spacing(x)` el resultado es el mismo número. Hay que tener en cuenta que para números negativos  $x < 0$  la distancia que nos devuelve `spacing(x)` es también negativa. No debería haber ningún número máquina entre  $x$  y  $x + spacing(x)$  para cualquier  $x$  finito.

## 6.4. Computing roots at maximum precision.

As we saw in section 5.3 all the numbers we use in a computer are subject to errors because of the finite size of the registers used to store them. Any number we want to represent has to be approximated by a machine number. Therefore, instead of looking for the approximation to the root of a function with a given tolerance, we can find it with the maximum precision allowed by the numerical representation used. This maximum precision is related to the computer's epsilon, since the distance between two consecutive numbers is the epsilon multiplied by two raised to the exponent of the number. Thus, no matter how many iterations we make, the best approximation we will be able to obtain of a root will be when the interval in which the root is sought is reduced to `eps` and therefore it is not possible to reduce it further.

In Numpy the method `spacing(a)` returns the distance between  $a$  and the next nearest adjacent number. So if we add a value less than `spacing(x)` to a number  $x$ , the result is the same number. Note that for negative numbers  $x < 0$  the distance returned by `spacing(x)` is also negative. There should not be any representable number between  $x + spacing(x)$  and  $x$  for any finite  $x$ .

```

import numpy as np

np.spacing(1)
Out[9]: 2.220446049250313e-16

np.spacing(1e-10)
Out[10]: 1.2924697071141057e-26

np.spacing(1e20)
Out[11]: 16384.0

10+np.spacing(10)/2
Out[12]: 10.0
10==(10+np.spacing(10)/2)
Out[13]: True

1e20==(1e20-np.spacing(1e20)/2)
Out[14]: True
np.spacing(-5)
Out[15]: -8.881784197001252e-16

-5+np.spacing(-5)
Out[16]: -5.000000000000001

-5-np.spacing(-5)
Out[17]: -4.999999999999999

```

Puesto que entre  $x$  y  $x + \text{spacing}(x)$  no hay ningún número máquina, podemos usar `spacing` para establecer la condición de parada de los algoritmos de búsqueda de raíces. Pararemos de buscar la raíz cuando el intervalo de búsqueda  $(a, b)$  sea tal que no haya números máquina entre  $a$  y  $b$ , es decir cuando  $|b - a| \leq |\text{np.spacing}(a)|$ .

Para el caso del método de la bisección se puede modificar el diagrama de flujo según se indica en la figura 6.26

Es más, en el método de la bisección se puede calcular en función del intervalo inicial el número de iteraciones necesario para alcanzar la máxima precisión, puesto que en cada iteración el intervalo se reduce a la mitad. Si llamamos  $d$  al intervalo de partida  $d = |a - b|$ , en la iteración  $n$  la longitud del intervalo será  $\frac{d}{2^{n-1}}$ . Se alcanzará el intervalo mínimo posible cuando su longitud sea la de `np.spacing(a)`. Pero según se explicaba en la sección ?? el valor de `eps` es igual a  $2$  elevado al número de bits de la mantisa. En el caso de un número

Since between  $x$  and  $x + \text{spacing}(x)$  there is no machine number, we can use `spacing` to establish the stop condition of the root search algorithms. We will stop searching for the root when the search interval  $(a, b)$  is such that there are no machine numbers between  $a$  and  $b$ , i.e., when  $|b - a| \leq |\text{np.spacing}(a)|$ .

For the case of the bisection method the flowchart can be modified as shown in the figure 6.26

Moreover, in the bisection method, the number of iterations required to achieve maximum accuracy can be calculated as a function of the starting interval, since at each iteration the interval is halved. If we call  $d$  the starting interval  $d = |a - b|$ , at iteration  $n$  the length of the interval will be  $\frac{d}{2^{n-1}}$ . The minimum possible interval will be reached when its length is that of `np.spacing(a)`. But as explained in the section 5.3 the value of `eps` is equal to  $2$  raised to the number of bits of the mantissa. In the case of a double precision number the

de doble precisión la mantisa tiene 52 bits y por lo tanto:

$$\frac{|d|}{2^{n-1}} < 2^{-52}$$

Y despejando y tomando logaritmos:

$$n > \log_2(|d|) + 53$$

Para métodos que no parten de un intervalo sino de un punto inicial, como el de Newton-Raphson o el del punto fijo también puede buscarse la raíz empleando `np.spacing(a)` de manera que se alcanzará la precisión máxima cuando el intervalo comprendido entre dos soluciones obtenidas en iteraciones consecutivas sea del tamaño de `np.spacing(a)`.

## 6.5. Ejercicios

- Crea una función en python que implemente el método de la bisección para obtener la raíz de una función cualquiera  $y = f(x)|x, y \in \mathbb{R}$ . La función deberá admitir como variables de entrada, la función  $f(x)$ , un intervalo de búsqueda  $[a, b]$ , un número máximo de iteraciones,  $nmax$ , a realizar y la tolerancia  $tol$  o error máximo admisible para la solución obtenida. Así mismo la función deberá devolver como variables de salida, el valor aproximado obtenido para la raíz  $r$ , el error cometido  $f(r)$  y el número de iteraciones empleado para alcanzar la solución.

Aplica el programa creado a la función de ejemplo empleada en el manual,  $f(x) = e^x - x^2$ , para comprobar que funciona correctamente.

- Crea una función en python que implemente el método de interpolación lineal para obtener la raíz de una función cualquiera  $y = f(x)|x, y \in \mathbb{R}$ . La función deberá admitir como variables de entrada, la función  $f(x)$ , un intervalo de búsqueda  $[a, b]$ , un número máximo de iteraciones,  $nmax$ , a realizar y la tolerancia  $tol$  o error máximo admisible para la solución obtenida. Así mismo la función

mantissa has 52 bits and therefore:

$$\frac{|d|}{2^{n-1}} < 2^{-52}$$

And clearing and taking logarithms:

$$n > \log_2(|d|) + 53$$

For methods that do not start from an interval but from an initial point, such as the Newton-Raphson or the fixed-point method, the root can also be found using `np.spacing(a)`, so that maximum precision will be achieved when the interval between two solutions obtained in consecutive iterations is the size of `np.spacing(a)`.

## 6.5. Problems

- Create a python function that implements the bisection method to obtain the root of any function  $y = f(x)|x, y \in \mathbb{R}$ . The function must admit as input variables, the function  $f(x)$ , a search interval  $[a, b]$ , a maximum number of iterations,  $nmax$ , to be carried out and the tolerance  $tol$  or maximum admissible error for the solution obtained. Likewise, the function must return as output variables, the approximate value obtained for the root  $r$ , the error committed  $f(r)$  and the number of iterations used to reach the solution.

Apply the program created to the example function used in the manual,  $f(x) = e^x - x^2$ , to check that it works correctly.

- Create a python function that implements the linear interpolation method to obtain the root of any function  $y = f(x)|x, y \in \mathbb{R}$ . The function must admit as input variables, the function  $f(x)$ , a search interval  $[a, b]$ , a maximum number of iterations,  $nmax$ , to be carried out and the tolerance  $tol$  or maximum admissible error for the solution obtained. Likewise, the function must return as output variables, the approximate value obtained for the root  $r$ , the error committed

deberá devolver como variables de salida, el valor aproximado obtenido para la raíz  $r$ , el error cometido  $f(r)$  y el número de iteraciones empleado para alcanzar la solución.

Aplica el programa creado a la función de ejemplo empleada en el manual,  $f(x) = e^x - x^2$ , para comprobar que funciona correctamente.

3. Crea una función en python que implemente el método de Newton-Raphson para obtener la raíz de una función cualquiera  $y = f(x)| x, y \in \mathbb{R}$ . La función deberá admitir como variables de entrada, la función  $f(x)$  y su función derivada  $f'(x)$ , un punto inicial de búsqueda  $x_0$ , un numero máximo de iteraciones,  $nmax$ , a realizar y la tolerancia  $tol$  o error máximo admisible para la solución obtenida. Así mismo la función deberá devolver como variables de salida, el valor aproximado obtenido para la raíz  $r$ , el error cometido  $f(r)$  y el número de iteraciones empleado para alcanzar la solución.

Aplica el programa creado a la función de ejemplo empleada en el manual,  $f(x) = e^x - x^2$ , para comprobar que funciona correctamente.

4. Crea una función en python que implemente el método de la secante para obtener la raíz de una función cualquiera  $y = f(x)| x, y \in \mathbb{R}$ . La función deberá admitir como variables de entrada, la función  $f(x)$  y su función derivada  $f'(x)$ , dos punto iniciales de búsqueda  $x_0, x_1$ , un numero máximo de iteraciones,  $nmax$ , a realizar y la tolerancia  $tol$  o error máximo admisible para la solución obtenida. Así mismo la función deberá devolver como variables de salida, el valor aproximado obtenido para la raíz  $r$ , el error cometido  $f(r)$  y el número de iteraciones empleado para alcanzar la solución.

Aplica el programa creado a la función de ejemplo empleada en el manual,  $f(x) =$

$f(r)$  and the number of iterations used to reach the solution.

Apply the program created to the example function used in the manual,  $f(x) = e^x - x^2$ , to check that it works correctly.

3. Create a python function that implements the Newton-Raphson method to obtain the root of any function  $y = f(x)|x, y \in \mathbb{R}$ . The function must admit as input variables, the function  $f(x)$  and its derivative function  $f'(x)$ , an initial search point  $x_0$ , a maximum number of iterations,  $nmax$ , to be carried out and the tolerance  $tol$  or maximum admissible error for the solution obtained. Likewise, the function must return as output variables, the approximate value obtained for the root  $r$ , the error committed  $f(r)$  and the number of iterations used to reach the solution.

Apply the program created to the example function used in the manual,  $f(x) = e^x - x^2$ , to check that it works correctly.

4. Create a python function that implements the secant method to obtain the root of any function  $y = f(x)|x, y \in \mathbb{R}$ . The function must admit as input variables, the function  $f(x)$  and its derivative function  $f'(x)$ , two initial search points  $x_0, x_1$ , a maximum number of iterations,  $nmax$ , to be carried out and the tolerance  $tol$  or maximum admissible error for the solution obtained. Likewise, the function must return as output variables, the approximate value obtained for the root  $r$ , the error committed  $f(r)$  and the number of iterations used to reach the solution.

Apply the program created to the example function used in the manual,  $f(x) = e^x - x^2$ , to check that it works correctly.

5. Create a python function that implements the fixed point iteration to obtain the root of any function  $y = f(x)|x, y \in \mathbb{R}$ . The function must admit as input variables, the auxiliary function  $g(x)$ , necessary to apply the method, an initial

$e^x - x^2$ , para comprobar que funciona correctamente.

5. Crea una función en python que implemente el método del punto fijo para obtener la raíz de una función cualquiera  $y = f(x) | x, y \in \mathbb{R}$ . La función deberá admitir como variables de entrada, la función auxiliar  $g(x)$ , necesaria para aplicar el método, un punto inicial de búsqueda  $x_0$ , un numero máximo de iteraciones,  $nmax$ , a realizar y la tolerancia  $tol$  o error máximo admisible para la solución obtenida. Así mismo la función deberá devolver como variables de salida, el valor aproximado obtenido para la raíz  $r$ , el error cometido  $f(r)$  y el número de iteraciones empleado para alcanzar la solución.

Aplica el programa creado a la función de ejemplo empleada en el manual,  $f(x) = e^x - x^2$ , empleando para ello los tres ejemplos de funciones auxiliare  $g(x)$  propuestos en la sección 6.2.5. Comprueba que se cumple en cada caso lo expuesto en dicha sección.

6. Se quiere aproximar el valor de  $\pi$  utilizando las dos siguientes funciones:

$$y = \cos(x) + 1$$

$$y = \cos(x/2)$$

- a) Dibuja las dos funciones y sus derivadas en el intervalo  $[\frac{\pi}{2}, \frac{3\pi}{2}]$ .
- b) Aplica el método de la secante para cada función, tomando como punto de partida  $x_0 = 3$  y obteniendo en ambos caso la solución con una tolerancia de 0.01. Elige para  $x_1$  un valor que consideres adecuado. El programa utilizado deberá añadir al gráfico obtenido en 6a), un punto en la posición  $(x_i, y_i)$  que represente el valor obtenido de la raíz en cada iteración  $i$ .
- c) Compara el número de iteraciones que se necesitan en cada caso para obtener la solución.

search point  $x_0$ , a maximum number of iterations,  $nmax$ , to be carried out and the tolerance  $tol$  or maximum admissible error for the solution obtained. Likewise, the function must return as output variables, the approximate value obtained for the root  $r$ , the error committed  $f(r)$  and the number of iterations used to reach the solution.

Apply the program created to the example function used in the manual,  $f(x) = e^x - x^2$ , using the three examples of auxiliary functions  $g(x)$  proposed in the section 6.2.5. Check that what is stated in that section is satisfied in each case.

6. The value of  $\pi$  is to be approximated using the following two functions:

$$y = \cos(x) + 1$$

$$y = \cos(x/2)$$

- a) Draw the two functions and their derivatives on the interval  $[\frac{\pi}{2}, \frac{3\pi}{2}]$ .
  - b) Apply the secant method for each function, taking  $x_0 = 3$  as a starting point and obtaining in both cases the solution with a tolerance of 0.01. Choose a value for  $x_1$  that you consider appropriate. The program used must add to the graph obtained in 6a), a point in the position  $(x_i, y_i)$  that represents the value obtained for the root in each iteration  $i$ .
  - c) Compare the number of iterations needed in each case to obtain the solution.
7. Steffensen's method allows the number of iterations used to obtain a root to be reduced by using the fixed point method. The algorithm calculates each iteration using two intermediate steps according to the following procedure,
- Where  $g(x)$  represents the auxiliary function for which the fixed point is sought. Given the function,

7. El método de Steffensen, permite disminuir el número de iteraciones empleadas para obtener una raíz empleando el método del punto fijo.

El algoritmo calcula cada iteración usando dos pasos intermedios de acuerdo con el siguiente procedimiento,

Donde  $g(x)$  representa la función auxiliar de la que se busca el punto fijo.

Dada la función,

$$y = x^2 - 2x - 3$$

se le puede aplicar el método del punto fijo a partir de dos reordenaciones distintas:

$$x^2 - 2x - 3 = 0 \Rightarrow x = \sqrt{2x + 3} \quad (6.1)$$

y

$$x^2 - 2x - 3 = 0 \Rightarrow x = \frac{3}{x - 2} \quad (6.2)$$

- a) Aplica el método del punto fijo, tomando como punto de partida  $x_0 = 4$ , mediante ambas reordenaciones. Calcula en ambos casos la raíz con una tolerancia de 0.0001. Indica en cada caso: el valor de la raíz y el número de interacciones empleadas. ¿Son razonables los resultados?
- b) Repite el cálculo empleando ahora el método de Steffensen y comprueba si emplea o no menos iteraciones que el punto fijo.
8. Un objeto se mueve de acuerdo con la siguiente ley:

$$x_1 = 3t^6 - 2t^5 + t + 25 \quad (6.3)$$

Donde  $t$  representa el tiempo en segundos y  $x_1$  la posición del objeto en metros y

Un segundo objeto sale en persecución del primero en el instante de tiempo  $t = 0$ . Su ley del movimiento es:

$$x_2 = 4t^6 \quad (6.4)$$

$$y = x^2 - 2x - 3$$

the fixed point method can be applied to it from two different reorderings:

$$x^2 - 2x - 3 = 0 \Rightarrow x = \sqrt{2x + 3} \quad (6.1)$$

and

$$x^2 - 2x - 3 = 0 \Rightarrow x = \frac{3}{x - 2} \quad (6.2)$$

- a) Apply the fixed point method, taking  $x_0 = 4$  as a starting point, using both rearrangements. Calculate in both cases the root with a tolerance of 0.0001. Indicate in each case: the value of the root and the number of interactions used. Are the results reasonable?
- b) Repeat the calculation using Steffensen's method and check whether or not it uses fewer iterations than the fixed point.

8. An object moves according to the following law:

$$x_1 = 3t^6 - 2t^5 + t + 25 \quad (6.3)$$

Where  $t$  represents the time in seconds and  $x_1$  the position of the object in metres

A second object leaves in pursuit of the first at time instant  $t = 0$ . Its law of motion is:

$$x_2 = 4t^6 \quad (6.4)$$

- a) Plot the position as a function of time for the two mobiles on the same graph, so that the range point can be observed.
- b) Calculate, using the bisection method or the method of *regula falsi*, the instant of time and the position at which the second mobile reaches the first one.

9. Maxwell's distribution,

$$f(v) = \sqrt{\frac{2}{\pi}} \left( \frac{m}{kT} \right)^{3/2} v^2 e^{-\frac{mv^2}{2kT}}, \quad (6.5)$$

- a) Dibuja en un mismo gráfica la posición en función del tiempo para los dos móviles, de modo que se observe el punto de alcance.
- b) Calcula, empleando el método de la bisección o el de *regula falsi*, el instante de tiempo y la posición en que el segundo móvil alcanza al primero.

9. La distribución de Maxwell,

$$f(v) = \sqrt{\frac{2}{\pi}} \left(\frac{m}{kT}\right)^{3/2} v^2 e^{-\frac{mv^2}{2kT}}, \quad (6.5)$$

da la probabilidad de que una partícula de un gas se mueva con velocidad  $v$ . Donde  $m$  es la masa de la partícula en Kg,  $T$  la temperatura en grados Kelving del gas y  $k = 1.380649 \times 10^{-23} \text{JK}^{-1}$  la constante de Boltzmann.

- a) Dibuja una gráfica de la distribución de Maxwell a temperatura ambiente  $T = 300\text{K}$  para el Nitrógeno  $\text{N}_2$  cuya masa molecular vale  $m = 4.65 \times 10^{-26}\text{Kg}$ .
- b) Calcula a dicha temperatura cuál es el valor más probable de la velocidad de una partícula.
- c) Calcula cuánto hay que bajar la temperatura para que la probabilidad de encontrar una partícula con la velocidad calculada en el apartado 9b valga 0.001. Dibuja sobre la misma gráfica del ejercicio 9a la distribución de Maxwell del Nitrógeno para esta nueva temperatura.

## 6.6. Test del curso 2020/2021

1. Una señal de FM, viene representada por la función

$$x(t) = 2 \cos \left( \pi t + \frac{3}{2} \sin \left( \frac{\pi}{2} t \right) \right). \quad (6.6)$$

- a) **1 punto.** Dibuja la función en el

gives the probability that a particle of a gas moves with velocity  $v$ . Where  $m$  is the mass of the particle in kg,  $T$  the temperature in degrees Kelving of the gas and  $k = 1.380649 \times 10^{-23} \text{JK}^{-1}$  the Boltzmann constant.

- a) Draw a graph of the Maxwell distribution at room temperature  $T = 300\text{K}$  for Nitrogen  $\text{N}_2$  whose molecular mass is  $m = 4.65 \times 10^{-26}\text{Kg}$ .
- b) Calculate at this temperature what is the most probable value of the velocity of a particle.
- c) Calculate how much the temperature has to be lowered so that the probability of finding a particle with the velocity calculated in section 9b is 0.001. Draw on the same graph as in exercise 9a the Maxwell distribution of Nitrogen for this new temperature.

## 6.6. 2020/21 Test

1. An FM signal, represented by the function

$$x(t) = 2 \cos \left( \pi t + \frac{3}{2} \sin \left( \frac{\pi}{2} t \right) \right). \quad (6.6)$$

- a) **1 pt.** Plot the function in the in-

- intervalo  $[0, \pi]$  Emplea para ello valores equiespaciados  $0.01\pi$ .
- b) **3 puntos.** Calcula los tres primeros instantes de tiempo en los que la señal vale cero, empleando para ello el método de la bisección. Elige en cada caso los intervalos adecuados. Obtén los resultados empleando una tolerancia  $tol = 0.001$  e indica el número de iteraciones empleadas.
2. Han Solo ha situado en reposo al Halcón Milenario en un punto del universo alejado de toda galaxia. Tras dejar a Chewbacca en una nave auxiliar, Han pone en marcha los motores (modelo *Girodyne SRB42 Sublight Engine Thrusters*) y el Halcón acelera con una aceleración uniforme  $a_0 = 3 \times 10^3 \text{ m/s}^2$ . Chewabaca –que no estudió físicas– desconoce la Teoría de la Relatividad, y por lo tanto piensa que puede calcular la posición  $x_{cl}$  con respecto del Halcón empleando la función clásica del movimiento rectilíneo uniformemente acelerado
- $$x_{cl}(t) = \frac{1}{2}a_0 t^2. \quad (6.7)$$
- En realidad el Halcón no puede acelerar indefinidamente (sobrepasaría la velocidad de la luz, cosa imposible con los *Sublight Engine Thrusters*), por lo que la posición real  $x_{rel}$  a la que se encuentra viene expresada por
- $$x_{rel}(t) = \frac{c^2}{a_0} \sqrt{1 + \frac{a_0^2 t^2}{c^2}} - \frac{c^2}{a_0}, \quad (6.8)$$
- donde  $c \approx 3 \times 10^8 \text{ m/s}$  representa la velocidad de la luz en el vacío.
- En los siguientes pasos vamos a calcular cuánto tiempo transcurre hasta que Chewbacca comete un error de un milliparsec<sup>3</sup> en el cálculo de la posición del Halcón.
- a) **2 puntos.** Define la función de error  $e(t) = x_{cl}(t) - x_{rel}(t)$  y simplifícalo con los datos numéricos dados.
- intervalo  $[0, \pi]$  Use equi-spaced values for  $0.01\pi$ .
- b) **3 pts.** Calculate the first three time instants in which the signal is zero, using the bisection method. In each case, choose the appropriate intervals. Obtain the results using a tolerance  $tol = 0.001$  and indicate the number of iterations used.
2. Han Solo has idled the Millennium Falcon at a point in the universe far from any galaxy. After leaving Chewbacca in an auxiliary ship, Han starts the engines (model *Girodyne SRB42 Sublight Engine Thrusters*) and the Falcon accelerates with a uniform acceleration  $a_0 = 3 \times 10^3 \text{ m/s}^2$ . Chewabaca –who did not study physics– does not know the Theory of Relativity, and therefore thinks he can calculate the position  $x_{cl}$  with respect to the Falcon using the classical function of uniformly accelerated rectilinear motion
- $$x_{cl}(t) = \frac{1}{2}a_0 t^2. \quad (6.7)$$
- In reality the Falcon cannot accelerate indefinitely (it would exceed the speed of light, which is impossible with the *Sublight Engine Thrusters*), so the real position  $x_{rel}$  at which it is located is expressed by
- $$x_{rel}(t) = \frac{c^2}{a_0} \sqrt{1 + \frac{a_0^2 t^2}{c^2}} - \frac{c^2}{a_0}, \quad (6.8)$$
- where  $c \approx 3 \times 10^8 \text{ m/s}$  is the speed of light in vacuum.
- In the following steps we are going to calculate how much time elapses until Chewbacca makes an error of one milliparsec<sup>3</sup> in the calculation of the Falcon's position.
- a) **2 pts.** Define the error function  $e(t) = x_{cl}(t) - x_{rel}(t)$  and simplify it with the given numerical data. Then rewrite the error function as  $e(t) =$

<sup>3</sup>1 Parsec  $\approx 3 \times 10^{16} \text{ m}$ .

<sup>3</sup>1 Parsec  $\approx 3 \times 10^{16} \text{ m}$ .

Acto seguido, reescribe la función error como  $e(t) = t - g(t)$  para poder aplicar más adelante el algoritmo del punto fijo. Con una sola descomposición de  $e(t)$  es suficiente. De hecho, quédate únicamente con aquella que tenga una  $g(t)$  más simple según tu criterio.

- b) **1.5 puntos.** Dibuja en una misma gráfica la función  $y = g(t)$  y la bisectriz del primer cuadrante  $y = t$ , y comprueba que  $g(t)$  tiene efectivamente un punto fijo. Para la gráfica escoge el intervalo de tiempo  $[0, 3 \times 10^5]$  segundos con intervalos de  $1 \times 10^4$  segundos. 3\*
- c) **2.5 puntos.** Emplea el método del punto fijo para obtener el tiempo en el que el error en posición alcanza un valor igual a un milliParsec. Da el resultado en días terrestres. Toma como valor inicial  $t_0 = 1$  y da el resultado con una tolerancia  $tol = 0.1$ . Indica el número de iteraciones que ha empleado el método para obtener el resultado.
- d) **Bonus track: 1 punto.** Un Parsec equivale a la distancia desde el Sol a una estrella que, vista desde la Tierra, subtende un ángulo de paralaje de 1 segundo de grado. Da dos razones por las que el Halcón Milenario nunca pudo atravesar el corredor Kessel en menos de 12 Parsecs, como afirma Han Solo.

$t - g(t)$  so that the fixed point algorithm can be applied later. A single decomposition of  $e(t)$  is sufficient. In fact, keep only the one with the simplest  $g(t)$  according to your criteria.

- b) **1.5 pts.** Plot on the same graph the function  $y = g(t)$  and the bisector of the first quadrant  $y = t$ , and check that  $g(t)$  does indeed have a fixed point. For the graph choose the time interval  $[0, 3 \times 10^5]$  seconds with intervals of  $1 \times 10^4$  seconds.
- c) **2.5 pts.** Use the fixed point iteration to obtain the time at which the position error reaches a value equal to one milliParsec. Give the result in Earth days. Take as initial value  $t_0 = 1$  and give the result with a tolerance  $tol = 0.1$ . Indicates the number of iterations that the method has used to obtain the result.
- d) **Bonus track: 1 pt.** A Parsec is the distance from the Sun to a star that, when viewed from Earth, subtends a parallax angle of 1 second of a degree. Give two reasons why the Millennium Falcon could never have crossed the Kessel corridor in less than 12 Parsecs, as Han Solo claims.

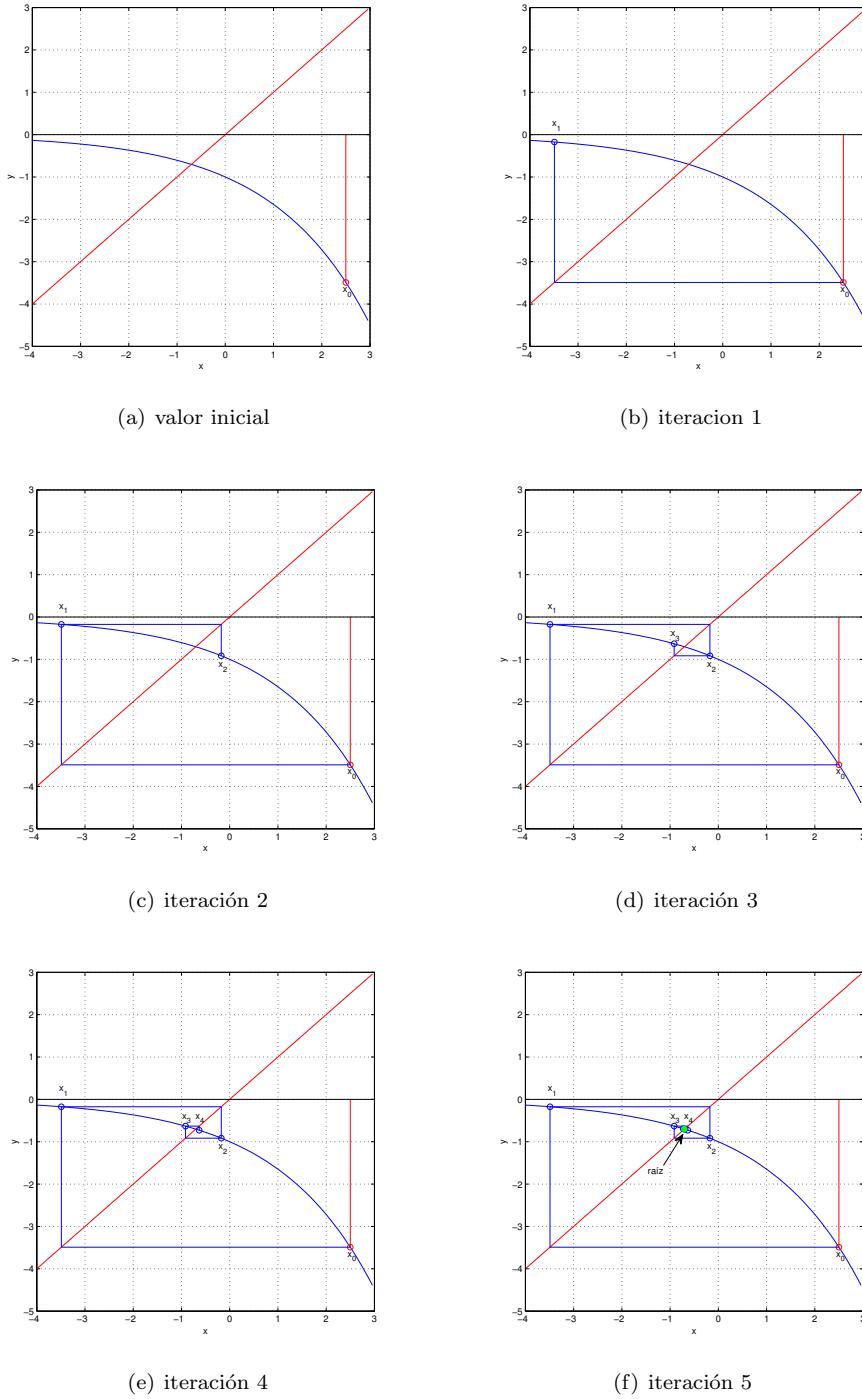


Figura 6.21: proceso de obtención de la raíz de la función  $f(x) = e^x - x^2$  aplicando el método del punto fijo sobre la función  $g(x) = -\sqrt{e^x}$

Figure 6.21: Fixed point iteration to compute the root of  $f(x) = e^x - x^2$  using the function  $g(x) = -\sqrt{e^x}$

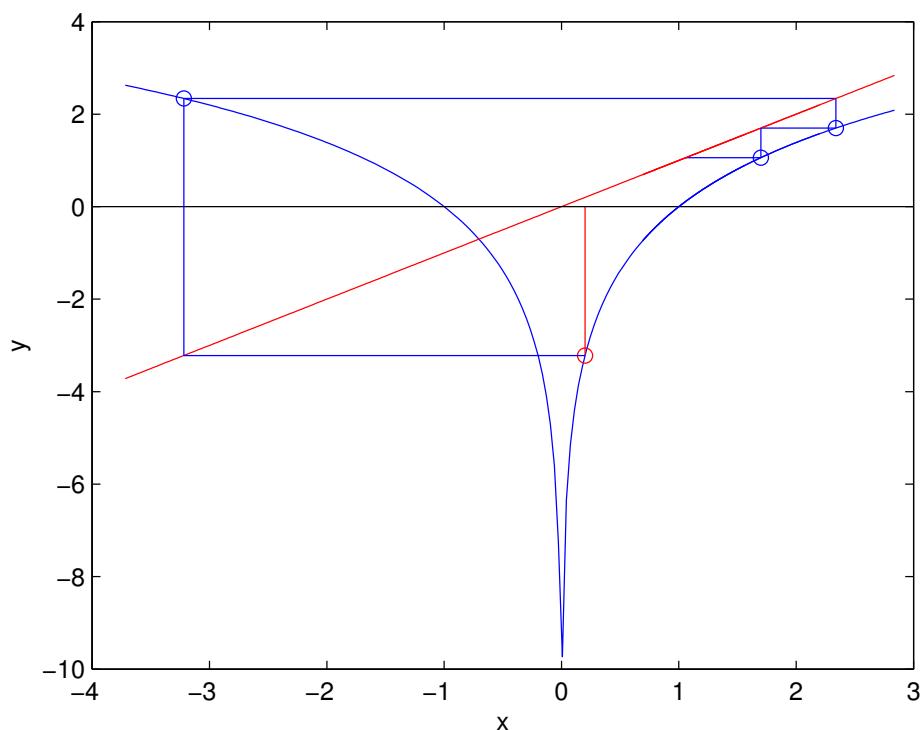
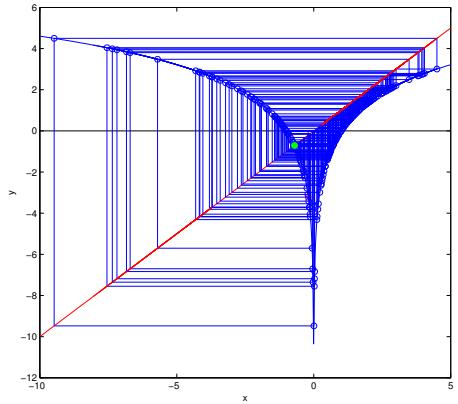
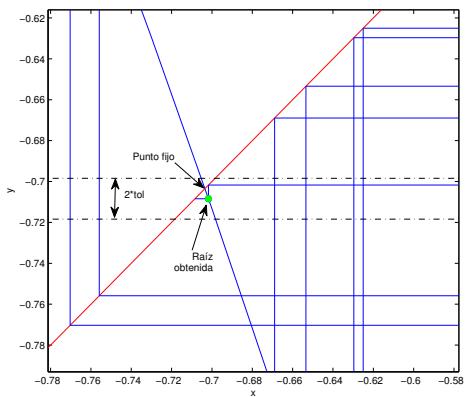


Figura 6.22: primeras iteraciones de la obtención de la raíz de la función  $f(x) = e^x - x^2$  aplicando el método del punto fijo sobre la función  $g(x) = \ln(x^2)$ .

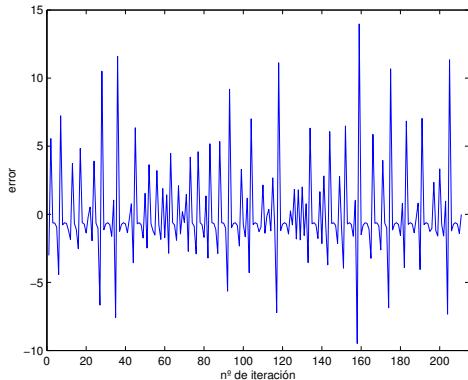
Figure 6.22: first iterations of obtaining the root of the function  $f(x) = e^x - x^2$  by applying the fixed point method on the function  $g(x) = \ln(x^2)$ .



(a) Evolución del algoritmo durante 211 iteraciones



(b) Vista detallada de las ultimas iteraciones de 6.23(a)



(c) Evolución del error

Figura 6.23: proceso de obtención de la raíz de la función  $f(x) = e^x - x^2$  aplicando el método del punto fijo sobre la función  $g(x) = \ln(x^2)$ , el método oscila sin converger a la solución.

Figure 6.23: Fixed point iteration to compute the root of function  $f(x) = e^x - x^2$  using the function  $g(x) = \ln(x^2)$ , iteration oscillates without converging to the solution.

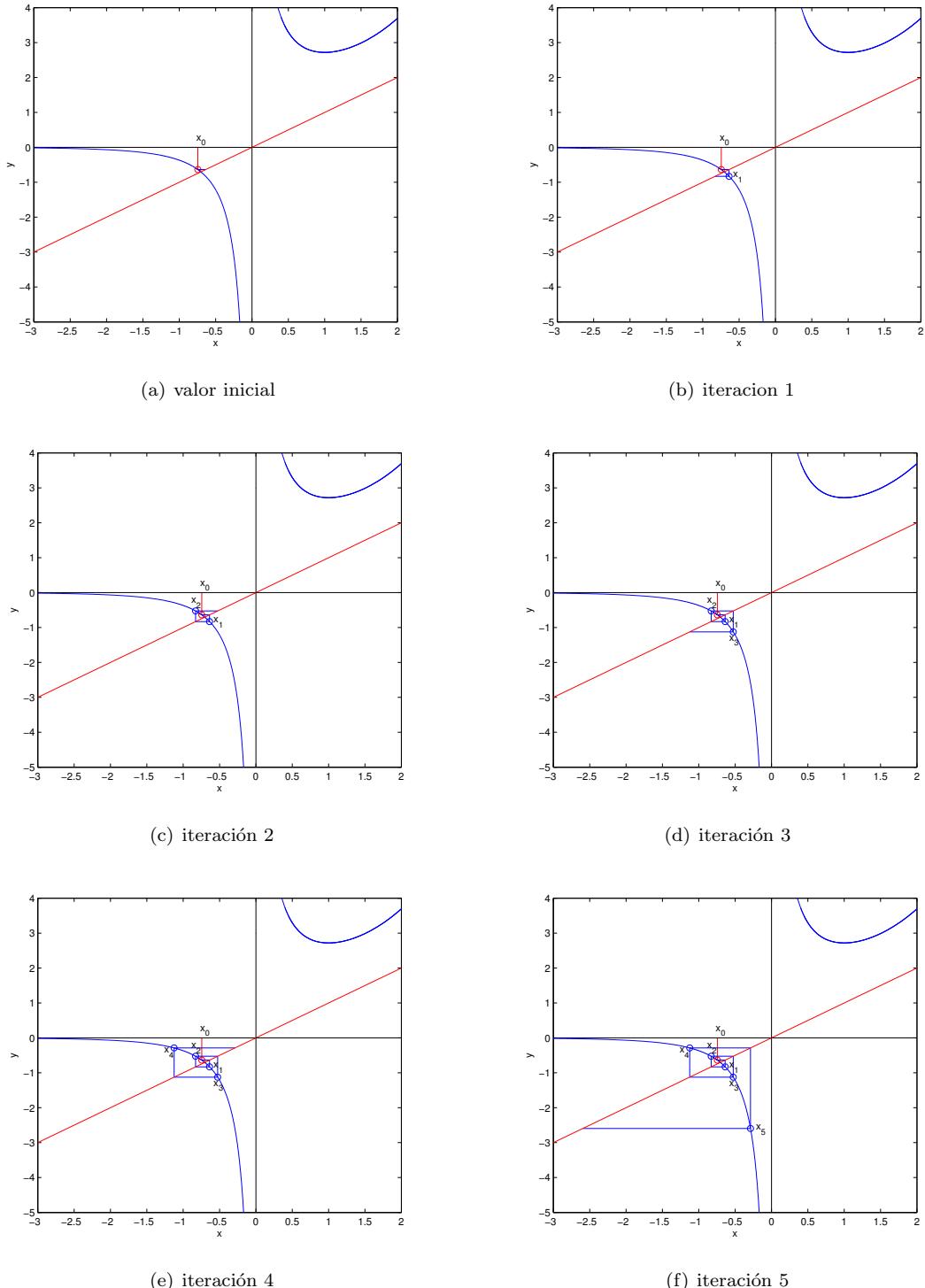


Figura 6.24: proceso de obtención de la raíz de la función  $f(x) = e^x - x^2$  aplicando el método del punto fijo sobre la función  $g(x) = \frac{e^x}{x}$ , el método diverge rápidamente.

Figure 6.24: fixed point iteration to compute the root of function  $f(x) = e^x - x^2$  using the function  $g(x) = \frac{e^x}{x}$ , the iteration diverges quickly.

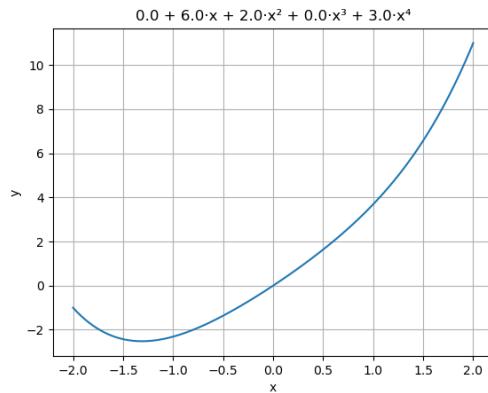


Figura 6.25: Polinomio dibujado usando `linspace`  
 Figure 6.25: Plotting a polynomial using `linspace`

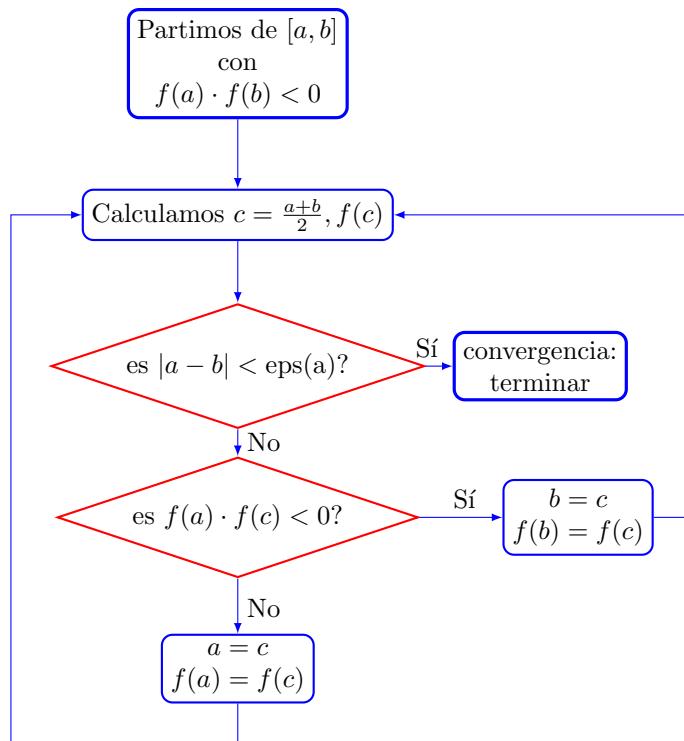


Figura 6.26: Diagrama de flujo del método de la bisección con precisión máxima

## Capítulo/Chapter 7

# Sistemas de ecuaciones lineales Linear equation systems

Después de cada guerra  
alguien tiene que limpiar.  
No se van a ordenar solas las  
cosas,  
digo yo. (Maria Wisława Anna  
Szymborska)

### 7.1. Introducción

Una ecuación lineal es aquella que establece una relación *lineal* entre dos o más variables, por ejemplo,

$$3x_1 - 2x_2 = 12$$

Se dice que es una relación lineal, porque las variables están relacionadas entre sí tan solo mediante sumas y productos por coeficientes constantes. En particular, el ejemplo anterior puede representarse geométricamente mediante una línea recta.

El número de variables relacionadas en una ecuación lineal determina la dimensión de la ecuación. La del ejemplo anterior es una ecuación bidimensional, puesto que hay dos variables. El número puede ser arbitrariamente grande en general,

$$a_1x_1 + a_2x_2 + \dots + a_nx_n = b$$

será una ecuación n-dimensional.

### 7.1. Introduction

A linear equation is an equation that establishes a *linear* relationship between two or more variables, for example,

$$3x_1 - 2x_2 = 12 \quad (7.1)$$

It is said to be a linear relationship, because the variables are related to each other only through additions and products by constant coefficients. In particular, the above example can be represented geometrically by a straight line.

The number of related variables in a linear equation determines the dimension of the equation. The above example is a two-dimensional equation, since there are two variables. The number can be arbitrarily large in general,

$$a_1x_1 + a_2x_2 + cdots + a_nx_n = b$$

will be an n-dimensional equation.

Como ya hemos señalado más arriba, una ecuación bidimensional admite una línea recta como representación geométrica, una ecuación tridimensional admitirá un plano y para dimensiones mayores que tres cada ecuación representará un hiperplano de dimensión n. Por supuesto, para dimensiones mayores que tres, no es posible obtener una representación gráfica de la ecuación.

Las ecuaciones lineales juegan un papel muy importante en la física y, en general en la ciencia y la tecnología. La razón es que constituyen la aproximación matemática más sencilla a la relación entre magnitudes físicas. Por ejemplo cuando decimos que la fuerza aplicada a un resorte y la elongación que sufre están relacionadas por la ley de Hooke,  $F = Kx$  estamos estableciendo una relación lineal entre las magnitudes fuerza y elongación. ¿Se cumple siempre dicha relación? Desde luego que no. Pero es razonablemente cierta para elongaciones pequeñas y, apoyados en ese sencillo modelo *lineal* de la realidad, se puede aprender mucha física.

Un sistema de ecuaciones lineales está constituido por varias ecuaciones lineales, que expresan relaciones lineales distintas sobre las mismas variables. Por ejemplo,

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 &= b_1 \\ a_{21}x_1 + a_{22}x_2 &= b_2 \end{aligned}$$

Se llaman soluciones del sistema de ecuaciones a los valores de las variables que satisfacen simultáneamente a todas las ecuaciones que componen el sistema. Desde el punto de vista de la obtención de las soluciones a las variables se les suele denominar incógnitas, es decir valores no conocidos que deseamos obtener o calcular.

Un sistema de ecuaciones puede tener infinitas soluciones, puede tener una única solución o puede no tener solución. En lo que sigue, nos centraremos en sistemas de ecuaciones que tienen una única solución.

Una primera condición para que un sistema de ecuaciones tengan una única solución es que el número de incógnitas presentes en el sistema coincida con el número de ecuaciones.

As we have already pointed out above, a two-dimensional equation admits a straight line as geometrical representation, a three-dimensional equation will admit a plane and for dimensions greater than three each equation will represent a hyperplane of dimension n. Of course, for dimensions greater than three, it is not possible to obtain a graphical representation of the equation.

Linear equations play a very important role in physics and, in general, in science and technology. The reason is that they are the simplest mathematical approximation of the relationship between physical quantities. For example, when we say that the force applied to a spring and the elongation it undergoes are related by Hooke's law,  $F = Kx$ , we are establishing a linear relationship between the quantities force and elongation. Does this relationship always hold true? Certainly not. But it is reasonably true for small elongations and, based on this simple *linear* model of reality, a lot of physics can be learned.

A system of linear equations is made up of several linear equations, which express different linear relationships about the same variables. For example,

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 &= b_1 \\ a_{21}x_1 + a_{22}x_2 &= b_2 \end{aligned}$$

The values of the variables that simultaneously satisfy all the equations that make up the system are called solutions of the system of equations. From the point of view of obtaining the solutions, the variables are usually called unknowns, i.e. unknown values that we wish to obtain or calculate.

A system of equations can have infinite solutions, it can have only one solution or it can have no solution. In what follows, we will focus on systems of equations that have only one solution.

A first condition for a system of equations to have a unique solution is that the number of unknowns present in the system coincides with the number of equations.

In general terms, we can say that we are going to study numerical methods to solve

De modo general podemos decir que vamos a estudiar métodos numéricos para resolver con un computador sistemas de  $n$  ecuaciones con  $n$  incógnitas,

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ \dots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

Una de las grandes ventajas de los sistemas de ecuaciones lineales es que puede expresarse en forma de producto matricial,

$$\left. \begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ \dots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned} \right\} \Rightarrow \underbrace{\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}}_A \cdot \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}}_x = \underbrace{\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}}_b$$

La matriz  $A$  recibe el nombre de matriz de coeficientes del sistema de ecuaciones, el vector  $x$  es el vector de incógnitas y el vector  $b$  es el vector de términos independientes. Para resolver un sistema de ecuaciones podríamos aplicar álgebra de matrices, como se ha visto en el capítulo 3:

$$A \cdot x = b \Rightarrow x = A^{-1} \cdot b$$

Es decir, bastaría invertir la matriz de coeficientes y multiplicarla por la izquierda por el vector de coeficientes para obtener el vector de términos independientes. De aquí podemos deducir una segunda condición para que un sistema de ecuaciones tenga una solución única; Su matriz de coeficientes tiene que tener inversa. Veamos algunos ejemplos sencillos.

Tomaremos en primer lugar un sistema de dos ecuaciones con dos incógnitas,

$$\begin{aligned} 4x_1 + x_2 &= 6 \\ 3x_1 - 2x_2 &= -1 \end{aligned}$$

Si expresamos el sistema en forma de producto de matrices obtenemos,

$$\begin{pmatrix} 4 & 1 \\ 3 & -2 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 6 \\ -1 \end{pmatrix}$$

with a computer systems of  $n$  equations with  $n$  unknowns,

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ \dots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

One of the great advantages of systems of linear equations is that they can be expressed in matrix product form,

$$\underbrace{\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}}_A \cdot \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}}_x = \underbrace{\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}}_b$$

The matrix  $A$  is called the coefficient matrix of the system of equations, the vector  $x$  is the vector of unknowns and the vector  $b$  is the vector of independent terms. To solve a system of equations we could apply matrix algebra, as discussed in chapter 3:

That is, it would be enough to invert the matrix of coefficients and multiply it on the left by the vector of coefficients to obtain the vector of independent terms. From this we can deduce a second condition for a system of equations to have a unique solution; its coefficient matrix must have an inverse. Let us look at some simple examples.

We will first take a system of two equations with two unknowns,

$$\begin{aligned} 4x_1 + x_2 &= 6 \\ 3x_1 - 2x_2 &= -1 \end{aligned}$$

If we express the system as a product of matrices we obtain,

e invirtiendo la matriz de coeficientes y multiplicándola por el vector de términos independientes se llega al vector de soluciones del sistema,

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 4 & 1 \\ 3 & -2 \end{pmatrix}^{-1} \cdot \begin{pmatrix} 6 \\ -1 \end{pmatrix} = \begin{pmatrix} 2/11 & 1/11 \\ 3/11 & -4/11 \end{pmatrix} \begin{pmatrix} 6 \\ -1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

En el ejemplo que acabamos de ver, cada ecuación corresponde a una recta en el plano, en la figura 7.1 se han representado dichas rectas gráficamente. El punto en que se cortan es precisamente la solución del sistema.

and by inverting the coefficient matrix and multiplying it by the vector of independent terms we arrive at the vector of solutions of the system,

In the example we have just seen, each equation corresponds to a line in the plane, in the figure 7.1 these lines have been represented graphically. The point where they intersect is precisely the solution of the system.

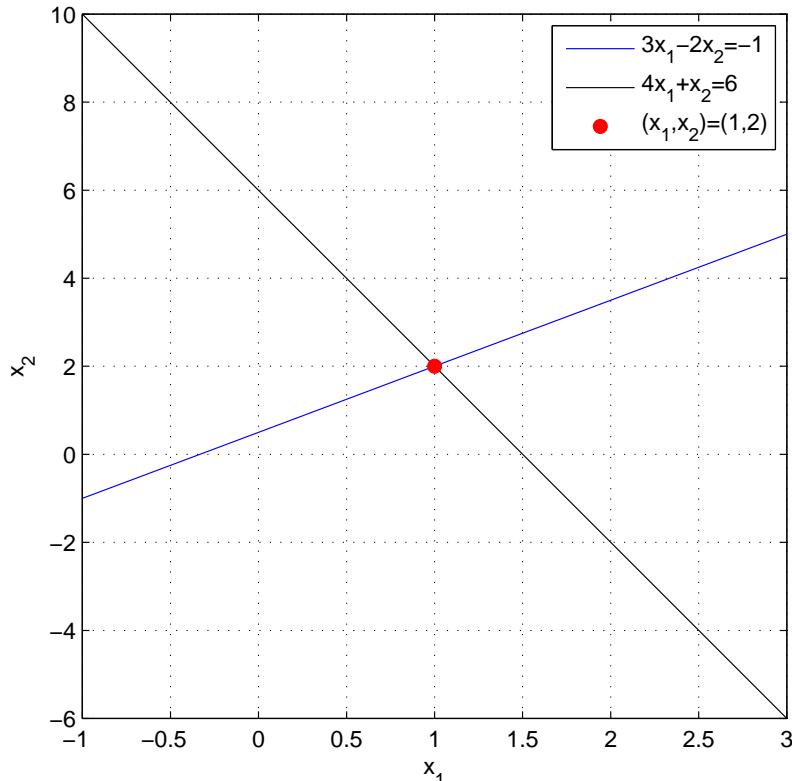


Figura 7.1: Sistema de ecuaciones con solución única  
Figure 7.1: Equation system with unique solution

Supongamos ahora el siguiente sistema, también de dos ecuaciones con dos incógnitas,

Now suppose the following system, also of

$$\begin{aligned} 4x_1 + x_2 &= 6 \\ 2x_1 + \frac{1}{2}x_2 &= -1 \end{aligned}$$

El sistema no tiene solución. Su matriz de coeficientes tiene determinante cero, por lo que no es invertible,

$$|A| = \begin{vmatrix} 4 & 1 \\ 2 & 1/2 \end{vmatrix} = 0 \Rightarrow \nexists A^{-1}$$

Si representamos gráficamente las dos ecuaciones de este sistema (figura 7.2) es fácil entender lo que pasa, las rectas son paralelas, no existe ningún punto  $(x_1, x_2)$  que pertenezca a las dos rectas, y por tanto el sistema carece de solución.

Dos rectas paralelas lo son, porque tienen la misma pendiente. Esto se refleja en la matriz de coeficientes, en que las filas son proporcionales; si multiplicamos la segunda fila por dos, obtenemos la primera.

Por último, el sistema,

$$\begin{aligned} 4x_1 + x_2 &= 6 \\ 2x_1 + \frac{1}{2}x_2 &= 3 \end{aligned}$$

posee infinitas soluciones. La razón es que la segunda ecuación es igual que la primera multiplicada por dos: es decir, representa exactamente la misma relación lineal entre las variables  $x_1$  y  $x_2$ , por tanto, todos los puntos de la recta son solución del sistema. De nuevo, la matriz de coeficientes del sistema no tiene inversa ya que su determinante es cero.

Para sistemas de ecuaciones de dimensión mayor, se cumple también que que el sistema no tiene solución única si el determinante de su matriz de coeficiente es cero. En todos los demás casos, es posible obtener la solución del sistema invirtiendo la matriz de coeficientes y multiplicando el resultado por el vector de términos independientes.

En cuanto un sistema de ecuaciones tiene una dimensión suficientemente grande, invertir su matriz de coeficientes se torna un problema costoso o sencillamente inabordable.

two equations with two unknowns,

$$\begin{aligned} 4x_1 + x_2 &= 6 \\ 2x_1 + \frac{1}{2}x_2 &= -1 \end{aligned}$$

The system has no solution. Its coefficient matrix has zero determinant, so it is not invertible,

$$|A| = \begin{vmatrix} 4 & 1 \\ 2 & 1/2 \end{vmatrix} = 0 \Rightarrow \nexists A^{-1}$$

If we represent graphically the two equations of this system (figure 7.2) it is easy to understand what happens, the lines are parallel, there is no point  $(x_1, x_2)$  that belongs to the two lines, and therefore the system has no solution.

Two parallel lines are parallel because they have the same slope. This is reflected in the coefficient matrix, where the rows are proportional; if we multiply the second row by two, we get the first row.

Finally, the system,

$$\begin{aligned} 4x_1 + x_2 &= 6 \\ 2x_1 + \frac{1}{2}x_2 &= 3 \end{aligned}$$

has infinite solutions. The reason is that the second equation is the same as the first equation multiplied by two: that is, it represents exactly the same linear relationship between the variables  $x_1$  and  $x_2$ , therefore, all the points on the line are solutions of the system. Again, the coefficient matrix of the system has no inverse since its determinant is zero.

For higher dimensional systems of equations, it is also true that the system has no unique solution if the determinant of its coefficient matrix is zero. In all other cases, it is possible to obtain the solution of the system by inverting the coefficient matrix and multiplying the result by the vector of independent terms.

As soon as a system of equations has a sufficiently large dimension, inverting its coefficient matrix becomes a costly or simply intractable problem.

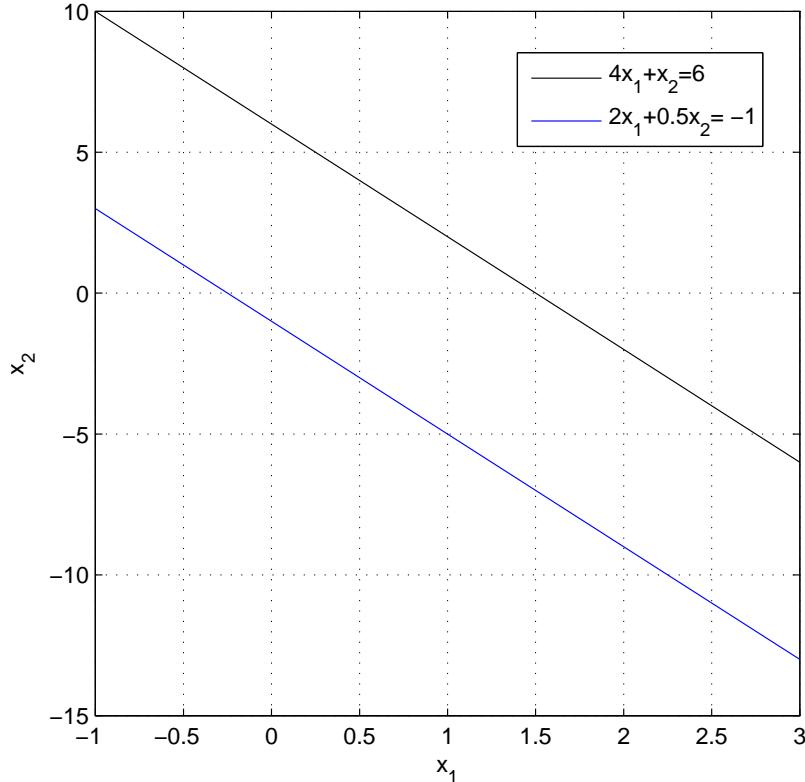


Figura 7.2: Sistema de ecuaciones sin solución

Figure 7.2: Equation system without solution

Desde un punto de vista numérico, la inversión de una matriz, presenta frecuentemente problemas debido al error de redondeo en las operaciones. Por esto, casi nunca se resuelven los sistemas de ecuaciones invirtiendo su matriz de coeficientes. A lo largo de este capítulo estudiaremos dos tipos de métodos de resolución de sistemas de ecuaciones. El primero de ellos recibe el nombre genérico de métodos directos, el segundo tipo lo constituyen los llamados métodos iterativos.

From a numerical point of view, the inversion of a matrix often presents problems due to the rounding error in the operations. For this reason, systems of equations are almost never solved by inverting their matrix of coefficients. Throughout this chapter we will study two types of methods for solving systems of equations. The first of these is known generically as direct methods, while the second type is known as iterative methods.

## 7.2. Condition number

In the introduction we have seen that for a system of equations to have a solution, its

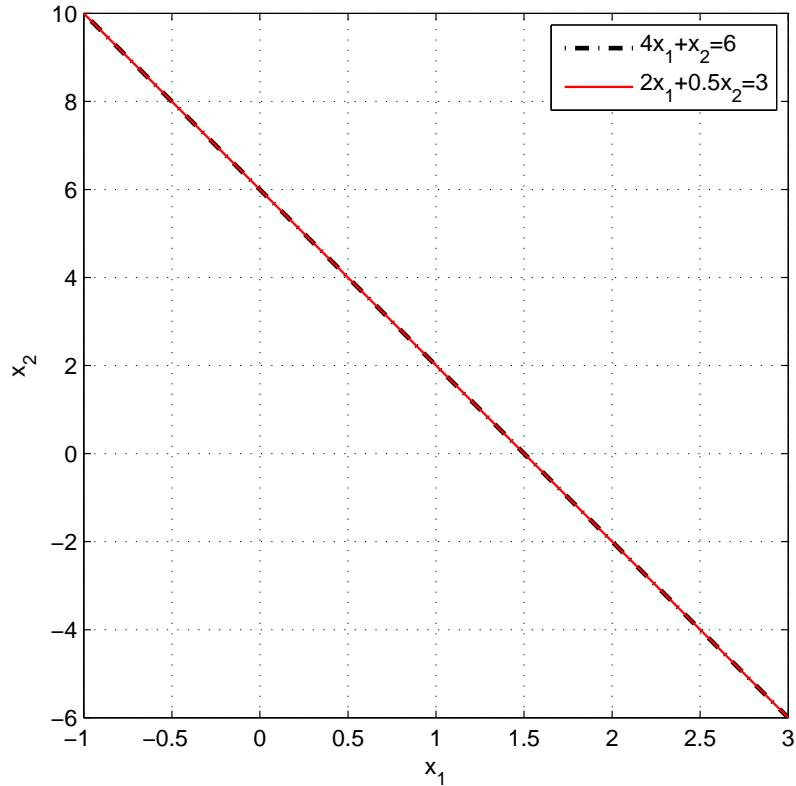


Figura 7.3: Sistema de ecuaciones con infinitas soluciones

Figure 7.3: Equation system with infinite solutions

## 7.2. Condicionamiento

En la introducción hemos visto que para que un sistema de ecuaciones tenga solución, es preciso que su matriz de coeficientes sea invertible. Sin embargo cuando tratamos de resolver un sistema de ecuaciones numéricamente, empleando un ordenador, debemos antes examinar cuidadosamente la matriz de coeficientes del sistema. Veamos un ejemplo: el sistema,

$$\begin{aligned} 4x_1 + x_2 &= 6 \\ 2x_1 + 0.4x_2 &= -1 \end{aligned}$$

coefficient matrix must be invertible. However, when we try to solve a system of equations numerically, using a computer, we must first carefully examine the matrix of coefficients of the system. Let us look at an example: the system,

$$\begin{aligned} 4x_1 + x_2 &= 6 \\ 2x_1 + 0.4x_2 &= -1 \end{aligned}$$

Has the following solutions,

$$x = \begin{pmatrix} -8.5 \\ 40 \end{pmatrix}$$

Tiene como soluciones,

$$x = \begin{pmatrix} -8.5 \\ 40 \end{pmatrix}$$

Si alteramos ligeramente uno de sus coeficientes,

$$\begin{aligned} 4x_1 + x_2 &= 6 \\ 2x_1 + 0.49x_2 &= -1 \end{aligned}$$

Las soluciones se alteran bastante; se vuelven aproximadamente 10 veces más grande,

$$x = \begin{pmatrix} -98.5 \\ 400 \end{pmatrix}$$

y si volvemos a alterar el mismo coeficiente,

$$\begin{aligned} 4x_1 + x_2 &= 6 \\ 2x_1 + 0.499x_2 &= -1 \end{aligned}$$

La solución es aproximadamente 100 veces más grande,

$$x = \begin{pmatrix} -998.5 \\ 4000 \end{pmatrix}$$

La razón para estos cambios es fácil de comprender intuitivamente; a medida que aproximamos el coeficiente a 0.5, estamos haciendo que las dos ecuaciones lineales sean cada vez más paralelas, pequeñas variaciones en la pendiente, modifican mucho la posición del punto de corte.

Cuando pequeñas variaciones en la matriz de coeficientes generan grandes variaciones en las soluciones del sistema, se dice que el sistema está mal condicionado, en otras palabras: que no es un sistema bueno para ser resuelto numéricamente. Las soluciones obtenidas para un sistema mal condicionado, hay que tomarlas siempre con bastante escepticismo.

Para estimar el buen o mal condicionamiento de un sistema, se emplea el número de condición, que definimos en el capítulo 3 en la sección 3.5.5, al hablar de la factorización SVD. El número de condición de una matriz

If we slightly change one of the coefficients,

$$\begin{aligned} 4x_1 + x_2 &= 6 \\ 2x_1 + 0.49x_2 &= -1 \end{aligned}$$

The solutions become quite altered; they become about 10 times larger,

$$x = \begin{pmatrix} -98.5 \\ 400 \end{pmatrix}$$

if we change again the same coefficient,

$$\begin{aligned} 4x_1 + x_2 &= 6 \\ 2x_1 + 0.499x_2 &= -1 \end{aligned}$$

The solution is about 100 times larger,

$$x = \begin{pmatrix} -998.5 \\ 4000 \end{pmatrix}$$

The reason for these changes is easy to understand intuitively; as we bring the coefficient closer to 0.5, we are making the two linear equations more and more parallel, and small variations in the slope greatly modify the position of the cut-off point.

When small variations in the matrix of coefficients generate large variations in the solutions of the system, it is said that the system is ill-conditioned, in other words: that it is not a good system to be solved numerically. The solutions obtained for an ill-conditioned system must always be taken with considerable scepticism.

To estimate the good or bad conditioning of a system, we use the condition number, which we defined in the chapter 3 in section 3.5.5, when talking about SVD factorisation. The condition number of a matrix is the quotient of its largest and smallest singular values. The closer the condition number is to 1, the better conditioned the matrix is, and the larger the condition number, the worse conditioned it is.

In Python, we can use the function `cond` inside the `linalg` module in `numpy` to compute the condition number of a matrix. We can apply to the last example coefficient matrix

es el cociente entre sus valores singulares mayor y menor. Cuanto más próximo a 1 sea el número de condición, mejor condicionada estará la matriz y cuanto mayor sea el número de condición peor condicionada estará.

En Python dentro del paquete `linalg` de `numpy` está la función `cond` que nos permite obtener el número de condición de una matriz. Si lo aplicamos a la matriz de coeficientes del último ejemplo mostrado,

```
import numpy as np

A=np.array([[4,1],[2,0.499]])

np.linalg.cond(A)
Out[7]: 5312.250061755533
```

El número está bastante alejado de uno, lo que, en principio, indica un mal condicionamiento del sistema.

Incidentalmente, podemos calcular la factorización svd de la matriz de coeficientes y dividir el valor singular mayor entre el menor para comprobar que el resultado es el mismo que nos da la función `cond`,

```
[U,S,Vt]=np.linalg.svd(A)

S[0]/S[1]
Out[14]: 5312.250061755533
```

## 7.3. Métodos directos

### 7.3.1. Sistemas triangulares

Vamos a empezar el estudio de los métodos directos por los algoritmos de resolución de los sistemas más simples posibles, aquellos cuya matriz de coeficientes es una matriz diagonal, triangular superior o triangular inferior.

**Sistemas diagonales.** Un sistema diagonal es aquel cuya matriz de coeficientes es una matriz diagonal.

The number is quite far from one, which, in principle, indicates a ill-condition of the system.

Incidentally, we can calculate the factorisation svd of the coefficient matrix and divide the largest singular value by the smallest to check that the result is the same as that given by the function `cond`,

### 7.2.1. Direct methods

### 7.2.2. Triangular systems

We are going to begin the study of direct methods with the algorithms for solving the simplest possible systems, those whose coefficient matrix is a diagonal, upper triangular or lower triangular matrix.

**Diagonal systems.** A diagonal system is one whose coefficient matrix is a diagonal matrix.

$$\left. \begin{array}{l} a_{11}x_1 = b_1 \\ a_{22}x_2 = b_2 \\ \dots \\ a_{nn}x_n = b_n \end{array} \right\} \Rightarrow \underbrace{\begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix}}_A \cdot \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}}_x = \underbrace{\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}}_b$$

Su resolución es trivial, basta dividir cada término independiente por el elemento correspondiente de la diagonal de la matriz de coeficientes,

$$x_i = \frac{b_i}{a_{ii}}$$

Para obtener la solución basta crear en Python un sencillo bucle `for`,

```
import numpy as np

def diag_sys(A,b):
    [f,c]=np.shape(A)
    x=np.zeros([f,1])
    for i in range(f):
        x[i]=b[i]/A[i,i]
    return x
```

**Sistemas triangulares inferiores: método de sustituciones progresivas.** Un sistema triangular inferior de  $n$  ecuaciones con  $n$  incógnitas tendrá la forma general,

$$\left. \begin{array}{l} a_{11}x_1 = b_1 \\ a_{21}x_1 + a_{22}x_2 = b_2 \\ \dots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n \end{array} \right\} \Rightarrow \underbrace{\begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}}_A \cdot \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}}_x = \underbrace{\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}}_b$$

El procedimiento para resolverlo a *mano* es muy sencillo, despejamos la primera la primera incógnita de la primera ecuación,

$$x_1 = \frac{b_1}{a_{11}}$$

A continuación sustituimos este resultado en la segunda ecuación, y despejamos  $x_2$ ,

$$x_2 = \frac{b_2 - a_{21}x_1}{a_{22}}$$

De cada ecuación vamos obteniendo una componente del vector solución, sustituyendo las soluciones obtenidas en las ecuaciones anteriores, así cuando llegamos a la ecuación  $i$ ,

$$x_i = \frac{b_i - \sum_{j=1}^{i-1} a_{ij}x_j}{a_{ii}}$$

Its resolution is trivial, just divide each independent term by the corresponding element of the diagonal of the coefficient matrix,

$$x_i = \frac{b_i}{a_{ii}}$$

To compute the solution it is enough to code a simple `for` loop,

**Lower triangular systems: method of progressive substitutions.** A lower triangular system of  $n$  equations with  $n$  unknowns will have the general form,

The procedure to solve to *hand* is very simple, we clear the first unknown from the first equation,

$$x_1 = \frac{b_1}{a_{11}}$$

Next, we substitute this result in the second equation and clear  $x_2$

$$x_2 = \frac{b_2 - a_{21}x_1}{a_{22}}$$

From each equation we obtain a component of the solution vector, substituting the solutions obtained in the previous equations, so when we arrive at the equation  $i$ ,

$$x_i = \frac{b_i - \sum_{j=1}^{i-1} a_{ij}x_j}{a_{ii}}$$

If we repeat this same process until we

Si repetimos este mismo proceso hasta llegar a la última ecuación del sistema,  $n$ , habremos obtenido la solución completa.

El siguiente código calcula la solución de un sistema triangular inferior mediante sustituciones progresivas,

```
import numpy as np

def progressive(A,b):
    ''' This function computes the solution of a lower equation system using
    progressive substitution. It receives the coefficient matrix A and the
    independent term vector b. It returns the vector solution x
    '''

    # Coefficient matrix size. Return error if not square
    [f,c]=np.shape(A)
    if f!=c:
        print("A is not square")
        return
    # To build the solution vector x
    x=np.zeros(f)
    x=b.copy()
    for i in range(f):
        '''The inner block subtracts to the independent term the previous solution
        multiplied by the correspondent coefficient'''
        for j in range(0,i):
            x[i]-=x[j]*A[i,j]
        '''Finally divide the result by the diagonal coefficient'''
        x[i]=(x[i])/A[i,i]
    return x
```

**Sistemas triangulares superiores: método de sustituciones regresivas.** En este caso, el sistema general de  $n$  ecuaciones con  $n$  incógnitas tendrá la forma general,

$$\left. \begin{array}{l} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \vdots \\ a_{nn}x_n = b_n \end{array} \right\} \Rightarrow \underbrace{\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix}}_A \cdot \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}}_x = \underbrace{\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}}_b$$

El método de resolución es idéntico al de un sistema triangular inferior, simplemente que ahora, empezamos a resolver por la última ecuación,

$$x_n = \frac{b_n}{a_{nn}}$$

reach the last equation of the system,  $n$ , we will have obtained the complete solution.

The following code calculates the solution of a lower triangular system by progressive substitutions,

**Upper triangular systems: method of backward substitutions.** In this case, the general system of  $n$  equations with  $n$  unknowns will have the general form,

The method of solution is identical to that of a lower triangular system, except that now, we start solving for the last equation,

$$x_n = \frac{b_n}{a_{nn}}$$

Y seguimos sustituyendo hacia arriba,

$$x_{n-1} = \frac{b_{n-1} - a_{(n-1)n}x_n}{a_{(n-1)(n-1)}}$$

$$x_i = \frac{b_i - \sum_{j=i+1}^n a_{ij}x_j}{a_{ii}}$$

El código para implementar este método es similar al de las sustituciones progresivas. Se deja como ejercicio el construir una función en Python que calcule la solución de un sistema triangular superior por el método de las sustituciones regresivas.

### 7.3.2. Métodos basados en las factorizaciones

Puesto que sabemos como resolver sistemas triangulares, una manera de resolver sistemas más complejos sería encontrar métodos para reducirlos a sistemas triangulares. De este modo evitamos invertir la matriz de coeficientes y los posibles problemas de estabilidad numérica derivados de dicha operación. Si podemos expresar la matriz  $A$  como el producto de matrices diagonales o triangulares podremos resolver el sistema usando los métodos anteriores. **Factorizar** una matriz es descomponer la misma en el producto de dos o más matrices de alguna forma canónica.

**Factorización LU.** La factorización LU consiste en factorizar una matriz en el producto de dos, una triangular inferior  $L$  y una triangular superior  $U$ . La factorización podía incluir pivoteo de filas, para alcanzar una solución numéricamente estable. En este caso la factorización LU toma la forma,

$$P \cdot A = L \cdot U$$

Donde  $P$  representa una matriz de permutaciones.

Podemos verlo también de otra manera,  $A = P^T \cdot L \cdot U$  ya que al ser  $P$  una matriz de permutación es invertible y  $P^{-1} = P^T$ .

Supongamos que queremos resolver un sistema de  $n$  ecuaciones lineales con  $n$  incógnitas que representamos genéricamente en forma matricial, como siempre,

$$A \cdot x = b$$

And we keep substituting upwards,

$$x_{n-1} = \frac{b_{n-1} - a_{(n-1)n}x_n}{a_{(n-1)(n-1)}}$$

$$x_i = \frac{b_i - \sum_{j=i+1}^n a_{ij}x_j}{a_{ii}}$$

The code to implement this method is similar to that of forward substitutions. It is left as an exercise to build a Python function that calculates the solution of an upper triangular system by the method of backward substitutions.

### 7.2.3. Methods based on factorisations

Since we know how to solve triangular systems, one way to solve more complex systems would be to find methods to reduce them to triangular systems. In this way we avoid inverting the matrix of coefficients and the possible problems of numerical stability derived from such an operation. If we can express the matrix  $A$  as the product of diagonal or triangular matrices we can solve the system using the above methods. To **factorise** a matrix is to decompose it into the product of two or more matrices of some canonical form.

**LU factorisation** . LU factorisation consists of factoring a matrix into the product of two, a lower triangular  $L$  and an upper triangular  $U$ . The factorisation could include pivoting rows, to achieve a numerically stable solution. In this case the LU factorisation takes the form,

$$P \cdot A = L \cdot U$$

Where  $P$  represents a matrix of permutations. We can also look at it in another way,  $A = P^T$ ,  $A = P^{-1} = P^T$ , since  $P$  is a permutation matrix and  $P^{-1} = P^T$ .

Suppose we want to solve a system of  $n$  linear equations with  $n$  unknowns that we represent generically in matrix form, as usual,

$$A \cdot x = b$$

If we compute the LU factorisation of the coefficient matrix,

Si calculamos la factorización LU de su matriz de coeficientes,

$$A \rightarrow A = P \cdot L \cdot U$$

Podemos transformar nuestro sistema de ecuaciones en uno equivalente sustituyendo  $A$  por su descomposición.

$$A \cdot x = b \rightarrow P \cdot L \cdot U \cdot x = b$$

Para poder resolver el sistema usando sustituciones progresivas y reversivas vamos a multiplicar ambos términos de la ecuación por  $P^{-1}$ , recordando que  $P^{-1} = P^T$ .

$$P^{-1} \cdot P \cdot L \cdot U \cdot x = P^{-1} \cdot b \rightarrow L \cdot U \cdot x = P^T \cdot b$$

El nuevo sistema puede resolverse en dos pasos empleando sustituciones regresivas y sustituciones progresivas. Para ello, asociamos el producto  $U \cdot x$ , a un vector de incógnitas auxiliar al que llamaremos  $z$ ,

$$U \cdot x = z$$

Si sustituimos nuestro vector auxiliar  $z$  en la expresión matricial de nuestro sistema de ecuaciones,

$$L \cdot \underbrace{U \cdot x}_z = P^T \cdot b \rightarrow L \cdot z = P^T \cdot b$$

El sistema resultante es triangular inferior, por lo que podemos resolverlo por sustituciones progresivas, y obtener de este modo los valores de  $z$ . Podemos finalmente obtener la solución del sistema a través de la definición de  $z$ ;  $U \cdot x = z$ , se trata de un sistema triangular superior, que podemos resolver mediante sustituciones regresivas.

Veamos un ejemplo. Supongamos que queremos resolver el sistema de ecuaciones lineales,

$$\begin{pmatrix} 1 & 3 & 2 \\ 2 & -1 & 1 \\ 1 & 4 & 3 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 13 \\ 3 \\ 18 \end{pmatrix}$$

En primer lugar deberíamos comprobar que la matriz de coeficiente esta bien condicionada,

$$A \rightarrow A = P \cdot L \cdot U$$

We can transform our system of equations into an equivalent replacing  $A$  by its lu factorisation,

$$A \cdot x = b \rightarrow P \cdot L \cdot U \cdot x = b$$

In order to solve the system using progressive and reversible substitutions we will multiply both terms of the equation by  $P^{-1}$ , remembering that  $P^{-1} = P^T$ .

$$P^{-1} \cdot P \cdot L \cdot U \cdot x = P^{-1} \cdot b \rightarrow L \cdot U \cdot x = P^T \cdot b$$

The new system can be solved in two steps using backward and forward substitutions. To do this, we associate the product  $U \cdot x$ , to a vector of auxiliary unknowns which we will call  $z$ ,

If we substitute our auxiliary vector  $z$  into the matrix expression of our system of equations,

$$L \cdot \underbrace{U \cdot x}_z = P^T \cdot b \rightarrow L \cdot z = P^T \cdot b$$

The resulting system is lower triangular, so we can solve it by forward substitutions, and thus obtain the values of  $z$ . We can finally obtain the solution of the system through the definition of  $z$ ;  $U \cdot x = z$ , it is an upper triangular system, which we can solve by backward substitutions.

Let us look at an example. Suppose we want to solve the system of linear equations,

First we should check that the coefficient matrix is well conditioned,

```
import numpy as np
A=np.array([[1, 3, 2],[2, -1, 1],[1, 4, 3]])
cond=np.linalg.cond(A)
print(cond)
24.382675394986972
```

No es un valor grande, con lo que podemos considerar que  $A$  está bien condicionada. Calculamos la factorización LU de la matriz de coeficientes, para ello podemos emplear el método `lu` del paquete de álgebra lineal de `scipy`. Este método nos devuelve las matrices  $P$ ,  $L$  y  $U$  que satisfacen que  $A = P \cdot L \cdot U$

```
import numpy as np
import scipy as sc

A=np.array([[1, 3, 2],[2, -1, 1],[1, 4, 3]])
b=np.array([[13],[3],[18]])
cond=np.linalg.cond(A)
P,L,U=sc.linalg.lu(A)
```

```
L: [[1.          0.          0.          ]
 [0.5         1.          0.          ]
 [0.5         0.77777778 1.          ]]
U: [[ 2.         -1.          1.          ]
 [ 0.          4.5          2.5          ]
 [ 0.          0.          -0.44444444]]
P: [[0. 0. 1.]
 [1. 0. 0.]
 [0. 1. 0.]]
```

A continuación debemos aplicar la matriz de permutaciones transpuesta, al vector de términos independientes del sistema, para poder construir el sistema equivalente  $L \cdot U \cdot x = P^T \cdot b$ ,

```
Pb=np.transpose(P).dot(b)
print("Pb: ",Pb)

Pb: [[ 3.]
 [18.]
 [13.]]
```

Empleamos la matriz  $L$  obtenida y el producto  $bp = P^T \cdot b$  que acabamos de calcular,

It is not a large value, so we can consider that  $A$  is well conditioned. We compute the LU factorisation of the coefficient matrix, for this we can use the `lu` method of the `scipy` linear algebra package. This method returns the matrices  $P$ ,  $L$  and  $U$  that satisfy that  $A = P \cdot L \cdot U$

Next we must apply the transpose matrix of permutations to the vector of independent terms of the system, in order to construct the equivalent system  $L \cdot U \cdot x = P^T \cdot b$ ,

We use the matrix  $L$  obtained and the product  $bp = P^T \cdot b$  that we have just calculated,

para obtener, por sustituciones progresivas, el vector auxiliar  $z$  descrito más arriba. Empleamos para ello la función `progressive`, cuyo código incluimos en la sección anterior,

```
z=progressive(L, bP)
print("z:",z)
```

```
z: [[ 3.
      [16.5
       [-1.33333333]]]
```

Finalmente, podemos obtener la solución del sistema sin más que aplicar el método de las sustituciones regresiva a la matriz  $U$  y al vector auxiliar  $z$  que acabamos de obtener,<sup>1</sup>

---

<sup>1</sup>La función `regressive` no se ha suministrado. Su construcción se ha dejado como ejercicio en la sección anterior.

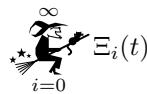
```
x=regressive(U, z)
print("x:",x)
```

```
x: [[1.]
[2.]
[3.]]
```

Para comprobar que la solución es correcta basta multiplicar la matriz de coeficientes del sistema original por el resultado obtenido para  $x$  y comprobar que obtenemos como resultado el vector de términos independientes.

```
print(A.dot(x))
```

```
[[13.
[ 3.]
[18.]]
```



Al resolver un sistema de ecuaciones mediante la factorización `lu` estamos operando únicamente sobre la matriz de coeficientes, de manera que si hay que resolver múltiples veces sistemas con esa misma matriz en los que varía el vector de términos independientes los

to obtain, by progressive substitutions, the auxiliary vector  $z$  described above. To do this we use the function `progressive`, whose code is included in the previous section,

Finally, we can obtain the solution of the system by simply applying the method of regressive substitutions to the matrix  $U$  and the auxiliary vector  $z$  that we have just obtained,<sup>1</sup>

---

<sup>1</sup>The function `regressive` has not been supplied. Its construction has been left as an exercise in the previous section.

To check that the solution is correct, it is enough to multiply the matrix of coefficients of the original system by the result obtained for  $x$  and check that we obtain as a result the vector of independent terms.

When solving a system of equations by factoring, we are only operating on the matrix of coefficients, so that if we have to solve multiple times systems with the same matrix in which the vector of independent terms varies, the calculations are minimal. These methods

cálculos son mínimos. Son métodos muy eficaces.

Además de la factorización LU hay otras maneras de factorizar matrices que son útiles para resolver sistemas de ecuaciones.

**Factorización de Cholesky.** La factorización de Cholesky permite descomponer una matriz  $A$  en el producto de una matriz triangular inferior, por su traspuesta.

$$A = L \cdot L^T$$

Para ello, es preciso que  $A$  sea simétrica y definida positiva. Una Matriz  $A_{n \times n}$  es definida positiva si dado un vector  $x$  no nulo cumple,

$$x^T \cdot A \cdot x > 0, \forall x \neq 0$$

Por tanto, en el caso particular de un sistema cuya matriz de coeficientes fuera simétrica y definida positiva, podríamos descomponerla empleando la factorización de Cholesky y resolver el sistema de modo análogo a como hicimos con la factorización LU, sustituyendo  $A$  por el producto  $L \cdot L^T$ ,

$$A \cdot x = b \rightarrow L \cdot L^T \cdot x = b$$

Definimos el vector auxiliar  $z$ ,

$$L^T \cdot x = z$$

Resolvemos por sustituciones progresivas el sistema,

$$L \cdot z = b$$

y por último obtenemos  $x$  resolviendo por sustituciones regresivas el sistema,

$$L^T \cdot x = z$$

La siguiente secuencia de código muestra la resolución en python del sistema

$$\begin{pmatrix} 2 & 5 & 1 \\ 5 & 14 & 2 \\ 1 & 2 & 6 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 15 \\ 39 \\ 23 \end{pmatrix}$$

Para ello usaremos la función `cholesky` del paquete `linalg` de `scipy`. Esta función nos devuelve por defecto la matriz triangu-

are very efficient.

In addition to factoring, there are other ways of factoring matrices that are useful for solving systems of equations.

**Cholesky factorisation** . The Cholesky factorisation allows to decompose a matrix  $A$  into the product of a lower triangular matrix, by its transpose.

$$A = L \cdot L^T$$

For this,  $A$  must be symmetric and positive definite. A Matrix  $A_{n \times n}$  is positive definite if given a non-zero vector  $x$  it satisfies,

$$x^T \cdot A \cdot x > 0, \forall x \neq 0$$

Therefore, in the particular case of a system whose coefficient matrix is symmetric and positive definite, we could decompose it using the Cholesky factorisation and solve the system in a similar way as we did with the LU factorisation, substituting  $A$  for the product  $L^T$ ,

$$A \cdot x = b \rightarrow L \cdot L^T \cdot x = b$$

We define the auxiliary vector  $z$ ,

$$L^T \cdot x = z$$

We solve the system by progressive substitutions,

$$L^T \cdot z = b$$

and finally we obtain  $x$  by solving the system by backward substitutions,

$$L^T \cdot x = z$$

Next python code sequence shows the resolution of the system

To do so, we will use the `cholesky` function from the `linalg` package of `scipy`. By default, this function returns the upper trian-

lar superior de la descomposición de cholesky  $A = U^T \cdot U$ . Para que nos devuelva la descomposición usando la triangular inferior,  $A = L \cdot L^T$  le tendremos que indicar que el parámetro `lower=True`.

Si queremos usar directamente la función `cholesky` sin indicar `lower=True`, simplemente tendremos que tener en cuenta que  $L^T = U$  y  $L = U^T$ . Por tanto, si se emplea directamente el comando `cholesky`:  $A \cdot x = b \rightarrow U^T \cdot U \cdot x = b$ .

```
A=np.array([[2., 5., 1.],[5., 14., 2.],[1., 2., 6.]])
b=np.array([[15.],[39.],[23.]])
L=sc.linalg.cholesky(A,lower=True)
print("L: ",L)
z=progressive(L, b)
print("z: ",z)
LT=np.transpose(L)
print("LT: ",LT)
x=regressive(LT,z)
print("x: ", x)
print(A.dot(x))

L: [[ 1.41421356  0.          0.          ]
 [ 3.53553391  1.22474487  0.          ]
 [ 0.70710678 -0.40824829  2.30940108]]
z: [[10.60660172]
 [ 1.22474487]
 [ 6.92820323]]
LT: [[ 1.41421356  3.53553391  0.70710678]
 [ 0.          1.22474487 -0.40824829]
 [ 0.          0.          2.30940108]]
x: [[1.]
 [2.]
 [3.]
 [[15.]
 [39.]
 [23.]]]
```

**Factorización QR** La factorización QR, descompone una matriz en el producto de una matriz ortogonal  $Q$  por una matriz triangular superior  $R$ . Una matriz  $A_{n \times n}$  es ortogonal cuando su inversa coincide con su traspuesta.

$$A^T = A^{-1}$$

Si obtenemos la factorización QR de la matriz de coeficientes de un sistema,

gular matrix of the cholesky decomposition  $A = U^T U$ . In order to return the decomposition using the lower triangular one,  $A = L \cdot L^T$ , we will have to set the parameter `lower=True`.

If we want to use directly the function `cholesky` without indicating `lower=True`, we will simply have to take into account that  $L^T = U$  and  $L = U^T$ . Therefore, if the command `cholesky` is used directly:  $A \cdot x = b \rightarrow U^T \cdot U^T \cdot x = b$ .

**QR factorisation** . The QR factorisation decomposes a matrix into the product of an orthogonal matrix  $Q$  by an upper triangular matrix  $R$ . A matrix  $A_{n \times n}$  is orthogonal when its inverse coincides with its transpose.

$$A^T = A^{-1}$$

If we obtain the QR factorisation of the coefficient matrix of a system,

$$A \cdot x = b \rightarrow Q \cdot R \cdot x = b$$

Podemos resolver ahora el sistema en dos pasos. En primer lugar, como  $Q$  es ortogonal,  $Q^{-1} = Q^T$ , podemos multiplicar por  $Q^T$  a ambos lados de la igualdad,

$$Q \cdot R \cdot x = b \rightarrow Q^T \cdot Q \cdot R \cdot x = Q^T \cdot b \rightarrow R \cdot x = Q^T \cdot b$$

Pero el sistema resultante, es un sistema triangular superior, por lo que podemos resolverlo por sustituciones regresivas. Tomando el mismo ejemplo que resolvimos antes por factorización LU,

$$\begin{pmatrix} 1 & 3 & 2 \\ 2 & -1 & 1 \\ 1 & 4 & 3 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 13 \\ 3 \\ 18 \end{pmatrix}$$

podemos ahora resolverlo mediante factorización QR. Para ello aplicamos la función `qr` incluida en el paquete `linalg` de `scipy`, a la matriz de coeficientes del sistema. A continuación multiplicamos  $Q^T$  por el vector de términos independientes y resolvemos el sistema triangular  $R \cdot x = Q^T \cdot b$  usando la función `regressive`.

```
# QR factorising
A=np.array([[1, 3, 2],[2, -1, 1],[1, 4, 3]])
b=np.array([[13],[3],[18]])
Q,R=sc.linalg.qr(A)
print("Q: ",Q)
print("R: ", R)
Qb=np.transpose(Q).dot(b)
x=regressive(R, Qb)
print("x: ",x)

Q: [[-0.40824829  0.46369464 -0.78633365]
 [-0.81649658 -0.5707011   0.08737041]
 [-0.40824829  0.67770756  0.61159284]]
R: [[-2.44948974 -2.04124145 -2.85773803]
 [ 0.          4.67261526  2.38981086]
 [ 0.          0.          0.34948162]]
x: [[1.]
 [2.]
 [3.]]
```

**Factorización SVD.** La descomposición en valores singulares de una matriz (`svd`), des-

$$A \cdot x = b \rightarrow Q \cdot R \cdot x = b$$

We can now solve the system in two steps. First, since  $Q$  is orthogonal,  $Q^{-1} = Q^T$ , we can multiply by  $Q^T$  on both sides of the equality,

$$Q \cdot R \cdot x = b \rightarrow Q^T \cdot Q \cdot R \cdot x = Q^T \cdot b \rightarrow R \cdot x = Q^T \cdot b$$

But the resulting system is an upper triangular system, so we can solve it by backward substitutions. Taking the same example we solved before by LU factorisation,

we can now solve it by QR factorisation. To do so, we apply the `qr` function included in the `linalg` package of `scipy`, to the coefficient matrix of the system. Next we multiply  $Q^T$  by the independent term vector and solve the  $R \cdot x = Q^T \cdot b$  using the `regressive` function.

**SVD factorisation**. The singular value decomposition (`svd`), decomposes any matrix in-

compone una matriz cualquiera en el producto de tres matrices,

$$A = U \cdot S \cdot V^T$$

Donde  $U$  y  $V$  son matrices ortogonales y  $S$  es una matriz diagonal con los valores singulares. Si calculamos la factorización svd de la matriz de coeficiente de un sistema,

$$A \cdot x = b \rightarrow U \cdot S \cdot V^T \cdot x = b$$

Como en el caso de la factorización QR, podemos aprovechar la ortogonalidad de las matrices  $U$  y  $V$  para simplificar el sistema,

$$U \cdot S \cdot V^T \cdot x = b \rightarrow U^T \cdot U \cdot S \cdot V^T \cdot x = U^T \cdot b \rightarrow S \cdot V^T \cdot x = U^T \cdot b$$

Como en casos anteriores, podemos crear un vector auxiliar  $z$ ,

$$V^T \cdot x = z$$

de modo que resolvemos primero el sistema,

$$S \cdot z = U^T \cdot b$$

Como la matriz  $S$  es diagonal, se trata de un sistema diagonal que, como hemos visto, es trivial de resolver.

Una vez conocido  $z$  podemos obtener la solución del sistema original, haciendo ahora uso de la ortogonalidad de la matriz  $V$ ,

$$V^T \cdot x = z \rightarrow V \cdot V^T \cdot x = V \cdot z \rightarrow x = V \cdot z$$

Volvamos a nuestro ejemplo,

$$\begin{pmatrix} 1 & 3 & 2 \\ 2 & -1 & 1 \\ 1 & 4 & 3 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 13 \\ 3 \\ 18 \end{pmatrix}$$

En primer lugar hallamos la descomposición en valores singulares (svd) de la matriz de coeficientes del sistema empleando la función `svd` del paquete `linalg` de `scipy`. Esta función nos devuelve las matrices ortogonales  $U$  y  $V^T$  y un vector fila con los autovalores de  $A$  ordenados de mayor a menor. El siguiente paso será construir la matriz  $S$  con los autovalores, para ello podemos usar la función `diag` de `numpy`: `S=np.diag(s)` y construir el vector  $U^T \cdot b$ .

to the product of three matrices,

$$A = U = U \text{sec : SVD}, S = V^T \quad (7.2)$$

Where  $U$  and  $V$  are orthogonal matrices and  $S$  is the singular value diagonal matrix. If we calculate the factorisation svd of the coefficient matrix of a system,

$$A = x = b \rightarrow U = S = V^T = x = b \quad (7.3)$$

As in the case of QR factorisation, we can take advantage of the orthogonality of the  $U$  and  $V$  matrices to simplify the system,

$$U \cdot S \cdot V^T \cdot x = b \rightarrow U^T \cdot U \cdot S \cdot V^T \cdot x = U^T \cdot b \rightarrow S \cdot V^T \cdot x = U^T \cdot b$$

As in previous cases, we can create an auxiliary vector  $z$ ,

$$V^T \cdot x = z$$

so we solve the system first,

$$S^T \cdot z = U^T \cdot b$$

Since the matrix  $S$  is diagonal, this is a diagonal system which, as we have seen, is trivial to solve.

Once  $z$  is known, we can obtain the solution of the original system, making use of the orthogonality of the matrix  $V$ ,

Back to our example,

First we find the singular value decomposition (svd) of the coefficient matrix of the system using the function `svd` from the `linalg` package of `scipy`. This function returns the orthogonal matrices  $U$  and  $V^T$  and a row vector with the eigenvalues of  $A$  ordered from largest to smallest. The next step will be to construct the matrix  $S$  with the eigenvalues, for this we can use the function `diag` of `numpy`: `S=np.diag(s)` and construct the vector  $U^T \cdot b$ .

With the matrices already constructed, we

Con las matrices ya construidas resolvemos el sistema diagonal  $S \cdot z = U^T \cdot b$  con la función `diag_sys`. Una vez conocido  $z$  calculamos  $x$  sabiendo que la matriz  $V$  es ortogonal ( $x = V \cdot z$ ).

solve the diagonal system  $S^T \cdot z = U^T \cdot b$  with the function `diag_sys`. Once  $z$  is known we calculate  $x$  knowing that the matrix  $V$  is orthogonal ( $x = V \cdot z$ ).

```
#SVD decomposition
A=np.array([[1, 3, 2],[2, -1, 1],[1, 4, 3]])
b=np.array([[13],[3],[18]])
U,s,V=sc.linalg.svd(A)
print("U: ",U)
print("V: ",V)
print("s: ",s)
S=np.diag(s)
print("S: ",S)
Utb=np.transpose(U).dot(b)
z=diag_sys(S, Utb)
print("z: ",z)
x=np.transpose(V).dot(z)
print("x:",x)
```

```
U: [[-0.59080923 -0.00525033 -0.80679421]
 [-0.04109702 -0.99848485  0.03659281]
 [-0.80576392  0.0547762  0.58969829]]
V: [[-0.23386523 -0.78353016 -0.5756627 ]
 [-0.79834737  0.49268931 -0.34626396]
 [-0.55493111 -0.3785997  0.74075213]]
s: [6.32315973 2.43934396 0.25933002]
S: [[6.32315973 0.          0.          ]
 [0.          2.43934396 0.          ]
 [0.          0.          0.25933002]]
z: [[-3.52791365]
 [-0.85176062]
 [ 0.91012588]]
x: [[1.]
 [2.]
 [3.]]
```



### 7.3.3. El método de eliminación de Gauss.

El método de eliminación gaussiana consiste en *hacer cero* todos los elementos situados por debajo de la diagonal de una matriz.

### 7.2.4. The Gaussian elimination method.

The Gaussian elimination method consists of making all the elements below the diagonal of a matrix zero. This is done by progressively

Para ello se sustituyen progresivamente las filas de la matriz, exceptuando la primera, por combinaciones adecuadas de dicha fila con las anteriores.

Veamos en qué consiste con un ejemplo. Supongamos que tenemos la siguiente matriz de orden  $4 \times 4$ ,

$$A = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 2 & 0 & 1 & -2 \\ 3 & 2 & 1 & 8 \\ 5 & 2 & 3 & 2 \end{pmatrix}$$

Si sustituimos la segunda fila por el resultado de restarle la primera multiplicada por 2 y dividida por 3 obtendríamos la matriz,

$$A = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 2 & 0 & 1 & -2 \\ 3 & 2 & 1 & 8 \\ 5 & 2 & 3 & 2 \end{pmatrix} \rightarrow [2 \ 0 \ 1 \ -2] - \frac{2}{3}[3 \ 4 \ 2 \ 5] \rightarrow U_1 = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 0 & -2.6 & -0.33 & -5.33 \\ 3 & 2 & 1 & 8 \\ 5 & 2 & 3 & 2 \end{pmatrix}$$

De modo análogo, si sustituimos ahora la tercera fila por el resultado de restarle la primera multiplicada por 3 y dividida 3,

$$U_1 = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 0 & -2.6 & -0.33 & -5.33 \\ 3 & 2 & 1 & 8 \\ 5 & 2 & 3 & 2 \end{pmatrix} \rightarrow [3 \ 2 \ 1 \ 8] - \frac{3}{3}[3 \ 4 \ 2 \ 5] \rightarrow U_1 = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 0 & -2.6 & -0.33 & -5.33 \\ 0 & -2 & -1 & 3 \\ 5 & 2 & 3 & 2 \end{pmatrix}$$

Por último si sustituimos la última fila por el resultado de restarle la primera multiplicada por 5 y dividida por 3,

$$U_1 = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 0 & -2.6 & -0.33 & -5.33 \\ 0 & -2 & -1 & -3 \\ 5 & 2 & 3 & 2 \end{pmatrix} \rightarrow [5 \ 2 \ 3 \ 2] - \frac{5}{3}[3 \ 4 \ 2 \ 5] \rightarrow U_1 = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 0 & -2.6 & -0.33 & -5.33 \\ 0 & -2 & -1 & -3 \\ 0 & -4.66 & -0.33 & -6.33 \end{pmatrix}$$

El resultado que hemos obtenido, tras realizar esta transformación, es una nueva matriz  $U$  en la que todos los elementos de su primera columna, por debajo de la diagonal, son ceros.

Podemos proceder de modo análogo para *eliminar* ahora los elementos de la segunda columna situados por debajo de la diagonal. Para ellos sustituimos la tercera fila por la diferencia entre ella y la segunda fila multipli-

replacing the rows of the matrix, except for the first row, with suitable combinations of that row with the previous rows.

Let's see what it consists of with an example. Let's suppose we have the following matrix of order  $4 \times 4$ ,

$$A = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 2 & 0 & 1 & -2 \\ 3 & 2 & 1 & 8 \\ 5 & 2 & 3 & 2 \end{pmatrix}$$

If we replace the second row by the result of subtracting the first row multiplied by 2 and divided by 3 we obtain the matrix,

$$U_1 = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 0 & -2.6 & -0.33 & -5.33 \\ 3 & 2 & 1 & 8 \\ 5 & 2 & 3 & 2 \end{pmatrix}$$

Similarly, if we now replace the third row by the result of subtracting the first row multiplied by 3 and dividing 3,

$$U_1 = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 0 & -2.6 & -0.33 & -5.33 \\ 0 & -2 & -1 & 3 \\ 5 & 2 & 3 & 2 \end{pmatrix}$$

Finally, if we replace the last row by the result of subtracting the first row multiplied by 5 and divided by 3,

$$U_1 = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 0 & -2.6 & -0.33 & -5.33 \\ 0 & -2 & -1 & -3 \\ 0 & -4.66 & -0.33 & -6.33 \end{pmatrix}$$

The result we have obtained, after performing this transformation, is a new matrix  $U$  in which all the elements of its first column, below the diagonal, are zeros.

We can proceed in a similar way to *eliminate* now the elements of the second column below the diagonal. For them we replace the third row by the difference between it and the second row multiplied by  $-2$  and divided by

cada por  $-2$  y dividida por  $-2.6$ .

|  $-2.6$ .

$$U_1 = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 0 & -2.6 & -0.33 & -5.33 \\ 0 & -2 & -1 & -3 \\ 0 & -4.66 & -0.33 & -6.33 \end{pmatrix} \rightarrow [0 \ -2 \ -1 \ -3] - \frac{-2}{-2.6}[0 \ -2.6 \ -0.33 \ -5.33] \rightarrow$$

$$U_2 = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 0 & -2.6 & -0.33 & -5.33 \\ 0 & 0 & -0.75 & 7 \\ 0 & -4.66 & -0.33 & -6.33 \end{pmatrix}$$

Y sustituyendo la última fila por la diferencia entre ella y la segunda multiplicada por  $-4.66$  y dividida por  $-2.6$ ,

$$U_2 = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 0 & -2.6 & -0.33 & -5.33 \\ 0 & -2 & -1 & -3 \\ 0 & -4.66 & -0.33 & -6.33 \end{pmatrix} \rightarrow [0 \ -4.66 \ -0.33 \ -6.33] - \frac{-4.66}{-2.6}[0 \ -2.6 \ -0.33 \ -5.33] \rightarrow$$

$$U_2 = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 0 & -2.6 & -0.33 & -5.33 \\ 0 & 0 & -0.75 & 7 \\ 0 & 0 & 0.25 & 3 \end{pmatrix}$$

De este modo, los elementos de la segunda columna situados debajo de la diagonal, han sido sustituidos por ceros. Un último paso, nos llevará hasta una matriz triangular superior; sustituimos la última fila por la diferencia entre ella y la tercera fila multiplicada por  $0.25$  y dividida por  $-0.75$ ,

And substituting the last row by the difference between it and the second row multiplied by  $-4.66$  and divided by  $-2.6$ ,

In this way, the elements of the second column below the diagonal have been replaced by zeros. A final step will lead us to an upper triangular matrix; we replace the last row by the difference between it and the third row multiplied by  $0.25$  and divided by  $-0.75$ ,

$$U_2 = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 0 & -2.6 & -0.33 & -5.33 \\ 0 & 0 & -0.75 & 7 \\ 0 & 0 & 0.25 & 3 \end{pmatrix} \rightarrow [0 \ 0 \ 0.25 \ 3] - \frac{0.25}{-0.75}[0 \ 0 \ -0.75 \ 7] \rightarrow$$

$$U_3 = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 0 & -2.6 & -0.33 & -5.33 \\ 0 & 0 & -0.75 & 7 \\ 0 & 0 & 0 & 5.33 \end{pmatrix} = U$$

Podemos ahora, a partir del ejemplo, deducir un procedimiento general. Para *eliminar* —convertir en  $0$ — el elemento  $a_{ij}$  situado por debajo de la diagonal principal,  $i > j$ :

- Dividimos los elementos de la fila  $j$  por el elemento de dicha fila que a su vez

We can now, from the example, deduce a general procedure. To *eliminate* —convert to  $0$ — the element  $a_{ij}$  below the main diagonal,  $i > j$ :

- Divide the row  $j$  elements by the diagonal element of the same row,  $a_{jj}$ , eq.

- pertenece a la diagonal,  $a_{jj}$ , eq. 7.1
2. Multiplicamos el resultado de la operación anterior por el elemento  $a_{ij}$ , eq. 7.2
  3. Finalmente, sustituimos la fila  $i$  de la matriz de partida por la diferencia entre ella y el resultado de la operación anterior, eq 7.3.
- 7.1:
2. Multiply the previous result by the element  $a_{ij}$ , eq. 7.2
  3. Finally, substitute initial matrix row  $i$  by the difference between it and the previous result, eq. 7.3.

$$[0/a_{jj} \quad 0/a_{jj} \quad \cdots \quad a_{jj}/a_{jj} \quad a_{jj+1}/a_{jj} \quad \cdots] \quad (7.1)$$

$$[a_{ij} \cdot 0/a_{jj} \quad a_{ij} \cdot 0/a_{jj} \quad \cdots \quad a_{ij} \cdot a_{jj}/a_{jj} \quad a_{ij} \cdot a_{jj+1}/a_{jj} \quad \cdots] \quad (7.2)$$

$$[0 \quad 0 \quad \cdots \quad a_{ij} \quad a_{ij+1} \quad \cdots] - [a_{ij} \cdot 0/a_{jj} \quad a_{ij} \cdot 0/a_{jj} \quad \cdots \quad a_{ij} \cdot a_{jj}/a_{jj} \quad a_{ij} \cdot a_{jj+1}/a_{jj} \quad \cdots] \quad (7.3)$$

Este procedimiento se aplica iterativamente empezando en por el elemento  $a_{21}$  de la matriz y desplazando el cálculo hacia abajo, hasta llegar a la última fila y hacia la derecha hasta llegar en cada fila al elemento anterior a la diagonal.

El siguiente código aplica el procedimiento descrito a una matriz de cualquier orden,

```
import numpy as np
def eligauss(A):
    '''This function obtains an upper triangular matrix, starting from a given
    matrix, by applying the Gaussian elimination method.
    It does not perform row pivoting'''

    #Matrix shape
    [f,c]=np.shape(A)
    U=A.copy()
    #For all the columns in A (except the last one)
    for i in range(c-1):
        # For all the rows below the diagonal
        for j in range(i+1,f):
            U[j,:]=U[j,:]-U[i,:]*U[j,i]/U[i,i]
    return U
```

La idea fundamental, es sacar partido de las siguientes propiedades de todo sistema de ecuaciones lineales;

1. Un sistema de ecuaciones lineales no cambia aunque se altere el orden de sus ecuaciones.

This procedure is applied iteratively starting at the  $a_{21}$  element of the matrix and moving the computation downwards until the last row is reached and rightwards until the element before the diagonal is reached in each row.

The following code applies the procedure described above to a matrix of any order,

The basic idea is to take advantage of the following properties of any system of linear equations;

1. A system of linear equations does not change even if the order of its equations is altered.

2. Un sistema de ecuaciones lineales no cambia aunque se multiplique cualquiera de sus ecuaciones por una constante distinta de cero.
3. Un sistema de ecuaciones no cambia si se sustituye cualquiera de sus ecuaciones por una combinación lineal de ella con otra ecuación.

Si usamos la representación matricial de un sistema de ecuaciones, Cualquiera de los cambios descritos en las propiedades anteriores afecta tanto a la matriz de coeficientes como al vector de términos independientes, por ejemplo dado el sistema,

$$\begin{pmatrix} 1 & 3 & 2 \\ 2 & -1 & 1 \\ 1 & 4 & 3 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 13 \\ 3 \\ 18 \end{pmatrix}$$

Si cambio de orden la segunda ecuación con la primera obtengo el siguiente sistema equivalente,

$$\begin{pmatrix} 2 & -1 & 1 \\ 1 & 3 & 2 \\ 1 & 4 & 3 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 13 \\ 18 \end{pmatrix}$$

Es decir, se intercambia la primera fila de la matriz de coeficientes con la segunda, y el primer elemento del vector de términos independientes con el segundo. El vector de incógnitas permanece inalterado.

Si ahora sustituimos la segunda fila, por la diferencia entre ella y la primera multiplicada por 0.5, obtenemos de nuevo un sistema equivalente,

$$\begin{pmatrix} 2 & -1 & 1 \\ 0 & 3.5 & 1.5 \\ 1 & 4 & 3 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 11.5 \\ 18 \end{pmatrix}$$

Acabamos de dar los dos primeros pasos en el proceso de eliminación de Gauss para convertir la matriz de coeficientes del sistema en una matriz triangular superior: hemos 'pivoteado' las dos primeras filas y después hemos transformado en cero el primer elemento de la segunda fila, combinándola con la primera. Hemos aplicado también esta misma combinación al segundo elemento del vector de térmi-

2. A system of linear equations does not change even if any of its equations is multiplied by a non-zero constant.
3. A system of equations does not change if you replace any of its equations by a linear combination of it with another equation.

If we use the matrix representation of a system of equations, any of the changes described in the above properties affect both the matrix of coefficients and the vector of independent terms, e.g. given the system,

If I change the order of the second equation with the first equation I obtain the following equivalent system,

That is, the first row of the coefficient matrix is exchanged with the second, and the first element of the vector of independent terms with the second. The vector of unknowns remains unchanged.

If we now substitute the second row, by the difference between it and the first row multiplied by 0.5, we obtain again an equivalent system,

We have just taken the first two steps in the Gaussian elimination process to convert the coefficient matrix of the system into an upper triangular matrix: we have 'pivoted' the first two rows and then transformed the first element of the second row to zero, combining it with the first row. We have also applied this same combination to the second element of the vector of independent terms, so that the

nos independientes, para que el sistema obtenido sea equivalente al original.

Para poder trabajar de una forma cómoda con el método de eliminación de Gauss, se suele construir una matriz, —conocida con el nombre de matriz ampliada—, añadiendo a la matriz de coeficientes, el vector de términos independientes como una columna más,

$$A, b \rightarrow AM = (A|b)$$

En nuestro ejemplo,

$$\begin{pmatrix} 1 & 3 & 2 \\ 2 & -1 & 1 \\ 1 & 4 & 3 \end{pmatrix}, \begin{pmatrix} 13 \\ 3 \\ 18 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 3 & 2 & 13 \\ 2 & -1 & 1 & 3 \\ 1 & 4 & 3 & 18 \end{pmatrix}$$

Podemos aplicar directamente a la matriz ampliada  $AM$  el programa de eliminación de Gauss, `eligauss` anterior ???. Si aplicamos `eligauss` a la matriz ampliada del sistema del ejemplo. Para construir la matriz ampliada podemos usar el método `concatenate` de `numpy` indicando que el eje en el que deseamos concatenar  $A$  y  $b$  es el 1 (columnas):  $AM=np.concatenate((A,b),axis=1)$

```
A=np.array([[1., 3., 2.],[2., -1., 1.],[1., 4., 3.]])
b=np.array([[13.],[3.],[18.]])
# Extended matrix
AM=np.concatenate((A,b),axis=1)
print("AM: ",AM)
GA=eligauss(AM)
print("GA: ",GA)

AM: [[ 1.  3.  2.  13.]
 [ 2. -1.  1.  3.]
 [ 1.  4.  3.  18.]]
GA: [[ 1.          3.          2.          13.         ]
 [ 0.         -7.         -3.         -23.        ]
 [ 0.          0.        0.57142857  1.71428571]]
```

El programa ha obtenido como resultado una nueva matriz en la que los elementos situados por debajo de la diagonal son ahora cero. Podemos reconstruir, a partir del resultado obtenido un sistema equivalente actual Separando la última columna del resultado,

$$\begin{pmatrix} 1 & 3 & 2 & 13 \\ 0 & -7 & -3 & -23 \\ 0 & 0 & 0.57142857 & 1.71428571 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 3 & 2 \\ 0 & -7 & -3 \\ 0 & 0 & 0.571 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 13 \\ -23 \\ 1.7143 \end{pmatrix}$$

system obtained is equivalent to the original.

In order to work comfortably with the Gaussian elimination method, a matrix is usually constructed, known as an extended matrix, by adding the vector of independent terms to the coefficient matrix as an additional column,

$$A, b \rightarrow AM = (A|b)$$

In our example,

We can apply directly to the extended matrix  $AM$  the Gaussian elimination program, `eligauss` ???. If we apply `eligauss` to the extended matrix of the system of the example. To construct the extended matrix we can use the `concatenate` method of `numpy` indicating that the axis on which we want to concatenate  $A$  and  $b$  is 1 (columns):  $AM=np.concatenate((A,b),axis=1)$

The program has obtained as a result a new matrix in which the elements below the diagonal are now zero. We can reconstruct, from the result obtained, a current equivalent system by separating the last column of the result,

El sistema resultante de la eliminación de Gauss es triangular superior, con lo que podemos resolverlo directamente mediante sustituciones regresivas,

```
RA=GA[:,0:3]
nb=GA[:,3]
print("RA: ",RA)
print("nb: ",nb)
x=regressive(RA, nb)
print("x: ",x)

RA: [[ 1.        3.        2.        ]
 [ 0.       -7.       -3.        ]
 [ 0.        0.       0.57142857]]
nb: [ 13.       -23.        1.7142857]
x: [1. 2. 3.]
```

El programa `eligauss` presenta un problema: ¿qué ocurre si el elemento de la diagonal es cero o un valor relativamente pequeño?. Si eso ocurre es posible que no pueda completarse el proceso de eliminación o bien que los cálculos sean inestables. Para ello vamos a incluir el pivoteo de filas, es decir, reordenaremos las filas de la matriz para que el elemento de la diagonal sea mayor que ese mismo elemento de las filas posteriores. A continuación, incluimos una versión modificada de `eligauss` que incluye el pivoteo de filas.

```
def eligauss(A):
    '''This function obtains an upper triangular matrix, starting from a given
    matrix, by applying the Gaussian elimination method.
    It includes row pivoting. If the diagonal element is less than the same
    element in next rows, rows are interchanged'''
```

```
#Matrix shape
[f,c]=np.shape(A)
U=A.copy()
#For all the columns in A (except the last one)
for i in range(c-1):
    #Row pivoting
    # Search the maximum in column i
    maxcol= np.abs(U[i,i])
    index = i
    for l in range(i,f):
        if np.abs(U[l,i])>maxcol:
            maxcol=np.abs(U[l,i])
            index=l
    # If we have found an element U[l,i] greater than U[i,i] we interchange
```

The system resulting from Gaussian elimination is upper triangular, so we can solve it directly by backward substitutions,

The `eligauss` program has a problem: what happens if the diagonal element is zero or a relatively small value? If that happens, it is possible that the elimination process cannot be completed or that the calculations are unstable. For this we will include row pivoting, i.e. we will rearrange the rows of the matrix so that the element on the diagonal is larger than the same element in the rows after it. Below, we include a modified version of `eligauss` that includes row pivoting.

```

# row l with row i
if index!=i:
    aux=np.array([U[i,:]])
    U[i,:]=U[index,:]
    U[index,:]=aux[:]
# End of Row pivoting
# For all the rows below the diagonal
print(U)
for j in range(i+1,f):
    U[j,:]=U[j,:]-U[i,:]*U[j,i]/U[i,i]
return U

```

Podemos aplicar esta función a nuestro ejemplo de siempre y separando en la matriz ampliada la matriz de coeficientes y el vector de términos independientes podemos resolver el sistema por sustituciones regresivas,

```

A=np.array([[1., 3., 2.],[2., -1., 1.],[1., 4., 3.]])
b=np.array([[13.],[3.],[18.]])
# Extended matrix
AM=np.concatenate((A,b),axis=1)
print("AM: ",AM)
GA=eligaussp(AM)
print("GA: ",GA)
RA=GA[:,0:3]
nb=GA[:,3]
print("RA: ",RA)
print("nb: ",nb)
x=regressive(RA, nb)
print("x: ",x)

AM: [[ 1.  3.  2.  13.]
 [ 2. -1.  1.  3.]
 [ 1.  4.  3.  18.]]
[[ 2. -1.  1.  3.]
 [ 1.  3.  2.  13.]
 [ 1.  4.  3.  18.]]
[[ 2. -1.  1.  3.]
 [ 0.  4.5  2.5  16.5]
 [ 0.  3.5  1.5  11.5]]
[[ 2.          -1.          1.          3.          ]
 [ 0.          4.5          2.5          16.5         ]
 [ 0.          0.          -0.44444444 -1.33333333]]
GA: [[ 2.          -1.          1.          3.          ]
 [ 0.          4.5          2.5          16.5         ]
 [ 0.          0.          -0.44444444 -1.33333333]]
RA: [[ 2.          -1.          1.          ]
 [ 0.          4.5          2.5          ]
 [ 0.          0.          -0.44444444]]

```

We can apply this function to our usual example and by separating the coefficient matrix and the vector of independent terms in the extended matrix we can solve the system by backward substitutions,

nb: [ 3.                16.5                -1.33333333 ]  
x: [ 1. 2. 3.]

### 7.3.4. Gauss-Jordan y matrices en forma reducida escalonada

El método de eliminación de Gauss, permite obtener a partir de una matriz arbitraria, una matriz triangular superior. Una vez obtenida ésta, podríamos seguir transformando la matriz de modo que hiciéramos cero todos los elementos situados por encima de su diagonal. Para ello bastaría aplicar el mismo método de eliminación de Gauss, la diferencia es que ahora eliminaríamos -haríamos ceros- los elementos situados por encima de la diagonal principal de la matriz, empezando por la última columna y moviéndonos de abajo a arriba y de derecha a izquierda. Este proceso se conoce con el nombre de eliminación de Gauss-Jordan. Por ejemplo supongamos que partimos de la matriz que acabamos de obtener por eliminación de Gauss en ejemplo anterior,

$$GA = \begin{pmatrix} 1 & 4 & 3 & 18 \\ 0 & -1 & -1 & -5 \\ 0 & 0 & 4 & 12 \end{pmatrix}$$

Empezaríamos por hacer cero el elemento  $ga_{23}$  que es el que está situado encima del último elemento de la diagonal principal, para ello restaríamos a la segunda columna la tercera multiplicada por  $-1$  y dividida por  $4$ ,

$$\left( \begin{array}{cccc} 1 & 4 & 3 & 18 \\ 0 & -1 & -1 & -5 \\ 0 & 0 & 4 & 12 \end{array} \right) = \left( \begin{array}{cccc} 1 & 4 & 3 & 18 \\ 0 & -1 & 0 & -2 \\ 0 & 0 & 4 & 12 \end{array} \right)$$

A continuación, eliminaríamos el elemento situado en la misma columna, una fila por encima. Para ello restamos a la primera la tercera multiplicada por  $3$  y dividida por  $4$ ,

$$\left( \begin{array}{cccc} 1 & 4 & 3 & 18 \\ 0 & -1 & 0 & -2 \\ 0 & 0 & 4 & 12 \end{array} \right) = \left( \begin{array}{cccc} 1 & 4 & 0 & 9 \\ 0 & -1 & 0 & -2 \\ 0 & 0 & 4 & 12 \end{array} \right)$$

Como hemos llegado a la primera columna, pasamos a eliminar los elementos de la siguiente columna de la izquierda. En este ejem-

### 7.2.5. Gauss-Jordan and matrices in stepwise reduced form

The Gaussian elimination method allows us to obtain an upper triangular matrix from an arbitrary matrix. Once this has been obtained, we could continue transforming the matrix in such a way as to make all the elements located above its diagonal zero. To do this, we would simply apply the same Gaussian elimination method, the difference being that we would now eliminate - make zeros - the elements located above the main diagonal of the matrix, starting with the last column and moving from bottom to top and from right to left. This process is known as Gauss-Jordan elimination. For example, suppose we start from the matrix we have just obtained by Gaussian elimination in the previous example,

$$GA = \begin{pmatrix} 1 & 4 & 3 & 18 \\ 0 & -1 & -1 & -5 \\ 0 & 0 & 4 & 12 \end{pmatrix}$$

We would start by making the element  $ga_{23}$ , which is the one located above the last element of the main diagonal, zero. To do this we would subtract the third column multiplied by  $-1$  and divided by  $4$  from the second column,

$$\left( \begin{array}{cccc} 1 & 4 & 3 & 18 \\ 0 & -1 & 0 & -2 \\ 0 & 0 & 4 & 12 \end{array} \right) = \left( \begin{array}{cccc} 1 & 4 & 3 & 18 \\ 0 & -1 & 0 & -2 \\ 0 & 0 & 4 & 12 \end{array} \right)$$

Next, we would eliminate the element located in the same column, one row above. To do this we subtract from the first the third multiplied by  $3$  and divided by  $4$ ,

$$\left( \begin{array}{cccc} 1 & 4 & 0 & 9 \\ 0 & -1 & 0 & -2 \\ 0 & 0 & 4 & 12 \end{array} \right) = \left( \begin{array}{cccc} 1 & 4 & 0 & 9 \\ 0 & -1 & 0 & -2 \\ 0 & 0 & 4 & 12 \end{array} \right)$$

As we have reached the first column, we move on to remove the elements in the next column on the left. In this example we only

Plano solo tenemos un elemento por encima de la diagonal. Para eliminarlo restamos de la primera fila la segunda multiplicada por 4 y dividida por  $-1$ ,

$$\begin{pmatrix} 1 & 0 & 4 - 4 \cdot (-1)/(-1) & 0 & 9 - (-2) \cdot 4/(-1) \\ 0 & & -1 & 0 & -2 \\ 0 & & 0 & 4 & 12 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & -1 & 0 & -2 \\ 0 & 0 & 4 & 12 \end{pmatrix}$$

Si ahora separamos en la matriz ampliada resultante, la matriz de coeficientes y el vector de términos independientes, obtenemos un sistema diagonal, cuya solución es trivial,

$$\begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & -1 & 0 & -2 \\ 0 & 0 & 4 & 12 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 4 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ -2 \\ 12 \end{pmatrix} \Rightarrow x = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

El siguiente programa `gauss-jordan`, añade las líneas de código necesarias a `eligausspp` para realizar la eliminación de Gauss-Jordan completa de una matriz.

```
def gaussjordan(A):
    '''This function implements gauss-jordan elimination to obtain a diagonal
    matrix'''

    #Matrix shape
    [f,c]=np.shape(A)
    U=A.copy()
    # Step 1: reduce matrix A to a triangular matrix
    #For all the columns in A (except the last one)
    for i in range(c-1):
        #Row pivoting
        # Search the maximum in column i
        maxcol= np.abs(U[i,i])
        index = i
        for l in range(i,f):
            if np.abs(U[l,i])>maxcol:
                maxcol=np.abs(U[l,i])
                index=l
        # If we have found an element U[l,i] greater than U[i,i] we interchange
        # row l with row i
        if index!=i:
            aux=np.array([U[i,:]])
            U[i,:]=U[index,:]
            U[index,:]=aux[:]
        # End of Row pivoting
        # For all the rows below the diagonal
        print(U)
        for j in range(i+1,f):
            U[j,:]=U[j,:]-U[i,:]*U[j,i]/U[i,i]
```

have one element above the diagonal. To eliminate it we subtract from the first row the second row multiplied by 4 and divided by  $-1$ ,

If we now separate the coefficient matrix and the vector of independent terms into the resulting expanded matrix, we obtain a diagonal system whose solution is trivial,

The following program, `gauss-jordan`, adds the necessary lines of code to `eligausspp` to perform the complete Gauss-Jordan elimination of a matrix.

```
# Step 2: obtain the diagonal matrix
# For all the columns begining by the end
for i in range(c-2,-1,-1):
    # For all the rows above the diagonal
    for j in range(i-1,-1,-1):
        U[j,:]=U[j,:]-U[i,:]*U[j,i]/U[i,i]
return U
```

Si tras aplicar la eliminación de Gauss-Jordan a la matriz ampliada de un sistema, dividimos cada fila por el elemento que ocupa la diagonal principal, obtendríamos en la última columna las soluciones del sistema. En nuestro ejemplo,

$$\begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \end{pmatrix}$$

La matriz resultante se dice que esta en *forma escalonada reducida por filas*. Se deja como un ejercicio, añadir el código necesario al programa anterior para que dé como resultado la *forma escalonada reducida por filas* de la matriz ampliada de un sistema.

## 7.4. Métodos iterativos

Los métodos iterativos se basan en una aproximación distinta al problema de la resolución de sistemas de ecuaciones lineales. La idea en todos ellos es buscar un método que, a partir de un valor inicial para la solución del sistema, vaya refinándolo progresivamente, acercándose cada vez más a la solución real. La figura 7.4, muestra un diagrama de flujo general para todos los métodos iterativos de resolución de sistemas.

Siguiendo el diagrama de flujo, el primer paso, es proponer un vector con soluciones del sistema. Si se conocen valores próximos a las soluciones reales se eligen dichos valores como solución inicial. Si, como es habitual, no se tiene idea alguna de cuáles son las soluciones,

If, after applying the Gauss-Jordan elimination to the extended matrix of a system, we divide each row by the element occupying the main diagonal, we obtain the solutions of the system in the last column. In our example,

The resulting matrix is said to be in *row echelon form*. It is left as an exercise to add the necessary code to the above program to result in the *row-reduced echelon form* of the expanded matrix of a system.

## 7.3. Iterative methods

Iterative methods are based on a different approach to the problem of solving systems of linear equations. The idea in all of them is to find a method that, starting from an initial value for the solution of the system, refines it progressively, getting closer and closer to the real solution. Figure 7.4, shows a general flow diagram for all iterative methods of solving systems.

Following the flow chart, the first step is to propose a vector with solutions of the system. If values close to the real solutions are known, these values are chosen as the initial solution. If, as usual, you have no idea what the solutions are, it is most usual to start with

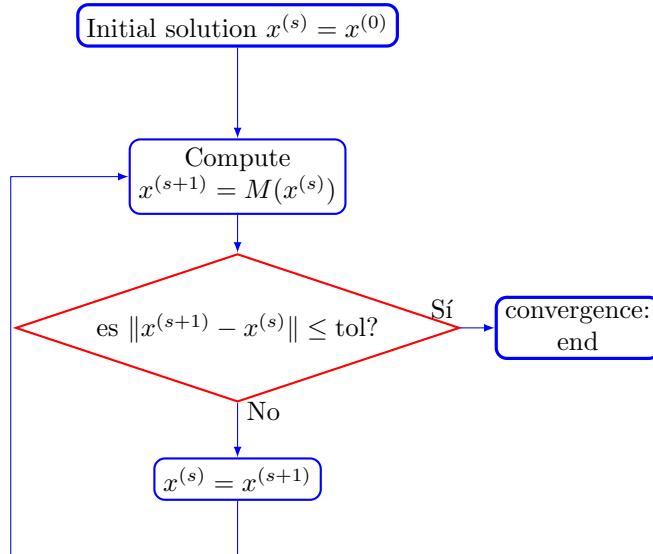


Figura 7.4: Diagrama de flujo general de los métodos iterativos para resolver sistemas de ecuaciones. La función  $M(x)$  es la que especifica en cada caso el método.

Figure 7.4: General flowchart of iterative methods for solving systems of equations. The function  $M(x)$  is the one that specifies the method in each case.

lo más habitual es empezar con el vector (0),

$$x^{(0)} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

A partir de la primera solución, se calcula una segunda, siguiendo la especificaciones concretas del método que se esté usando. En el diagrama de flujo se ha representado de modo genérico el método mediante la función  $M(\cdot)$ . Dedicaremos buena parte de esta sección a estudiar algunos de los métodos más usuales.

Una vez que se tienen dos soluciones se comparan. Para ello, se ha utilizado el módulo del vector diferencia de las dos soluciones. Este módulo nos da una medida de cuanto se parecen entre sí las dos soluciones. Si la diferencia es suficientemente pequeña, –menor que un cierto valor de tolerancia  $tol$ – damos la solución por buena y el algoritmo termina. En caso contrario, copiamos la última solución obtenida en la penúltima y repetimos todo el proceso. El bucle se repite hasta que se cumpla la condición de que el módulo de la diferencia

the vector (0),

$$x^{(0)} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

From the first solution, a second solution is calculated, following the specific specifications of the method being used. In the flowchart, the method is represented in a generic way by the function  $M(\cdot)$ . We will spend a good part of this section looking at some of the more common methods.

Once two solutions are available, they are compared. To do this, the modulus of the difference vector of the two solutions has been used. This modulus gives us a measure of how much the two solutions resemble each other. If the difference is small enough, –less than a certain tolerance value  $tol$ – we consider the solution to be good and the algorithm terminates. Otherwise, we copy the last solution obtained into the penultimate one and repeat the whole process. The loop repeats until the condition that the modulus of the difference of the dif-

de dos soluciones sucesivas sea menor que la tolerancia establecida.

Una pregunta que surge de modo inmediato del esquema general que acabamos de introducir es si el proceso descrito converge; y, caso de hacerlo, si converge a la solución correcta del sistema.

La respuesta es que los sistemas iterativos no siempre convergen, es decir, no está garantizado que tras un cierto número de iteraciones la diferencia entre dos soluciones sucesivas sea menor que un valor de tolerancia arbitrario. La convergencia, como veremos más adelante, dependerá tanto del sistema que se quiere resolver como del método iterativo empleado. Por otro lado, lo que sí se cumple siempre es que, si el método converge, las sucesivas soluciones obtenidas se van aproximando a la solución real del sistema.

#### 7.4.1. El método de Jacobi.

**Obtención del algoritmo.** Empezaremos por introducir el método iterativo de Jacobi, por ser el más intuitivo de todos. Para introducirlo, emplearemos un ejemplo sencillo. Supongamos que queremos resolver el siguiente sistema de ecuaciones,

$$\begin{aligned} 3x_1 + 3x_2 &= 6 \\ 3x_1 + 4x_2 &= 7 \end{aligned}$$

Supongamos que supiéramos de antemano el valor de  $x_2$ , para obtener  $x_1$ , bastaría entonces despejar  $x_1$ , por ejemplo de la primera ecuación, y sustituir el valor conocido de  $x_2$ ,

$$x_1 = \frac{6 - 3x_2}{3}$$

De modo análogo, si conocieráramos previamente  $x_1$ , podríamos despejar  $x_2$ , ahora por ejemplo de la segunda ecuación, y sustituir el valor conocido de  $x_1$ ,

$$x_2 = \frac{7 - 3x_1}{4}$$

El método de Jacobi lo que hace es *suponer* conocido el valor de  $x_2$ ,  $x_2^{(0)} = 0$ , y con él

difference of two successive solutions is less than the set tolerance is fulfilled.

One question that immediately arises from the general scheme just introduced is whether the process described converges; and, if it does, whether it converges to the correct solution of the system.

The answer is that iterative systems do not always converge, that is, it is not guaranteed that after a certain number of iterations the difference between two successive solutions will be less than an arbitrary tolerance value. Convergence, as we will see later, will depend both on the system to be solved and on the iterative method used. On the other hand, what is always true is that, if the method converges, the successive solutions obtained approach the real solution of the system.

#### 7.3.1. Jacobi's method

**Obtaining the algorithm.** We will begin by introducing Jacobi's iterative method, as it is the most intuitive of all. To introduce it, we will use a simple example. Suppose we want to solve the following system of equations,

Suppose we knew the value of  $x_2$  beforehand, to obtain  $x_1$ , it would then suffice to clear  $x_1$ , for example from the first equation, and substitute the known value of  $x_2$ ,

In the same way, if we knew previously  $x_1$ , we could clear  $x_2$ , now in the second equation and substitute the  $x_1$  known value.

Jacobi's method does this by assuming the known value of  $x_2$ ,  $x_2^{(0)} = 0$ , and using it to

obtener un valor de  $x_1^{(1)}$ ,

| obtain a value of  $x_1^{(1)}$ ,

$$x_1^{(1)} = \frac{6 - 3x_2^{(0)}}{3} = \frac{6 - 3 \cdot 0}{3} = 2$$

a continuación *supone* conocido el valor de  $x_1$ ,  $x_1^{(0)} = 0$  y con él obtiene un nuevo valor para  $x_2$ ,

| then  $x_1, x_1^{(0)} = 0$  and with it obtains a new value for  $x_2$ ,

$$x_2^{(1)} = \frac{7 - 3x_1^{(0)}}{4} = \frac{7 - 3 \cdot 0}{4} = 1.75$$

En el siguiente paso, tomamos los valores obtenidos,  $x_1^{(1)}$  y  $x_2^{(1)}$  como punto de partida para calcular unos nuevos valores,

| In the next step, we take the obtained values,  $x_1^{(1)}$  and  $x_2^{(1)}$  as a starting point to calculate new values,

$$\begin{aligned} x_1^{(2)} &= \frac{5 - 3x_2^{(1)}}{3} = \frac{6 - 3 \cdot 1.75}{3} = 0.25 \\ x_2^{(2)} &= \frac{7 - 3x_1^{(1)}}{4} = \frac{7 - 3 \cdot 1.67}{4} = 0.25 \end{aligned}$$

y en general,

| and in general,

$$\begin{aligned} x_1^{(s+1)} &= \frac{6 - 3x_2^{(s)}}{3} \\ x_2^{(s+1)} &= \frac{7 - 3x_1^{(s)}}{4} \end{aligned}$$

Si repetimos el mismo cálculo diez veces obtenemos,

| If we repeat the same calculation ten times we get,

$$\begin{aligned} x_1^{(10)} &= 0.7627 \\ x_2^{(10)} &= 0.7627 \end{aligned}$$

Si lo repetimos veinte veces,

| If we repeat it twenty times,

$$\begin{aligned} x_1^{(20)} &= 0.9437 \\ x_2^{(20)} &= 0.9437 \end{aligned}$$

La solución exacta del sistema es  $x_1 = 1$ ,  $x_2 = 1$ . Según aumentamos el número de iteraciones, vemos cómo las soluciones obtenidas se aproximan cada vez más a la solución real.

A partir del ejemplo, podemos generalizar el método de Jacobi para un sistema cualquiera de  $n$  ecuaciones con  $n$  incógnitas,

| The exact solution of the system is  $x_1 = 1$ ,  $x_2 = 1$ . As we increase the number of iterations, we see how the solutions obtained get closer and closer to the real solution.

| From the example, we can generalise Jacobi's method for any system of  $n$  equations with  $n$  unknowns,

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ \cdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

La solución para la incógnita  $x_i$  en la iteración,  $s + 1$  a partir de la solución obtenida en la iteración  $s$  toma la forma,

$$x_i^{(s+1)} = \frac{b_i - \sum_{j \neq i} a_{ij} x_j^{(s)}}{a_{ii}}$$

A continuación, incluimos el código correspondiente al cálculo de una iteración del algoritmo de Jacobi. Este código es el que corresponde, para el método de jacobi, a la función  $M(\cdot)$  del diagrama de flujo de la figura 7.4.

```
# For all the equations
xs1=b.copy()
for i in range(nf):
    for j in range(i):
        xs1[i]-=A[i,j]*xs[j]

    for j in range(i+1,nf):
        xs1[i]-=A[i,j]*xs[j]
    xs1[i]=xs1[i]/A[i,i]
```

**Expresión matricial para el método de Jacobi.** El método de Jacobi que acabamos de exponer puede expresarse también en forma matricial. En python, el empleo en forma matricial, tiene la ventaja de ahorrar bucles en el cálculo de la nueva solución a partir de la anterior. Si expresamos un sistema general de orden  $n$  en forma matricial,

$$\underbrace{\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}}_A \cdot \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}}_x = \underbrace{\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}}_b$$

Podríamos separar en tres sumandos la expresión de la izquierda del sistema: una matriz diagonal  $D$ , una matriz estrictamente diagonal superior  $U$ , y una matriz estrictamente triangular inferior  $L$ ,

The solution for the unknown  $x_i$  in iteration  $s + 1$  from the solution obtained in iteration  $s$  takes the form,

Below, we include the code corresponding to the calculation of one iteration of Jacobi's algorithm. This code corresponds, for Jacobi's method, to the function  $M(\cdot)$  in the flowchart in figure 7.4.

**Matrix expression for Jacobi's method**. Jacobi's method as described above can also be expressed in matrix form. In python, the use of matrix form has the advantage of saving loops in the calculation of the new solution from the previous one. If we express a general system of order  $n$  in matrix form,

We could separate the left-hand expression of the system into three summands: a diagonal matrix  $D$ , a strictly upper diagonal matrix  $U$ , and a strictly lower triangular matrix  $L$ ,

$$\left[ \underbrace{\begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix}}_D + \underbrace{\begin{pmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix}}_U + \underbrace{\begin{pmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{pmatrix}}_L \right] \cdot \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}}_x = \underbrace{\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}}_b$$

Si pasamos los términos correspondientes a las dos matrices triangulares al lado derecho de la igualdad,

$$\begin{aligned} & \overbrace{\begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix}}^D \cdot \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}}_x = \\ &= \underbrace{\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}}_b - \left[ \underbrace{\begin{pmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix}}_U + \underbrace{\begin{pmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{pmatrix}}_L \right] \cdot \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}}_x \end{aligned}$$

Si examinamos las filas de las matrices resultantes a uno y otro lado del igual, es fácil ver que cada una de ellas coincide con la expresión correspondiente a una iteración del método de Jacobi, sin mas que cambiar los valores de las incógnitas a la izquierda del igual por  $x^{(s+1)}$  y la derecha por  $x^{(s)}$ ,

If we pass the terms corresponding to the two triangular matrices to the right-hand side of the equality,

If we examine the rows of the resulting matrices on either side of the equal, it is easy to see that each of them coincides with the expression corresponding to an iteration of Jacobi's method, simply by changing the values of the unknowns on the left of the equal by  $x^{(s+1)}$  and on the right by  $x^{(s)}$ ,

$$\begin{aligned} & \overbrace{\begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix}}^D \cdot \underbrace{\begin{pmatrix} x_1^{(s+1)} \\ x_2^{(s+1)} \\ \vdots \\ x_n^{(s+1)} \end{pmatrix}}_{x^{(s+1)}} = \\ &= \underbrace{\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}}_b - \left[ \underbrace{\begin{pmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix}}_U + \underbrace{\begin{pmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{pmatrix}}_L \right] \cdot \underbrace{\begin{pmatrix} x_1^s \\ x_2^s \\ \vdots \\ x_n^s \end{pmatrix}}_{x^s} \end{aligned}$$

y multiplicar a ambos lados por la inversa de la matriz  $D^2$ ,

and multiply both sides by the inverse of the matrix  $D^2$

<sup>2</sup> $D$  es una matriz diagonal su inversa se calcula trivialmente sin más cambiar cada elemento de la diagonal por su inverso.

<sup>2</sup> $D$  is a diagonal matrix and its inverse is trivially calculated by simply changing each element of the diagonal by its inverse,

$$\underbrace{\begin{pmatrix} x_1^{(s+1)} \\ x_2^{(s+1)} \\ \vdots \\ x_n^{(s+1)} \end{pmatrix}}_{x^{(s+1)}} = \underbrace{\begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix}}_D^{-1} \cdot$$

$$\cdot \left[ \underbrace{\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}}_b - \left[ \underbrace{\begin{pmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix}}_U + \underbrace{\begin{pmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{pmatrix}}_L \right] \cdot \underbrace{\begin{pmatrix} x_1^s \\ x_2^s \\ \vdots \\ x_n^s \end{pmatrix}}_{x^s} \right]$$

El resultado final es una operación matricial,

$$x^{(s+1)} = D^{-1} \cdot b - D^{-1} \cdot (L + U) \cdot x^s$$

que equivale al código para una iteración y, por tanto a la función  $M(\cdot)$  del método de Jacobi.

Si analizamos la ecuación anterior, observamos que el término,  $D^{-1} \cdot b$  es fijo. Es decir, permanece igual en todas las iteraciones que necesitemos realizar para que la solución converja. El segundo término, tiene también una parte fija,  $-D^{-1}(L+U)$ . Se trata de una matriz de orden  $n$ , igual al del sistema que tratamos de resolver. Esta matriz, recibe el nombre de matriz del método, se suele representar por la letra  $H$  y como veremos después esta directamente relacionada con la convergencia del método.

Si queremos implementar en python el método de Jacobi en forma matricial, lo más eficiente es que calculemos en primer lugar las matrices  $D, L, U, f = D^{-1} \cdot b$  y  $H$ . Una vez calculadas, empleamos el resultado para aproximar la solución iterativamente. Veamos con un ejemplo cómo construir la matrices. Supongamos que tenemos el siguiente sistema,

$$\begin{pmatrix} 4 & 2 & -1 \\ 3 & -5 & 1 \\ 1 & -1 & 6 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 5 \\ -4 \\ 17 \end{pmatrix}$$

En primer lugar, calculamos la matriz  $D$ , a partir de la matriz de coeficientes del sistema, empleando la función `diag` de `numpy`. Aplicando `diag` a una matriz extraemos en un vector los elementos de su diagonal,

Aplicando `diag` a un vector construimos una matriz con los elementos del vector colocados sobre la diagonal de la matriz. El resto de los elementos son cero.

Si aplicamos dos veces `diag` sobre la matriz de coeficientes de un sistema, obtenemos directamente la matriz  $D$ ,

```
A=np.array([[4., 2., -1.], [3., -5., 1.], [1., -1., 6.]])
```

```
np.diag(A)
Out[1]: array([ 4., -5.,  6.])
```

The final result is a matrix operation,

which is equivalent to the code for one iteration and, therefore, to the function  $M(\cdot)$  of Jacobi's method.

If we analyse the above equation, we observe that the term,  $D^{-1} \cdot b$  is fixed. That is, it remains the same in all the iterations we need to perform for the solution to converge. The second term also has a fixed part,  $-D^{-1}(L+U)$ . This is a matrix of order  $n$ , equal to that of the system we are trying to solve. This matrix, called the method matrix, is usually represented by the letter  $H$  and, as we shall see later, it is directly related to the convergence of the method.

If we want to implement Jacobi's method in python in matrix form, the most efficient way is to first calculate the matrices  $D, L, U, f = D^{-1} \cdot b$  and  $H$ . Once calculated, we use the result to approximate the solution iteratively. Let's see with an example how to construct the matrices. Suppose we have the following system,

First, we calculate the matrix  $D$ , from the coefficient matrix of the system, using the `diag` function of `numpy`. Applying `diag` to a matrix, we extract the elements of its diagonal into a vector,

Applying `diag` to a vector constructs a matrix with the elements of the vector placed on the diagonal of the matrix. The rest of the elements are zero.

If we apply `diag` twice on the matrix of coefficients of a system, we obtain directly the matrix  $D$ ,

```
np.diag(np.diag(A))
Out[2]:
array([[ 4.,  0.,  0.],
       [ 0., -5.,  0.],
       [ 0.,  0.,  6.]])
```

Para calcular las matrices  $L$  y  $U$ , podemos emplear las funciones de numpy `triu` y `tril` que extraen una matriz triangular superior y una matriz triangular inferior, respectivamente, a partir de una matriz dada.

```
np.triu(A)
Out[3]:
array([[ 4.,  2., -1.],
       [ 0., -5.,  1.],
       [ 0.,  0.,  6.]])
```

```
np.tril(A)
Out[4]:
array([[ 4.,  0.,  0.],
       [ 3., -5.,  0.],
       [ 1., -1.,  6.]])
```

Las matrices  $L$  y  $U$  son estrictamente triangulares superior e inferior. Podemos obtenerlas restando a las matrices que acabamos de obtener la matriz  $D$ ,

```
np.triu(A)-np.diag(np.diag(A))
Out[5]:
array([[ 0.,  2., -1.],
       [ 0.,  0.,  1.],
       [ 0.,  0.,  0.]])
```

```
np.tril(A)-np.diag(np.diag(A))
Out[6]:
array([[ 0.,  0.,  0.],
       [ 3.,  0.,  0.],
       [ 1., -1.,  0.]])
```

También podemos obtenerlas directamente, empleando solo la matriz de coeficientes,

```
A-np.tril(A)
Out[7]:
array([[ 0.,  2., -1.],
       [ 0.,  0.,  1.],
       [ 0.,  0.,  0.]])
```

```
A-np.triu(A)
```

To compute the  $L$  and  $U$  matrices, we can use the numpy `triu` and `tril` which extract an upper triangular matrix and a lower triangular matrix, respectively, from a given matrix.

The matrices  $L$  and  $U$  are strictly upper and lower triangular. We can obtain them by subtracting the matrix  $D$  from the matrices we have just obtained,

We can also obtain them directly, using only the coefficient matrix,

```
Out[8]:
array([[ 0.,  0.,  0.],
       [ 3.,  0.,  0.],
       [ 1., -1.,  0.]])
```

Construimos a continuación el vector  $f$ ,  $f=(\text{np.linalg.inv}(A)).dot(b)$  y la matriz del sistema  $H H=-(\text{np.linalg.inv}(A)).dot(U+L)$

A partir de las matrices construidas, el código para calcular una iteración por el método de Jacobi sería simplemente,  $\mathbf{x}_1=\mathbf{f}+\mathbf{H}.\text{dot}(\mathbf{x}_0)$

En el siguiente fragmento de código se reúnen todas las operaciones descritas,

```
def jacobi(A,b,x0,tol,itmax):
    [nf,nc]=np.shape(A)
    xs=x0.copy()
    xs1=b.copy()
    it=0
    # Build the matrices D,U,f,H
    D=np.diag(np.diag(A))
    U=np.triu(A)-D
    L=np.tril(A)-D
    invD=np.linalg.inv(D)
    f=(invD).dot(b)
    H=-invD.dot(L+U)
    # First iteration
    xs1=f+H.dot(xs)
    error=np.linalg.norm(xs1-xs)
    #Iteration counter
    it+=1
    '''From here would come the code needed to calculate the successive iterations
    until the solution converges'''
```

#### 7.4.2. El método de Gauss-Seidel.

**Obtención del algoritmo.** Este método aplica una mejora, sencilla y lógica al método de Jacobi que acabamos de estudiar. Si volvemos a escribir la expresión general para calcular una iteración de una de las componentes de la solución de un sistema por el método de Jacobi,

We then construct the vector  $f f=(\text{np.linalg.inv}(A)).dot(b)$ , and the matrix of the system  $H H=-(\text{np.linalg.inv}(A)).dot(U+L)$

From the constructed matrices, the code to compute an iteration by Jacobi's method would be simply,  $\mathbf{x}_1=\mathbf{f}+\mathbf{H}.\text{dot}(\mathbf{x}_0)$ .

The following code fragment brings together all the operations described,

#### 7.3.2. The Gauss-Seidel method

**Obtaining the algorithm .** This method applies a simple and logical improvement to the Jacobi method we have just studied. If we rewrite the general expression for calculating an iteration of one of the components of the solution of a system by Jacobi's method,

$$x_i^{(s+1)} = \frac{b_i - \sum_{j \neq i} a_{ij} x_j^{(s)}}{a_{ii}}$$

Observamos que para obtener el término  $x_i^{(s+1)}$  empleamos todos los términos  $x_j^{(s)}$ ,  $j \neq i$  de la iteración anterior. Sin embargo, es fácil darse cuenta que, como el algoritmo va calculando las componentes de cada iteración por orden, cuando vamos a calcular  $x_i^{(s+1)}$ , disponemos ya del resultado de todas las componentes anteriores de la solución para la iteración  $s+1$ . Es decir, sabemos ya cuál es el resultado de  $x_l^{(s+1)}$ ,  $l < i$ . Si el algoritmo converge, estas soluciones serán mejores que las obtenidas en la iteración anterior. Si las usamos para calcular  $x_i^{(s+1)}$  el resultado que obtendremos, será más próximo a la solución exacta y por tanto el algoritmo converge más deprisa.

La idea, por tanto sería:

Para calcular  $x_1^{(s+1)}$  procedemos igual que en el método de Jacobi.

We observe that to obtain the term  $x_i^{(s+1)}$  we use all the terms  $x_j^{(s)}$ ,  $j \neq i$  of the previous iteration. However, it is easy to realise that, as the algorithm calculates the components of each iteration in order, when we calculate  $x_i^{(s+1)}$ , we already have the result of all the previous components of the solution for iteration  $s+1$ . That is, we already know the result of  $x_l^{(s+1)}$ ,  $l < i$ . If the algorithm converges, these solutions will be better than those obtained in the previous iteration. If we use them to calculate  $x_i^{(s+1)}$ , the result we obtain will be closer to the exact solution and therefore the algorithm converges faster.

The idea, therefore, would be:

To calculate  $x_1^{(s+1)}$  we proceed in the same way as in Jacobi's method.

Para calcular  $x_2^{(s+1)}$  usamos la solución que acabamos de obtener  $x_1^{(s+1)}$  y todas las restantes soluciones de las iteraciones anteriores,

$$x_2^{(s+1)} = \frac{b_2 - a_{21}x_1^{(s+1)} - \sum_{j=3}^n a_{2j}x_j^{(s)}}{a_{22}}$$

Para calcular  $x_3^{(s+1)}$  usamos las dos soluciones que acabamos de obtener  $x_1^{(s+1)}$ ,  $x_2^{(s+1)}$  y todas las restantes soluciones de las iteraciones anteriores,

$$x_3^{(s+1)} = \frac{b_3 - a_{31}x_1^{(s+1)} - a_{32}x_2^{(s+1)} - \sum_{j=4}^n a_{3j}x_j^{(s)}}{a_{33}}$$

Y para una componente  $i$  cualquiera de la solución, obtenemos la expresión general del método de Gauss-Seidel,

$$x_i^{(s+1)} = \frac{b_i - \sum_{j<i} a_{ij}x_j^{(s+1)} - \sum_{j>i} a_{ij}x_j^{(s)}}{a_{ii}}$$

El siguiente código implementa una iteración del método de Gauss-Seidel y puede considerarse como la función  $M(\cdot)$ , incluida en el diagrama de flujo de la figura 7.4, para dicho método.

Es interesante destacar, que el único cam-

To calculate  $x_2^{(s+1)}$  we use the solution we have just obtained  $x_1^{(s+1)}$  and all the remaining solutions from the previous iterations,

To calculate  $x_3^{(s+1)}$  we use the two solutions we have just obtained  $x_1^{(s+1)}$ ,  $x_2^{(s+1)}$  and all the remaining solutions from the previous iterations,

And for any  $i$  component of the solution, we obtain the general expression of the Gauss-Seidel method,

The following code implements an iteration of the Gauss-Seidel method and can be considered as the function  $M(\cdot)$ , included in the flowchart in figure 7.4, for the Gauss-Seidel method.

Interestingly, the only change in the entire

bio en todo el código respecto al método de Jacobi es la sustitución de la variable `xs(j)` por la variable `xs1(j)` al final del primer bucle for anidado.

```
# For all the equations
xs1=b.copy()
for i in range(nf):
    # For all the terms above x[i]
    for j in range(i):
        xs1[i]-=A[i,j]*xs1[j]
    #For all the terms below x[i]
    for j in range(i+1,nf):
        xs1[i]-=A[i,j]*xs[j]
    xs1[i]=xs1[i]/A[i,i]
error= np.linalg.norm(xs1-xs)
xs=xs1.copy()
it+=1
```

**Forma matricial del método de Gauss-Seidel.** De modo análogo a cómo hicimos para el método de Jacobi, es posible obtener una solución matricial para el método de Gauss-Seidel.

Supongamos que realizamos la misma descomposición en sumandos de la matriz de coeficientes que empleamos para el método de Jacobi,

$$A \cdot x = b \rightarrow (D + L + U) \cdot x = b$$

En este caso, en cálculo de cada componente de la solución en una iteración intervienen las componentes de la iteración anterior que están por debajo de la que queremos calcular, por tanto solo deberemos pasar a la derecha de la igualdad, la matriz  $U$ , Que contiene los coeficientes que multiplican a dichas componentes,

$$(D + L + U) \cdot x = b \rightarrow (D + L) \cdot x = b - U \cdot x$$

Sustituyendo  $x$  a cada lado de la igualdad por  $x^{(s+1)}$  y  $x^{(s)}$  y multiplicando por la inversa de  $D + U$  a ambos lados, obtenemos la expresión en forma matricial del cálculo de una iteración por el método de Gauss-Seidel,

$$x^{(s+1)} = (D + L)^{-1} \cdot b - (D + L)^{-1} \cdot U \cdot x^{(s)}$$

code from Jacobi's method is the replacement of the variable `xs(j)` with the variable `xs1(j)` at the end of the first nested for loop.

**Matrix form of the Gauss-Seidel method**. Analogous to how we did for Jacobi's method, it is possible to obtain a matrix solution for the Gauss-Seidel method.

Suppose we perform the same decomposition into summands of the coefficient matrix as we did for Jacobi's method,

In this case, the calculation of each component of the solution in an iteration involves the components of the previous iteration that are below the one we want to calculate, therefore we only have to pass to the right of the equality, the matrix  $U$ , which contains the coefficients that multiply these components,

Substituting  $x$  on each side of the equality for  $x^{(s+1)}$  and  $x^{(s)}$  and multiplying by the inverse of  $D + U$  on both sides, we obtain the expression in matrix form of the calculation of an iteration by the Gauss-Seidel method,

Igual que hemos hecho en con el método de Jacobi, podemos identificar las partes fijas que no cambian al iterar:  $f = (D + L)^{-1} \cdot b$  y la matriz del método, que en este caso es,  $H = -(D + L)^{-1} \cdot U$ .

El siguiente fragmento de código muestra la construcción de las matrices necesarias para implementar el método de Gauss-Seidel,

```
[nf ,nc]=np.shape(A)
xs=x0.copy()
xs1=b.copy()
error= np.linalg.norm(xs1-xs)
it=0
D=np.diag(np.diag(A))
U=np.triu(A)-D
L=np.tril(A)-D
invD=np.linalg.inv(D+L)
f=(invD).dot(b)
H=-invD.dot(U)
# First iteration
xs1=f+H.dot(xs)
error=np.linalg.norm(xs1-xs)
it+=1
'''From here would come the code needed to calculate the
successive iterations until the solution converges'''
```

En general, el método de Gauss-Seidel es capaz de obtener soluciones, para un sistema dado y un cierto valor de tolerancia, empleando menos iteraciones que el método de Jacobi. Por ejemplo, si aplicamos ambos métodos a la resolución del sistema,

$$\begin{pmatrix} 4 & 2 & -1 \\ 3 & -5 & 1 \\ 1 & -1 & 6 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 5 \\ -4 \\ 17 \end{pmatrix}$$

Empleando en ambos casos una tolerancia  $tol = 0.00001$ , el método de Jacobi necesita 23 iteraciones para lograr una solución, mientras que el de Gauss-Seidel solo necesita 13. La figura 7.5, muestra la evolución de la tolerancia  $tol = \|x^{(s+1)} - x^{(s)}\|$ , en función del número de iteración. En ambos casos el valor inicial de la solución se tomó como el vector  $(0, 0, 0)^T$ .

As we have done with Jacobi's method, we can identify the fixed parts that do not change when iterating:  $f = (D + L)^{-1} \cdot b$  and the method matrix, that in this case is,  $H = -(D + L)^{-1} \cdot U$ .

The following code fragment shows the construction of the matrices needed to implement the Gauss-Seidel method,

In general, the Gauss-Seidel method is able to obtain solutions, for a given system and a certain tolerance value, using fewer iterations than the Jacobi method. For example, if we apply both methods to the solution of the system,

Using in both cases a tolerance  $tol = 0.00001$ , the Jacobi method needs 23 iterations to achieve a solution, while the Gauss-Seidel method needs only 13. The figure 7.5, shows the evolution of the tolerance  $tol = \|x^{(s+1)} - x^{(s)}\|$ , as a function of the iteration number. In both cases the initial value of the solution was taken as the vector  $(0, 0, 0)^T$ .

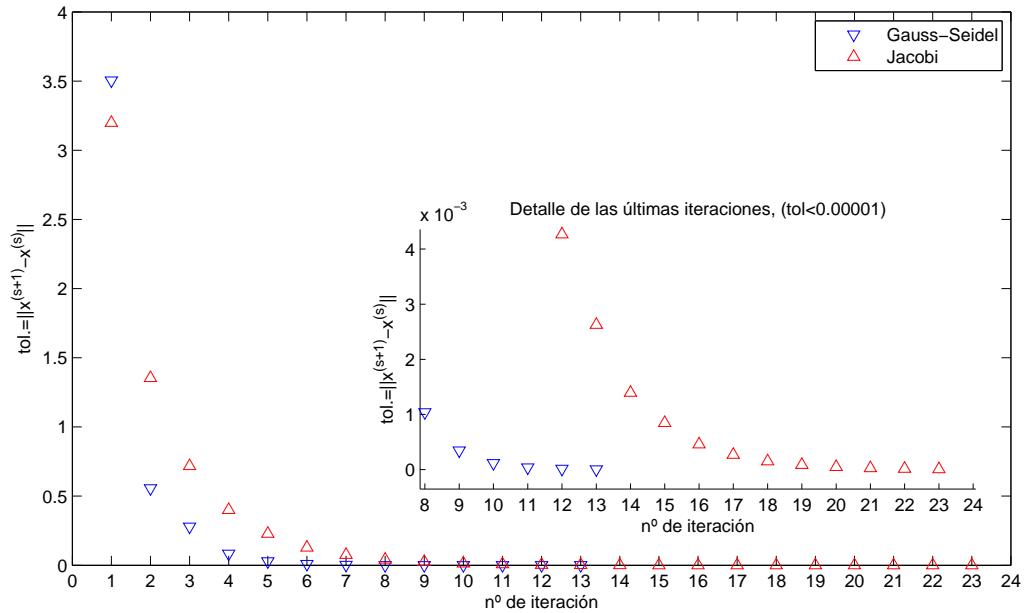


Figura 7.5: evolución de la tolerancia (módulo de la diferencia entre dos soluciones sucesivas) para un mismo sistema resuelto mediante el método de Gauss-Seidel y el método de Jacobi

Figure 7.5: evolution of the tolerance (modulus of the difference between two successive solutions) for the same system solved by the Gauss-Seidel method and the Jacobi method

#### 7.4.3. Amortiguamiento.

El amortiguamiento consiste en modificar un método iterativo, de modo que en cada iteración, se da como solución la media ponderada de los resultados de dos iteraciones sucesivas,

$$x^{(s+1)} = \omega \cdot x^* + (1 - \omega) \cdot x^{(s)}$$

Donde  $x^{(*)}$  representa el valor que se obtendría aplicando una iteración del método a  $x^{(s)}$ , es decir sería el valor de  $x^{(s+1)}$  si no se aplica amortiguamiento.

El parámetro  $\omega$  recibe el nombre de factor de relajamiento. Si  $0 < \omega < 1$  se trata de un método de subrelajación. Su uso permite resolver sistemas que no convergen si se usa el mismo método sin relajación. Si  $\omega > 1$  el método se llama de sobrerelajación, permite acelerar la convergencia respecto al mismo método sin relajación. Por último, si hacemos  $\omega = 1$ , recuperamos el método original sin re-

#### 7.3.3. Weighted methods.

In weighted iterations we modify an iterative method so that at each iteration, the weighted average of the results of two successive iterations is given as the solution,

$$x^{(s+1)} = \omega \cdot x^* + (1 - \omega) \cdot x^{(s)}$$

Where  $x^{(*)}$  represents the value that would be obtained by applying one iteration of the method to  $x^{(s)}$ , i.e. it would be the value of  $x^{(s+1)}$  if no weight were applied.

The parameter  $\omega$  is called the relaxation factor. If

$$0 < \omega < 1$$

this is a method of under-relaxation. Its use makes it possible to solve systems that do not converge if the same method is used without relaxation. If  $\omega > 1$  the method is called over-relaxation, it allows to accelerate the convergence with respect to the same method without

lajación.

**El método de Jacobi amortiguado.** Se obtiene aplicando el método de relajación que acabamos de describir, al método de Jacobi. La expresión general de una iteración del método de Jacobi amortiguando sería,

relaxation. Finally, if we make  $\omega = 1$ , we recover the original method without relaxation.

**The weighted Jacobi method.** This is obtained by applying the relaxation method just described to Jacobi's method. The general expression for an iteration of the damped Jacobi method would be,

$$x_i^{(s+1)} = \underbrace{\omega \cdot \frac{b_i - \sum_{j \neq i} a_{ij} x_j^{(s)}}{a_{ii}}}_{x^{(*)}} + (1 - \omega) \cdot x_i^s$$

Para implementarlo, bastaría añadir al código del método de Jacobi una línea incluyendo el promedio entre las dos soluciones sucesivas,

To implement it, it would be sufficient to add to the code of Jacobi's method a line including the average between the two successive solutions,

```
error= np.linalg.norm(xs1-xs)
it=0
while error>tol:
    # For all the equations
    xs1=b.copy()
    for i in range(nf):
        for j in range(i):
            xs1[i]-=A[i,j]*xs[j]

        for j in range(i+1,nf):
            xs1[i]-=A[i,j]*xs[j]
        xs1[i]=xs1[i]/A[i,i]
    # Weighted iteration
    xs1=w*xs1+(1-w)*xs
    error= np.linalg.norm(xs1-xs)
    xs=xs1.copy()
    it+=1
    if it>itmax:
        break
```

En forma matricial, la expresión general del método de Jacobi amortiguado sería,

In matrix form, the general expression for the damped Jacobi method would be,

$$x^{(s+1)} = \underbrace{\omega \cdot \left( D^{-1} \cdot b - D^{-1} \cdot (L + U) \cdot x^{(s)} \right)}_{x^{(*)}} + (1 - w) x^{(s)}$$

Si reorganizamos esta expresión,

Reordering the expression,

$$x^{(s+1)} = \omega \cdot D^{-1} \cdot b + [(1 - w) \cdot I - w \cdot D^{-1} \cdot (L + U)] \cdot x^{(s)}$$

Podemos identificar fácilmente el término fijo,  $f = \omega \cdot D^{-1} \cdot b$  y la matriz del método

We can easily identify the fixed term  $f = \omega \cdot D^{-1} \cdot b$  and the method matrix  $H = ((1 - w) \cdot I - w \cdot D^{-1} \cdot (L + U))$

$$H = ((1 - w) \cdot I - w \cdot D^{-1} \cdot (L + U)).$$

Para implementar el código del método de Jacobi amortiguado, debemos calcular la matriz identidad del tamaño de sistema y modificar las expresiones de  $f$  y  $H$ ,

```
[nf,nc]=np.shape(A)
xs=x0.copy()
xs1=b.copy()
error= np.linalg.norm(xs1-xs)
it=0
D=np.diag(np.diag(A))
U=np.triu(A)-D
L=np.tril(A)-D
invD=np.linalg.inv(D)
I=np.eye(nf)
f=w*(invD).dot(b)
H=(1-w)*I-w*invD.dot(L+U)
# First iteration
xs1=f+H.dot(xs)
error=np.linalg.norm(xs1-xs)
it+=1
'''From here would come the code needed to calculate the
successive iterations until the solution converges'''
```

**El método SOR.** El método SOR –*Successive OverRelaxation*– se obtiene aplicando amortiguamiento al método de Gauss-Seidel. Aplicando el mismo razonamiento que el caso de Jacobi amortiguado, la expresión general para una iteración del método SOR es,

$$x_i^{(s+1)} = \omega \cdot \overbrace{\frac{b_i - \sum_{j < i} a_{ij}x_j^{(s+1)} - \sum_{j > i} a_{ij}x_j^{(s)}}{a_{ii}}}^{x^{(*)}} + (1 - \omega) \cdot x_i^{(s)}$$

Al igual que en el caso de Jacobi amortiguado, para implementar el método SOR es suficiente añadir una línea que calcule el promedio de dos soluciones sucesivas,

```
# For all the equations
xs1=b.copy()
for i in range(nf):
    # For all the terms above x[i]
    for j in range(i):
        xs1[i]-=A[i,j]*xs1[j]
    #For all the terms below x[i]
```

To implement the code of the damped Jacobi method, we must calculate the identity matrix of the system size and modify the expressions of  $f$  and  $H$ ,

**The SOR method**. The SOR method –‘successive over-relaxation’ – is obtained by applying a weight to the Gauss-Seidel method. Applying the same reasoning as the weighted Jacobi case, the general expression for an iteration of the SOR method is,

As in the case of the damped Jacobi, to implement the SOR method it is sufficient to add a line that calculates the average of two successive solutions,

```

for j in range(i+1,nf):
    xs1[i]-=A[i,j]*xs[j]
    xs1[i]=xs1[i]/A[i,i]
xs1=w*xs1+(1-w)*xs
error= np.linalg.norm(xs1-xs)
xs=xs1.copy()
it+=1

```

En forma matricial la expresión para una iteración del método SOR sería,

$$x^{(s+1)} = \underbrace{\omega \cdot ((D + L)^{-1} \cdot b - (D + L)^{-1} \cdot U \cdot x^{(s)})}_{x^{(*)}} + (1 - \omega) \cdot x^{(s)}$$

Y tras reordenar,

$$x^{(s+1)} = \omega \cdot ((D + L)^{-1} \cdot b) + [(1 - \omega) \cdot I - \omega \cdot (D + L)^{-1} \cdot U] \cdot x^{(s)}$$

De nuevo, podemos identificar, el término fijo,  $f = \omega \cdot ((D + L)^{-1} \cdot b)$  y la matriz del método,  $H = ((1 - \omega) \cdot I - \omega \cdot (D + L)^{-1} \cdot U)$ .

El siguiente fragmento de código muestra la obtención de  $f$  y  $H$  para el método SOR,

```

[nf,nc]=np.shape(A)
xs=x0.copy()
xs1=b.copy()
error= np.linalg.norm(xs1-xs)
it=0
D=np.diag(np.diag(A))
U=np.triu(A)-D
L=np.tril(A)-D
invD=np.linalg.inv(D+L)
I=np.eye(nf)
f=w*(invD).dot(b)
H=(1-w)*I-w*invD.dot(U)
#First iteration
xs1=f+H.dot(xs)
error=np.linalg.norm(xs1-xs)
it+=1
xs=xs1.copy()

```

In matrix form the expression for one iteration of the SOR method would be,

And after reordering,

Again, we can identify the fixed term,  $f = \omega \cdot ((D + L)^{-1} \cdot b)$  and the method matrix,  $H = ((1 - \omega) \cdot I - \omega \cdot (D + L)^{-1} \cdot U)$ .

The following code fragment shows how  $f$  and  $H$  are obtained for the SOR method,

#### 7.4.4. Análisis de convergencia

En la introducción a los métodos iterativos dejamos abierta la cuestión de su convergencia. Vamos a analizar en más detalle en qué

#### 7.3.4. Convergence analysis

In the introduction to iterative methods we left open the question of their convergence. Let us analyse in more detail under what con-

condiciones podemos asegurar que un método iterativo, aplicado a un sistema de ecuaciones lineales concreto, converge.

En primer lugar, tenemos que definir qué entendemos por convergencia. Cuando un método iterativo converge, lo hace en forma asintótica. Es decir, haría falta un número infinito de iteraciones para alcanzar la solución exacta.

$$x^{(0)} \rightarrow x^{(1)} \rightarrow x^{(2)} \rightarrow \cdots \rightarrow x^{(\infty)} = x$$

Lógicamente, es inviable realizar un número infinito de iteraciones. Por esta razón, las soluciones de los métodos iterativos son siempre aproximadas; realizamos un número finito de iteraciones hasta cumplir una determinada condición de convergencia. Como no conocemos la solución exacta, imponemos dicha condición entre dos iteraciones sucesivas,

$$\|x^{(s+1)} - x^{(s)}\| \leq C \Rightarrow \|x^{(s+1)} - x\| = |e^{(s+1)}|$$

Donde  $e^{(s+1)}$  representaría el error *real* de convergencia cometido en la iteración  $s + 1$ .

Tomando como punto de partida la expresión general del cálculo de una iteración en forma matricial,

$$x^{(s+1)} = f + H \cdot x^{(s)}$$

Podemos expresar el error de convergencia como,

$$e^{(s+1)} = x^{(s+1)} - x = f + H \cdot x^{(s)} - x$$

Pero la solución exacta, si pudiera alcanzarse, cumpliría,

$$x = f + H \cdot x$$

Y si sustituimos en la expresión del error de convergencia,

$$e^{(s+1)} = f + H \cdot x^{(s)} - f - H \cdot x$$

Llegamos finalmente a la siguiente expresión, que relaciona los errores de convergencia de dos iteraciones sucesivas,

$$e^{(s+1)} = H \cdot e^{(s)}$$

ditions we can ensure that an iterative method, applied to a particular system of linear equations, converges.

First of all, we have to define what we mean by convergence. When an iterative method converges, it does so asymptotically. That is, it would take an infinite number of iterations to reach the exact solution.

$$x^{(0)} \rightarrow x^{(1)} \rightarrow x^{(2)} \rightarrow \cdots \rightarrow x^{(\infty)} = x$$

Logically, it is unfeasible to perform an infinite number of iterations. For this reason, the solutions of iterative methods are always approximate; we perform a finite number of iterations until a certain convergence condition is met. Since we do not know the exact solution, we impose this condition between two successive iterations,

$$\|x^{(s+1)} - x^{(s)}\| \leq C \Rightarrow \|x^{(s+1)} - x\| = |e^{(s+1)}|$$

Where  $e^{(s+1)}$  would represent the real convergence error committed in iteration  $s + 1$ .

Taking as a starting point the general expression for the calculation of an iteration in matrix form,

$$x^{(s+1)} = f + H \cdot x^{(s)}$$

We can express the convergence error as,

$$e^{(s+1)} = x^{(s+1)} - x = f + H \cdot x^{(s)} - x$$

But the exact solution, if it could be achieved, would comply,

$$x = f + H \cdot x$$

And if we substitute in the expression for the convergence error,

$$e^{(s+1)} = f + H \cdot x^{(s)} - f - H \cdot x$$

We finally arrive to the following expression, that relates the convergence errors between two successive iterations,

$$e^{(s+1)} = H \cdot e^{(s)}$$

Para que el error disminuya de iteración en iteración y el método converja, es necesario que la matriz del método  $H$  tenga norma-2 menor que la unidad.

Supongamos que un sistema de dimensión  $n$  su matriz del método  $H$ , tiene un conjunto de  $n$  autovectores linealmente independientes,  $w_1, w_2, \dots, w_n$ , cada uno asociado a su correspondiente autovalor,  $\lambda_1, \lambda_2, \dots, \lambda_n$ . El error de convergencia, es también un vector de dimensión  $n$ , por tanto podemos expresarlo como una combinación lineal de los  $n$  autovectores linealmente independientes de la matriz  $H$ . Supongamos que lo hacemos para el error de convergencia  $e^{(0)}$  correspondiente al valor inicial de la solución  $x^{(0)}$ ,

$$e^{(0)} = \alpha_1 \cdot w_1 + \alpha_2 \cdot w_2 + \dots + \alpha_n \cdot w_n$$

Si empleamos la ecuación deducida antes para la relación del error entre dos iteraciones sucesivas y recordando que aplicar una matriz a un autovector, es equivalente a multiplicarlo por el autovalor correspondientes:  $H \cdot w_i = \lambda_i \cdot w_i$ , obtenemos para el error de convergencia en la iteración  $s$ ,

For the error to decrease from iteration to iteration and for the method to converge, the method matrix  $H$  must have norm-2 less than unity.

Suppose that a system of dimension  $n$  its method matrix  $H$ , has a set of  $n$  linearly independent eigenvectors,  $w_1, w_2, \dots, w_n$ , each one associated to its corresponding eigenvalue,  $\lambda_1, \lambda_2, \dots, \lambda_n$ . The convergence error is also a vector of dimension  $n$ , so we can express it as a linear combination of the  $n$  linearly independent eigenvectors of the matrix  $H$ . Suppose we do this for the convergence error  $e^{(0)}$  corresponding to the initial value of the solution  $x^{(0)}$ ,

If we use the equation deduced earlier for the error ratio between two successive iterations and remembering that applying a matrix to an eigenvector is equivalent to multiplying it by the corresponding eigenvalue:  $H \cdot w_i = \lambda_i \cdot w_i$ , we obtain for the error of convergence at iteration  $s$ ,

$$\begin{aligned} e^{(1)} &= H \cdot e^{(0)} = \alpha_1 \cdot \lambda_1 \cdot w_1 + \alpha_2 \cdot \lambda_2 \cdot w_2 + \dots + \alpha_n \cdot \lambda_n \cdot w_n \\ e^{(2)} &= H \cdot e^{(1)} = H^2 \cdot e^{(0)} = \alpha_1 \cdot \lambda_1^2 \cdot w_1 + \alpha_2 \cdot \lambda_2^2 \cdot w_2 + \dots + \alpha_n \cdot \lambda_n^2 \cdot w_n \\ &\vdots \\ e^{(s)} &= H \cdot e^{(s-1)} = H^s \cdot e^{(0)} = \alpha_1 \cdot \lambda_1^s \cdot w_1 + \alpha_2 \cdot \lambda_2^s \cdot w_2 + \dots + \alpha_n \cdot \lambda_n^s \cdot w_n \end{aligned}$$

Para que el error tienda a cero,  $e^{(s)} \rightarrow 0$  al aumentar  $s$ , para cualquier combinación inicial de valores  $\alpha_i$ , esto es para cualquier aproximación inicial  $x^{(0)}$ , es necesario que todos los autovalores de la matriz del método cumplan,

$$|\lambda_i| < 1$$

Por tanto, el sistema converge si el radio espectral de la matriz del método es menor que la unidad.<sup>3</sup>

$$\rho(H) < 1 \Rightarrow \lim_{s \rightarrow \infty} e^{(s)} = 0$$

For the error to tend to zero,  $e^{(s)} \rightarrow 0$  as  $s$  increases, for any initial combination of values  $\alpha_i$ , i.e. for any initial approximation  $x^{(0)}$ , it is necessary that all the eigenvalues of the method matrix satisfy,

$$|\lambda_i| < 1$$

Therefore, the system converges if the spectral radius of the method matrix is less than unity.<sup>3</sup>

$$\rho(H) < 1 \Rightarrow \lim_{s \rightarrow \infty} e^{(s)} = 0$$

<sup>3</sup>El radio espectral de una matriz es el mayor de sus autovalores en valor absoluto.

<sup>3</sup>The spectral radius of a matrix is the largest of its eigenvalues in absolute value.

**Velocidad de convergencia.** Para un número de iteraciones suficientemente grande, el radio espectral de la matriz del método nos da la velocidad de convergencia. Esto es debido a que el resto de los términos del error, asociados a otros autovalores más pequeños tienden a cero más deprisa. Por tanto podemos hacer la siguiente aproximación, donde estamos suponiendo que el autovalor  $\lambda_n$  es el radio espectral,

$$e^{(s)} \approx c_n \rho(H)^s w_n = c_n \lambda_n^s w_n$$

Podemos ahora calcular cuantas iteraciones nos costará reducir un error inicial por debajo de un determinado valor. Esto dependerá de la solución inicial y de radio espectral de la matriz del método,

$$e^{(s)} \approx \rho(H)^s e^{(0)} \Rightarrow \frac{e^{(s)}}{e^{(0)}} \approx \rho(H)^s$$

así por ejemplo si queremos reducir el error inicial en  $m$  dígitos,

$$\frac{e^{(s)}}{e^{(0)}} \approx \rho(H)^s \leq 10^{-m} \Rightarrow s \geq \frac{-m}{\log_{10}(\rho(H))}$$

La matriz del método juega por tanto un papel fundamental tanto en la convergencia del método, como en la velocidad (número de iteraciones) con la que el método converge. Los métodos amortiguados, permiten modificar la matriz de convergencia, gracias al factor de amortiguamiento  $\omega$ , haciendo que sistemas para los que no es posible encontrar una solución mediante un método iterativo converjan.

Como ejemplo, el sistema,

$$\begin{pmatrix} 1 & 2 & -1 \\ 2 & -5 & 1 \\ 1 & -1 & 3 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ -5 \\ 8 \end{pmatrix}$$

No converge si tratamos de resolverlo por el método de Jacobi. Sin embargo si es posible obtener su solución empleando el método de Jacobi Amortiguado. La figura 7.6 muestra la evolución de la tolerancia para dicho sistema empleando ambos métodos.

Si calculamos el radio espectral de la matriz del método, para el método de Jacobi tendríamos,

**Speed of convergence**. For a sufficiently large number of iterations, the spectral radius of the method matrix gives the speed of convergence. This is because the remaining error terms associated with other smaller eigenvalues tend to zero faster. We can therefore make the following approximation, where we are assuming that the eigenvalue  $\lambda_n$  is the spectral radius,

We can now calculate how many iterations it will take to reduce an initial error below a certain value. This will depend on the initial solution and the spectral radius of the method matrix,

so for example if we want to reduce the initial error by  $m$  digits,

The method matrix therefore plays a fundamental role both in the convergence of the method and in the speed (number of iterations) with which the method converges. Damped methods allow the convergence matrix to be modified, thanks to the damping factor *omega*, making systems for which it is not possible to find a solution by means of an iterative method converge.

As an example, the system,

It does not converge if we try to solve it by the Jacobi method. However, it is possible to obtain its solution using the Weighted Jacobi method. Figure 7.6 shows the evolution of the tolerance for this system using both methods.

If we calculate the spectral radius of the method matrix, for the Jacobi method we would have,

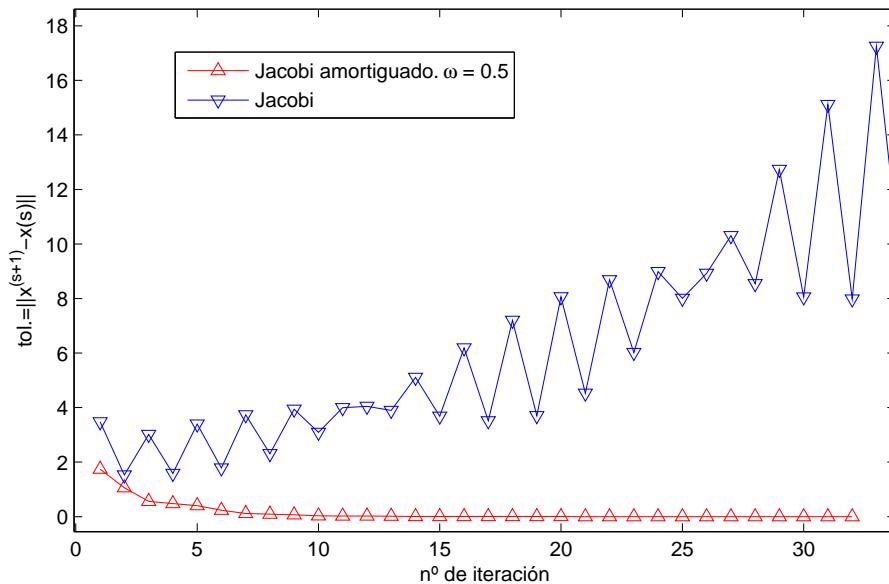


Figura 7.6: Evolución de la tolerancia para un mismo sistema empleando el método de Jacobi (diverge) y el de Jacobi amortiguado (converge).

Figure 7.6: Tolerance evolution for the same system using the Jacobi method (diverges) and the weighted Jacobi method (converges).

```
import numpy as np
A=np.array([[1,2,-1],[2,-5,1],[1,-1,3]])
D=np.diag(np.diag(A))
U=np.triu(A)-D
L=np.tril(A)-D
invD=np.linalg.inv(D)
H=invD.dot(L+U)
eigenvalues,eigenvectors=np.linalg.eig(H)
spectral_rad=np.max(np.abs(eigenvalues))
print("Eigenvalues: ",eigenvalues)
print("Spectral radius: ", spectral_rad)
```

```
Eigenvalues: [ 0.11872445+1.05306845j  0.11872445-1.05306845j -0.2374489 +0.j]
Spectral radius:  1.0597398959658624 ]
```

El radio espectral es mayor que la unidad y el método no converge.

Si repetimos el cálculo para el método de Jacobi amortiguado, con  $\omega = 0.5$

```
D=np.diag(np.diag(A))
U=np.triu(A)-D
L=np.tril(A)-D
invD=np.linalg.inv(D)
```

The spectral radius is greater than unity and the method does not converge.

If we repeat the calculation for the weighted Jacobi method, with  $\omega = 0.5$ ,

```
w=0.5
I=np.eye(3)
H=(1-w)*I-w*invD.dot(L+U)
eigenvalues,eigenvectors=np.linalg.eig(H)
spectral_rad=np.max(np.abs(eigenvalues))
print("Eigenvalues: ", eigenvalues)
print("Spectral radius: ", spectral_rad)
```

Eigenvalues: [0.44063777+0.52653422j 0.44063777-0.52653422j 0.61872445+0.j] ]  
 Spectral radius: 0.6865857095839855

El radio espectral es ahora menor que la unidad y el método converge.

Por último indicar que cualquiera de los métodos iterativos descrito converge para un sistema que cumpla que su matriz de coeficientes es estrictamente diagonal dominante.

The spectral radius is now smaller than unity and the method converges.

Finally, it should be noted that any of the iterative methods described converges for a system whose coefficient matrix is strictly diagonally dominant.

## 7.5. Ejercicios

1. Crea una función en python que calcule la solución de un sistema de ecuaciones triangular inferior empleando el método de sustituciones progresivas. La función deberá tomar como valores de entrada una matriz triangular inferior de dimensión ( $n \times n$ ) arbitraria y un vector columna ( $n \times 1$ ) de términos independientes. Deberá devolver como variable de salida un vector columna ( $n \times 1$ ) con las soluciones del sistema.
2. Crea una función en python que calcule la solución de un sistema de ecuaciones triangular superior empleando el método de sustituciones regresivas. La función deberá tomar como valores de entrada una matriz triangular inferior de dimensión ( $n \times n$ ) arbitraria y un vector columna ( $n \times 1$ ) de términos independientes. Deberá devolver un vector columna ( $n \times 1$ ) con las soluciones del sistema.
3. Crea una función que dadas una matriz cuadrada  $A$ , ( $n \times n$ ) y un vector columna  $b$ , ( $n \times 1$ ), construya la matriz ampliada  $Ab$ . Aplique el método de eliminación gausiana a la matriz ampliada y llame a la función creada en el ejercicio 2 Para resolver el sistema  $Ax = b$ . La función deberá devolver como variables de salida, La matriz resultante de la aplicar la eliminación gaussiana a la ampliada y un vector columna con las soluciones del sistema resuelto.
4. Crea una función que dadas una matriz cuadrada  $A$ , ( $n \times n$ ) y un vector columna  $b$ , ( $n \times 1$ ), construya la matriz ampliada  $Ab$ . Aplique el método de eliminación Gauss-Jordan a la matriz ampliada y devuelva como variable de salida la matriz en forma escalonada reducida por filas, de modo que la última fila sea la solución del sistema  $Ax = b$
5. Construye un programa que resuelva un sistema de ecuaciones de dimensión arbitraria, empleando el método de Jacobi simple (no en forma matricial). El programa deberá admitir como variables de entrada, una matriz de coeficientes  $A$ , ( $n \times n$ ), un vector de términos independientes  $b$ , ( $n \times 1$ ), una solución inicial  $x_0$ , ( $n \times 1$ ), un valor para la tolerancia máxima entre dos iteraciones sucesivas y un número máximo de iteraciones permitido. El programa

deberá devolver un vector columna con las soluciones del sistema, el número de iteraciones empleado y el error relativo entre las dos últimas iteraciones realizadas.

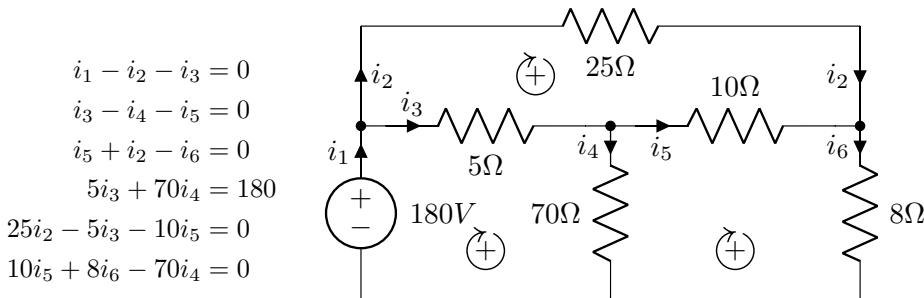
6. Repite el ejercicio anterior empleando ahora el método de Jacobi matricial. Añade el código necesario para que calcule en primer lugar el radio espectral de la matriz del método y caso de no cumplirse la condición de convergencia, el programa interrumpa su ejecución y devuelva un mensaje de error indicando el valor del radio espectral.
7. Construye un programa que resuelva un sistema de ecuaciones de dimensión arbitraria, empleando el método de Gauss-Seidel simple (no en forma matricial). El programa deberá admitir como variables de entrada, una matriz de coeficientes  $A$ , ( $n \times n$ ), un vector de términos independientes  $b$ , ( $n \times 1$ ), una solución inicial  $x_0$ , ( $n \times 1$ ), un valor para la tolerancia máxima entre dos iteraciones sucesivas y un número máximo de iteraciones permitido. El programa deberá devolver un vector columna con las soluciones del sistema, el número de iteraciones empleado y el error relativo entre las dos últimas iteraciones realizadas.
8. Repite el ejercicio anterior empleando ahora el método de Gauss-Seidel matricial. Añade el código necesario para que calcule en primer lugar el radio espectral de la matriz del método y caso de no cumplirse la condición de convergencia, el programa interrumpa su ejecución y devuelva un mensaje de error indicando el valor del radio espectral.
9. Construye un programa que resuelva un sistema de ecuaciones de dimensión arbitraria, empleando el método de Jacobi amortiguado simple (no en forma matricial). El programa deberá admitir como variables de entrada, una matriz de coeficientes  $A$ , ( $n \times n$ ), un vector de términos independientes  $b$ , ( $n \times 1$ ), una solución inicial  $x_0$ , ( $n \times 1$ ), un valor para el parámetro de amortiguamiento  $\omega$ , un valor para la tolerancia máxima entre dos iteraciones sucesivas y un número máximo de iteraciones permitido. El programa deberá devolver un vector columna con las soluciones del sistema, el número de iteraciones empleado y el error relativo entre las dos últimas iteraciones realizadas.
10. Repite el ejercicio anterior empleando ahora el método de Jacobi amortiguado matricial. Añade el código necesario para que calcule en primer lugar el radio espectral de la matriz del método y caso de no cumplirse la condición de convergencia, el programa interrumpa su ejecución y devuelva un mensaje de error indicando el valor del radio espectral.
11. Construye un programa que resuelva un sistema de ecuaciones de dimensión arbitraria, empleando el método SOR simple (no en forma matricial). El programa deberá admitir como variables de entrada, una matriz de coeficientes  $A$ , ( $n \times n$ ), un vector de términos independientes  $b$ , ( $n \times 1$ ), una solución inicial  $x_0$ , ( $n \times 1$ ), un valor para el parámetro de amortiguamiento  $\omega$ , un valor para la tolerancia máxima entre dos iteraciones sucesivas y un número máximo de iteraciones permitido. El programa deberá devolver un vector columna con las soluciones del sistema, el número de iteraciones empleado y el error relativo entre las dos últimas iteraciones realizadas.
12. Repite el ejercicio anterior empleando ahora el método SOR matricial. Añade el código necesario para que calcule en primer lugar el radio espectral de la matriz del método y caso de no cumplirse la condición de convergencia, el programa interrumpa su ejecución y devuelva un mensaje de error indicando el valor del radio espectral.
13. Resuelve el siguiente sistema de ecuaciones, empleando la factorización LU. Indica todos los

pasos empleados hasta obtener las solución del sistema.

$$\begin{aligned} 3x_1 + 3x_2 - 2x_3 + x_4 &= 13 \\ + 2x_2 - x_3 &= 3 \\ -2x_1 + x_2 + 5x_3 - 4x_4 &= 1 \\ 2x_1 - x_3 + 2x_4 &= 5 \end{aligned}$$

Repite el ejercicio empleando las factorizaciones QR y SVD.

14. Para calcular las intensidades de un circuito de corriente continua como el de la figura, es suficiente emplear las leyes de Kirchoff. La primera ley –*ley de nodos*– establece que la suma de corrientes que llegan a un nodo del circuito debe ser igual a cero. La segunda –*ley de mallas*– establece que la suma de las caídas de tensión en un malla cerrada tiene que ser cero. La caída de voltaje en una resistencia se calcula empleando la ley de Ohm:  $V = i \cdot R$ . Si aplicamos las leyes de Kirchoff al circuito de la figura, obtenemos el siguiente sistema de ecuaciones,



Las tres primeras ecuaciones corresponden a aplicar la *ley de nodos* a los nodos marcados con un punto negro en la figura.

Las tres últimas, a aplicar la *ley de mallas* a las tres mallas considerando positivo recorrerlas en el sentido de las agujas del reloj.

El sistema de ecuaciones obtenido puede expresarse en forma matricial como,

$$\begin{pmatrix} 1 & -1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & -1 & 0 \\ 0 & 1 & 0 & 0 & 1 & -1 \\ 0 & 0 & 5 & 70 & 0 & 0 \\ 0 & 25 & -5 & 0 & -10 & 0 \\ 0 & 0 & 0 & -70 & 10 & 8 \end{pmatrix} \begin{pmatrix} i_1 \\ i_2 \\ i_3 \\ i_4 \\ i_5 \\ i_6 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 180 \\ 0 \\ 0 \end{pmatrix} \quad (7.4)$$

- Define en python el sistema de ecuaciones (7.4). Obtén los valores de las intensidades en todas las ramas del circuito empleando directamente un solo comando o función de python. Calcula el condicionamiento del sistema y dí si en tu opinión el sistema está bien o mal condicionado.
- Emplea la factorización QR para obtener de nuevo la solución del sistema.
- Utilizando la versión permutada del sistema, obtenida mediante la factorización LU, es decir, usando  $P \cdot A$  como matriz del sistema y  $P \cdot b$ , como término independiente, calcula el radiopectral correspondiente al método de Jacobi e indica si tiene sentido o no emplear este método para obtener las intensidades del circuito de la figura.

- d) Emplea el método de Jacobi amortiguado, con un amortiguamiento de  $\omega = 0.1$  para calcular las intensidades del circuito de la figura, con una tolerancia de  $10^{-5}$ . Indica el número de iteraciones empleado hasta alcanzar la solución. (*Nota importante:* Usa la versión permutada del sistema y ten en cuenta que va a necesitar bastantes iteraciones –más de 3000– para converger).
- e) Emplea el método SOR, con un amortiguamiento de  $\omega = 0.2$  para calcular las intensidades del circuito de la figura, con una tolerancia de  $10^{-5}$ . Indica el número de iteraciones empleado hasta alcanzar la solución (*Nota importante:* Usa la versión permutada del sistema). Discute, a la vista de los resultados, cuál método funciona mejor.
- f) Escribe un código que permita dibujar el radiopectral de un método amortiguado en función de la matriz del método, para distintos valores del amortiguamiento. Emplea el programa para los casos de las dos preguntas anteriores. ¿Ha sido razonable la elección de los valores de  $\omega$  realizada en dichas preguntas?

## 7.6. Test del curso 2020/21

**Problema 1.** El método iterativo de Richardson para la resolución de sistemas de ecuaciones lineales emplea la siguiente fórmula de recurrencia

$$x^{(s+1)} = x^{(s)} + \omega (b - Ax^{(s)}), \quad (7.5)$$

donde  $A \in \mathbb{R}^{n \times n}$  y  $b \in \mathbb{R}^n$  representan respectivamente la matriz de coeficientes y el vector columna de términos independientes del sistema de ecuaciones  $Ax = b$  de orden  $n \in \mathbb{N}$ , y  $x^{(s)} \in \mathbb{R}^n$  es el vector solución en la iteración  $s \in \mathbb{N}$ . El parámetro  $\omega \in \mathbb{R}$  juega el mismo papel que el factor de relajación en los métodos amortiguados y debe ajustarse para asegurar la convergencia del método.

1. **1 punto.** Reescribe la fórmula de recurrencia del método Richardson (ecuación 8.2) en la forma matricial estándar

$$x^{(s+1)} = f + Hx^{(s)}. \quad (7.6)$$

2. **2 puntos.** Construye una función en python que implemente el método de Richardson empleando la forma matricial estándar, es decir, en cada iteración aplicamos la ecuación (8.3). La función deberá tomar como variables de entrada:

- a) La matriz de coeficientes  $A$ .
- b) El vector de términos independientes  $b$ .
- c) El valor inicial  $x^{(0)}$ .
- d) Número máximo de iteraciones a realizar.
- e) Tolerancia mínima para el error entre iteraciones sucesivas.
- f) Un valor para el parámetro  $\omega$ .

Así mismo, la función deberá devolver como variables de salida:

- a) La solución del sistema.
- b) La tolerancia alcanzada.
- c) El número de iteraciones empleado en obtener la solución.

- 3. 2 puntos.** Emplea los métodos de Richardson, Jacobi y Gauss-Seidel para obtener la solución del sistema de ecuaciones

$$\begin{pmatrix} 5 & 2 & 3 & -1 \\ 2 & 6 & 3 & 0 \\ 1 & -4 & 4 & -1 \\ 2 & 0 & 3 & 7 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 14 \\ 23 \\ 1 \\ 39 \end{pmatrix}.$$

Emplea un valor  $\omega = 0.16$  para el método de Richardson y una tolerancia de  $10^{-5}$  para los tres métodos. Escoge el valor  $x^{(0)} = [0, 0, 0, 0]^T$  para los tres métodos.

Clasifica los tres métodos de mejor a peor, tomando como criterio el número de iteraciones empleado por cada uno de ellos para alcanzar la solución.

- 4. 2 punto.** Haz una gráfica del radio espectral de  $H$  en función del parámetro  $\omega$  para el método de Richardson y el sistema de ecuaciones del apartado anterior. Emplea para ello valores de  $\omega$  comprendidos en el intervalo  $[0.01, 0.9]$ , toma una separación entre valores de 0.01. Representa cada valor como un punto independiente.

Determina, a la vista de la gráfica, si sería posible encontrar un valor de  $\omega$  para el que se alcance la solución del sistema en menos iteraciones, manteniendo la misma tolerancia. Indica, también de acuerdo con el gráfico, cuál es mayor valor de  $\omega$  admisible. Razona las respuestas.

**Problema 2.** El método de Gauss-Jordan permite resolver sistemas de ecuaciones lineales de la forma  $Ax = b$ ,  $A \in \mathbb{R}^{n \times n}$  y  $x, b \in \mathbb{R}^n$ . Si en lugar de emplear un vector columna de términos independientes, sustituimos  $b$  por una matriz  $B$  de dimensión  $n \times m$ , el método nos permite obtener como resultado una matriz de soluciones  $X$  también de dimensión  $n \times m$ :  $AX = B$ . Cada columna,  $x_j$ ,  $j \in \{1, \dots, m\}$ , de la matriz  $X$  representa la solución de sistema  $Ax_j = b_j$ , donde  $b_j$  es la columna correspondiente de la matriz  $B$ . Es decir: hemos resuelto  $m$  sistemas de ecuaciones simultáneamente.

- 1. 2 puntos.** Emplea el método de Gauss-Jordan para obtener la solución de  $AX = B$ , donde  $A$  es la matriz de coeficientes del Problema 1 y  $B$  es la matriz identidad de dimensión  $4 \times 4$ .
- 2. 1 punto.** ¿Sabrías decir qué relación hay entre las matrices  $X$  y  $A$ ?

## Capítulo/Chapter 8

# Interpolación y ajuste de funciones

Digo que ya tú sabes que la humildad es la baza y fundamento de todas las virtudes, y que sin ella no hay alguna que lo sea. Ella allana inconvenientes, vence dificultades, y es un medio que siempre a gloriosos fines nos conduce; de los enemigos hace amigos, templa la cólera de los airados y menoscaba la arrogancia de los soberbios; es madre de la modestia y hermana de la templanza; en fin, con ella no pueden atravesar triunfo que les sea de provecho los vicios, porque en su blandura y mansedumbre se embotan y despuntan las flechas de los pecados.

---

El coloquio de los perros. Miguel de Cervantes

En este capítulo vamos a estudiar distintos métodos de aproximación polinómica. En términos generales el problema consiste en sustituir una función  $f(x)$  por un polinomio,

$$f(x) \approx p(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + a_3 \cdot x^3 + \cdots + a_n \cdot x^n$$

Para obtener la aproximación podemos partir de la ecuación que define  $f(x)$ , por ejemplo la función error,

In this chapter, we will explore different methods of polynomial approximation. We can define the problem in general terms as finding a polynomial to represent a function  $f(x)$ .

To obtain the approximation, we can start from the equation that defines  $f(x)$ , for instance, the error function,

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

O bien, puede suceder que solo conoczamos algunos valores de la función, por ejemplo a

Nevertheless, it is possible that we only have some values of the function, such as when

través de una tabla de datos,

we only have a data table.

Tabla 8.1:  $f(x) = \text{erf}(x)$

$x$	$f(x)$
0.0	0.0000
0.1	0.1125
0.2	0.2227
0.3	0.3286
0.4	0.4284
0.5	0.5205

La aproximación de una función por un polinomio, tiene ventajas e inconvenientes.

Probablemente la principal ventaja, desde el punto de vista del cómputo, es que un polinomio es fácil de evaluar mediante un ordenador ya que solo involucra operaciones aritméticas sencillas. Además, los polinomios son fáciles de derivar e integrar, dando lugar a otros polinomios.

En cuanto a los inconvenientes hay que citar el crecimiento hacia infinito o menos infinito de cualquier polinomio para valores de la variable independiente alejados del origen. Esto puede dar lugar en algunos casos a errores de redondeo difíciles de manejar, haciendo muy difícil la aproximación para funciones no crecientes.

Vamos a estudiar tres métodos distintos; en primer lugar veremos la aproximación mediante el polinomio de Taylor, útil para aproximar una función en las inmediaciones de un punto. A continuación, veremos la interpolación polinómica y, por último, estudiaremos el ajuste polinómico por mínimos cuadrados.

El uso de uno u otro de estos métodos está asociado a la información disponible sobre la función que se desea aproximar y al uso que se pretenda hacer de la aproximación realizada.

## 8.1. El polinomio de Taylor.

Supongamos una función infinitamente derivable en un entorno de un punto  $x_0$ . Su expansión en serie de Taylor se define como,

$$f(x) = f(x_0) + f'(x_0) \cdot (x - x_0) + \frac{1}{2} f''(x_0) \cdot (x - x_0)^2 + \cdots + \frac{1}{n!} f^{(n)}(x_0) \cdot (x - x_0)^n + \frac{1}{(n+1)!} f^{(n+1)}(z) \cdot (x - x_0)^{n+1}$$

To approximate a function using a polynomial has pros and cons.

Perhaps the main advantage, from a computing point of view, is that a polynomial is easy to evaluate using a computer as it only involves simple arithmetical operations. Besides, Polynomial integration and derivation are operations that are easy to carry out, yielding other polynomials.

Among their drawbacks we have to remark the polynomial growing towards infinity or minus infinity as the independent variable goes away zero. This can give rise in some cases to rounding errors that are difficult to handle, making the approximation for non-increasing functions very difficult.

We will study three different methods: first we will see the approximation using the Taylor polynomial, very useful to approximate a function close to a point for which we know the function value. Later, we will discuss polynomial interpolation and finally we present mean square polynomial fitting.

We may relate the use of one or another method to the available information on the function we want to approximate and to the objectives we want to reach with the approximation.

## 8.1. The Taylor's polynomial.

Suppose we have an function infinitely derivable around a point  $x_0$ . We define the function taylor series expansion as,

Donde  $z$  es un punto sin determinar situado entre  $x$  y  $x_0$ . Si eliminamos el último término, la función puede aproximarse por un polinomio de grado  $n$

$$f(x) \approx f(x_0) + f'(x_0) \cdot (x - x_0) + \frac{1}{2} f''(x_0) \cdot (x - x_0)^2 + \cdots + \frac{1}{n!} f^{(n)}(x_0) \cdot (x - x_0)^n$$

El error cometido al aproximar una función por un polinomio de Taylor de grado  $n$ , viene dado por el término,

$$e(x) = |f(x) - p(x)| = \left| \frac{1}{(n+1)!} f^{(n+1)}(z) \cdot (x - x_0)^{n+1} \right|$$

Es fácil deducir de la ecuación que el error disminuye con el grado del polinomio empleando y aumenta con la distancia entre  $x$  y  $x_0$ . Además cuanto más suave es la función (derivadas pequeñas) mejor es la aproximación.

Por ejemplo para la función exponencial, el polinomio de Taylor de orden  $n$  desarrollado en torno al punto  $x_0 = 0$  es,

$$e^x \approx 1 + x + \frac{1}{2}x^2 + \cdots + \frac{1}{n!}x^n = \sum_{i=0}^n \frac{1}{i!}x^i$$

y el del logaritmo natural, desarrollado en torno al punto  $x_0 = 1$ ,

$$\log(x) \approx (x - 1) - \frac{1}{2}(x - 1)^2 + \cdots + \frac{(-1)^{n+1}}{n}(x - 1)^n = \sum_{i=1}^n \frac{(-1)^{i+1}}{i}(x - 1)^i$$

La existencia de un término general para los desarrollos de Taylor de muchas funciones elementales lo hace particularmente atractivo para aproximar funciones mediante un ordenador. Así por ejemplo, la siguiente función escrita en Python, aproxima el valor del logaritmo natural en un punto, empleando un polinomio de Taylor del grado que se desee,

Where  $z$  is an indeterminate point located between  $x$  and  $x_0$ . If we eliminate the last term the function would be approximated by a degree  $n$  polynomial.

The error we get when approximating a function by a degree  $n$  Taylor polynomial can be obtained as,

Looking at the error function, it is easy to realize that decreases when the polynomial degree increases and increases when the distance between  $x$  and  $x_0$  increases.

For instance, the Taylor polynomial of degree  $n$  for the exponential functions, around the point  $x_0 = 0$  is,

and the Taylor polynomial for the logarithm, around the point  $x = 1$  is,

The existence of a general term for Taylor's expansion of many essential functions makes it highly interesting to approximate functions using a computer. For example, the following function, written in Python, approximates the natural logarithm using Taylor's polynomial of any degree we want.

---

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3 """
4 Created on Fri Jul 12 15:02:07 2024
5
6
7 @author: juan
8 """
9 import numpy as np
```

## 348CAPÍTULO/CHAPTER 8. INTERP. Y AJUST. FUNCIONES. \*INTERP. &amp; FUNCT FITTING

```

10 from matplotlib import pyplot as pl
11 def taylorln(x,n):
12     """
13     Esta función aproxima el valor del logaritmo natural de un numero
14     empleando para ello un polinomio de Taylor de grado n desarrollado en
15     torno a x=1. Las variables de entrada son: x, valor para el que se desea
16     calcular el logaritmo. n Grado del polinomio que se empleará en el
17     cálculo. La variable de salida y es el logaritmo de x.
18     This fuction computes (approx) the value of the natural logarithm for a number
19     using a degree n Taylor's polynomial, at x0=1. input variables are: x, value to
20     calculate its logarithm. n degree of the polynomial useto aporaches the log.
21     the funtion returns the locaritmo computed.
22
23     Parameters
24     -----
25     x : Real
26
27     n : int
28         Taylor's polinomial Degree
29
30     Returns
31     -----
32     y : real
33         log(x)
34
35     """
36     y = 0
37     for i in range(1,n+1):
38         y = y+(-1)**(i+1)*(x-1)**i/i
39
40     return(y)
41
42
43 def taylorcum(fun,n,x):
44     """
45
46     Parameters
47     -----
48     fun : takes the taylor series of degree n, described in input function fun
49         and calculates and draws the result for an array of points, x
50         DESCRIPTION.
51     n : TYPE int
52         DESCRIPTION.
53         Taylor's polynomial degree'
54     x : TYPE array of real numbers
55         DESCRIPTION. point to calculate the Taylor series
56
57     Returns
58     -----
59
60     y : Type real
61         DESCRIPTION: array of values computed

```

```

62
63      """
64
65      y =np.array([fun(i,n) for i in x])
66      pl.plot(x,y)

```

La aproximación funciona razonablemente bien para puntos comprendidos en el intervalo  $0 < x < 2$ . La figura 8.1 muestra los resultados obtenidos en dicho intervalo para polinomios de Taylor del logaritmo natural de grados 2, 5, 10 y 20. La linea continua azul representa el valor del logaritmo obtenido con la función de Numpy `log`.

The approximation works reasonably well for points inside the interval  $0 < x < 2$ . Figure 8.1 shows the results achieved inside such interval, using Taylor's polynomial with degrees 2, 5, 10 and 20. The blue line represent the logarithm values we get using the numpy function `log`.

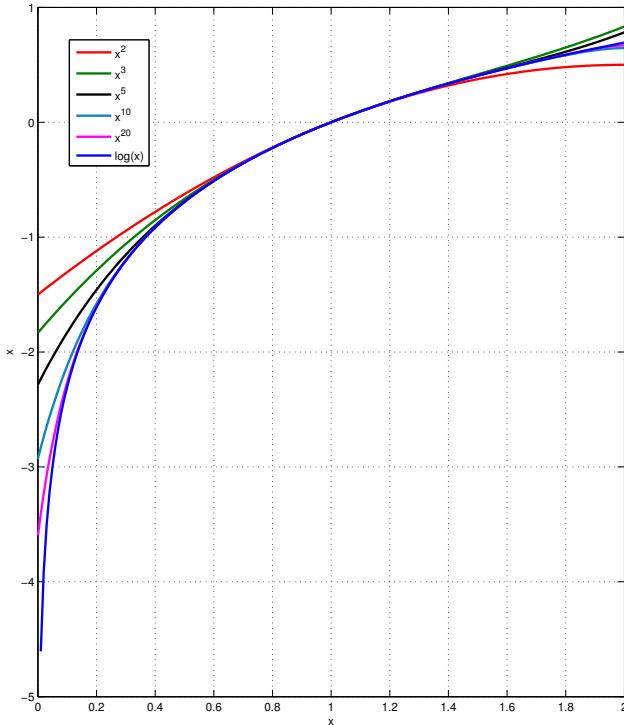


Figura 8.1: Comparación entre resultados obtenidos para polinomios de Taylor del logaritmo natural. (grados 2, 3, 5, 10, 20)

Figure 8.1: A comparison among the results achieved using Taylor polynomials to approach the logarithm. (2, 3, 5, 10, 20 degrees)

Las funciones  $\sin(x)$  y  $\cos(x)$ , son también simples de aproximar mediante polinomios de

Functions  $\sin(x)$  and  $\cos(x)$ , are also easy to approach using Taylor's polynomials. If we

Taylor. Si desarrollamos en torno a  $x_0 = 0$ , la serie del coseno solo tendrá potencias pares mientras que la del seno solo tendrá potencias impares,

$$\cos(x) \approx \sum_{i=0}^n \frac{(-1)^i}{(2i)!} x^{2i}$$

$$\sin(x) \approx \sum_{i=0}^n \frac{(-1)^i}{(2i+1)!} x^{2i+1}$$

En las figuras 8.2(a) y 8.2(b) Se muestran las aproximaciones mediante polinomios de Taylor de las funciones coseno y seno. Para el coseno se han empleado polinomios hasta grado 8 y para el seno hasta grado 9. En ambos casos se dan los resultados correspondientes a un periodo  $(-\pi, \pi)$ . Si se comparan los resultados con las funciones `cos` y `sin`, suministradas por Numpy, puede observarse que la aproximación es bastante buena para los polinomios de mayor grado empleados en cada caso.

## 8.2. Interpolación polinómica.

Se entiende por interpolación el proceso por el cual, dado un conjunto de pares de puntos  $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$  se obtiene una función  $f(x)$ , tal que,  $y_i = f(x_i)$ , para cada par de puntos  $(x_i, y_i)$  del conjunto. Si, en particular, la función empleada es un polinomio  $f(x) \equiv p(x)$ , entonces se trata de interpolación polinómica.

**Teorema de unicidad.** Dado un conjunto  $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$  de  $n + 1$  pares de puntos, tales que todos los valores  $x_i$  de dicho conjunto son diferentes entre sí, solo existe un polinomio  $p(x)$  de grado  $n$ , tal que  $y_i = p(x_i)$  para todos los pares de puntos del conjunto.

Si tratamos de interpolar los puntos con un polinomio de grado menor que  $n$ , es posible que no encontremos ninguno que pase por todos los puntos. Si, por el contrario empleamos un polinomio de grado mayor que  $n$ , nos encontramos con que no es único. Por último

expand around  $x_0 = 0$ , the cosine series will have only even powers and the sine series will only have odd powers,

Figures 8.2(a) and 8.2(b) show approximations to sine and cosine functions using Taylor's polynomials. We have used polynomials up to 8 degree for the cosine function and up to degree 9 for the sine function. In both cases we have calculated the results inside the interval  $(-\pi, \pi)$ . When we compare these results with those yielded by Numpy functions `cos` and `sin`, we see that the approximation is quite good for the higher degree polynomials we have used in each case.

## 8.2. Polynomial interpolation.

We define the interpolation as a process that, departing from a set of data pairs  $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$ , allows us to find a function  $f(x)$  such that  $y_i = f(x_i)$  for all pairs  $(x_i, y_i)$  of the set. In the case we use a polynomial as the interpolating function  $f(x) \equiv p(x)$ , we denote it as polynomial Interpolation.

**The interpolation theorem.** For any set of  $n + 1$  pairs of data points  $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$ , where no two  $x_i$  are the same, there is one and only one polynomial  $p(x)$  of degree  $n$  that interpolates these points, i.e., it satisfies that  $y_i = p(x_i)$  for all pairs on the dataset.

If we try to interpolate the points with a polynomial whose degree is less than  $n$ , it is possible that we will not find one that fits all the pairs in the dataset. Conversely, if we try to use a polynomial with a degree greater than  $n$ , it will not be unique. Eventually, if we use a

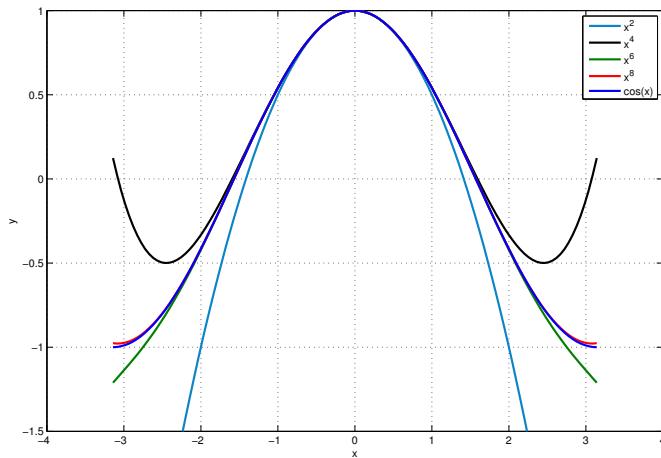
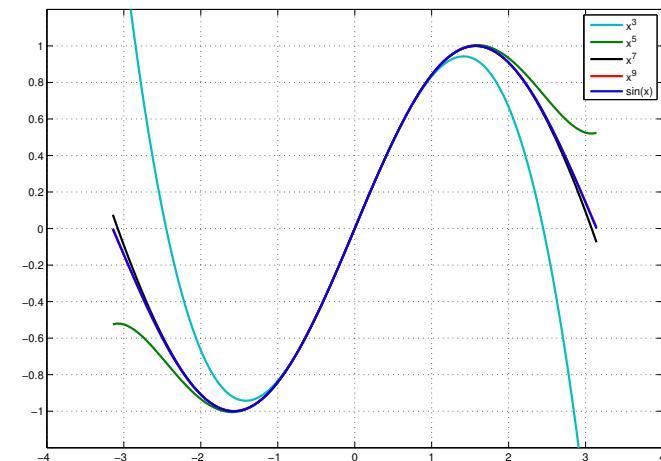
(a)  $\cos(x)$ , polinomios 2, 4, 6 y 8 grados \* polynomials 2, 4, 6 and 8 degrees(b)  $\sin(x)$ , polinomios 3, 5, 7 y 9 grados \*polynomials 2, 4, 6 and 8 degrees

Figura 8.2: Polinomios de Taylor para las funciones coseno y seno  
 Figure 8.2: Taylor polynomial for cosine and sine functions

si el polinomio empleado es de grado  $n$ , entonces será siempre el mismo con independencia del método que empleemos para construirlo.

### 8.2.1. La matriz de Vandermonde

Supongamos que tenemos un conjunto de pares de puntos  $\mathcal{A}$ ,

$x$	$f(x)$
$x_0$	$y_0$
$x_1$	$y_1$
$x_2$	$y_2$
$\vdots$	$\vdots$
$x_n$	$y_n$

Para que un polinomio de orden  $n$ ,

$$p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$$

pase por todos los pares de  $\mathcal{A}$  debe cumplir,

$$y_i = a_0 + a_1x_i + a_2x_i^2 + \cdots + a_nx_i^n, \forall (x_i, y_i) \in \mathcal{A}$$

Es decir, obtendríamos un sistema de  $n$  ecuaciones lineales, una para cada par de valores, en la que las incógnitas son precisamente los  $n + 1$  coeficientes  $a_i$  del polinomio.

Por ejemplo para los puntos,

$x$	$f(x)$
1	2
2	1
3	-2

Obtendríamos,

$$\begin{aligned} a_0 + a_1 \cdot 1 + a_2 \cdot 1^2 &= 2 \\ a_0 + a_1 \cdot 2 + a_2 \cdot 2^2 &= 1 \\ a_0 + a_1 \cdot 3 + a_2 \cdot 3^2 &= -2 \end{aligned}$$

que podríamos expresar en forma matricial como,

$$\begin{pmatrix} 1 & 1 & 1^2 \\ 1 & 2 & 2^2 \\ 1 & 3 & 3^2 \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \\ -2 \end{pmatrix}$$

degree  $n$  polynomial, this polynomial is always the same regardless method used to build it.

#### 8.2.1. The Vandermonde's matrix

Suppose we have a set  $\mathcal{A}$  of data points.

For a polynomial of degree  $n$ ,

$$p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$$

to go through all pairs in  $\mathcal{A}$ , it must satisfy,

So, we would have a system of  $n$  linear equations, one for each pair of data points, where the unknowns are the  $n + 1$  coefficients  $a_i$  of the polynomial.

for example, for the dataset,

we get,

which we can express in matrix form as,

Y en general, para  $n$  pares de datos,

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix}$$

La matriz de coeficientes del sistema resultante recibe el nombre de matriz de Vandermonde. Está formada por las  $n$  primeras potencias de cada uno de los valores de la variable independiente, colocados por filas. Es evidente que cuanto mayor es el número de datos, mayor tenderá a ser la diferencia de tamaño entre los elementos de cada fila. Por ello, en la mayoría de los casos, resulta ser una matriz mal condicionada para resolver el sistema numéricamente. En la práctica, para obtener el polinomio interpolador, se emplean otros métodos alternativos,

### 8.2.2. El polinomio interpolador de Lagrange.

A partir de los valores  $x_0, x_1, \dots, x_n$ , se construye el siguiente conjunto de  $n+1$  polinomios de grado  $n$

$$l_j(x) = \prod_{\substack{k=0 \\ k \neq j}}^n \frac{x - x_k}{x_j - x_k} = \frac{(x - x_0)(x - x_1) \cdots (x - x_{j-1})(x - x_{j+1}) \cdots (x - x_n)}{(x_j - x_0)(x_j - x_1) \cdots (x_j - x_{j-1})(x_j - x_{j+1}) \cdots (x_j - x_n)}$$

Los polinomios así definidos cumplen una interesante propiedad en relación con los valores  $x_0, x_1, \dots, x_n$ , empleados para construirlos,

$$l_j(x_i) = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$$

A partir de estos polinomios podemos construir ahora el siguiente polinomio de interpolación empleando las imágenes  $y_0, y_1, \dots, y_n$  correspondientes a los valores  $x_0, x_1, \dots, x_n$ ,

$$p(x) = \sum_{j=0}^n l_j(x) \cdot y_j$$

Efectivamente, es fácil comprobar que, tal y como se ha construido, este polinomio pasa por los pares de puntos  $(x_i, y_i)$ , puesto que

And, in general, for  $n$  data pairs,

The coefficient matrix of the resulting system is called the Vandermonde's matrix. Its elements are the  $n$  first powers of the independent variable values, allocated by rows. It is easy to notice that when the number of data increases the difference among the elements of a row will tend to increase also. For this reason in most cases the Vandermonde's matrix is a poor conditioned matrix and so not suitable for solving the system numerically. For this reason, in practice, the interpolation polynomial is computed using other alternative methods.

### 8.2.2. Lagrange Interpolating Polynomial

Departing from the values  $x_0, x_1, \dots, x_n$ , we build the following set of  $n+1$  polynomial of degree  $n$

These polynomials exhibit an interesting property when evaluated at the points  $x_0, x_1, \dots, x_n$ , we have used for building them.

From these polynomials we can build a interpolation polynomial using the codomain values  $y_0, y_1, \dots, y_n$  corresponding to the domain values  $x_0, x_1, \dots, x_n$ ,

It is easy to check that, using this method for building the interpolation polynomial, it passes through all pairs of points  $(x_i, y_i)$ , be-

$$p(x_i) = y_i.$$

El siguiente código de en Python calcula el valor en un punto  $x$  cualquiera del polinomio de interpolación de Lagrange construido a partir un conjunto de puntos  $\mathcal{A} \equiv \{(x_i, y_i)\}$ .

$$\text{cause } p(x_i) = y_i.$$

The following python function `lagrang` calculates, at any point  $x$ , the Lagrange interpolating polynomial value built from a set of pairs of points  $\mathcal{A} \equiv \{(x_i, y_i)\}$ .

---

lgr-pol.py

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Created on Tue Jul 23 10:40:23 2024
5
6  @author: juan
7  A code to implement the Lagrange polynomial
8  """
9  import numpy as np
10
11 def lagrang(x,y,x1):
12     """
13         Function to calculate the Lagrange interpolating polynomial at point x1
14
15         Parameters
16         -----
17         x : TYPE real np array
18             DESCRIPTION. Table Values x to be interpolated
19         y : TYPE real np array
20             DESCRIPTION. Table values y to be interpolated
21         x1 : TYPE real
22             DESCRIPTION. Point a which the polynomial is evaluated
23
24         Returns
25         -----
26         y1 : TYPE real
27             DESCRIPTION. Lagrange polynomial Value at x1. y1 = lagrang(x1)
28
29         """
30         y1 = 0
31         n = x.shape[0]
32         for j in range(0,n):
33             l1j = 1
34             for i in range(0,j):
35                 l1j = l1j*(x1-x[i])/(x[j]-x[i])
36             for i in range(j+1,n):
37                 l1j = l1j*(x1-x[i])/(x[j]-x[i])
38             y1 = y1 + l1j*y[j]
39         return(y1)
40
41 def langrmult(x,y,x1):
42     """
43         Takes the funtion lagrang and calculates the values of the Lagrange
44         polinomial at any point in array x1
45

```

```

46     Parameters
47     -----
48     x : TYPE real np array
49         DESCRIPTION. Table Values x to be interpolated
50     y : TYPE real np array
51         DESCRIPTION. Table values y to be interpolated
52     x1 : TYPE real
53         DESCRIPTION. array of point a which the polynomial is evaluated
54
55     Returns
56     -----
57     y1 : TYPE real
58         DESCRIPTION. Lagrange polynomial Valueat x1. y1 = PLagrange(x1)
59
60     """
61     yaux = [] #fast way to create an array same size as x
62     for i in x1:
63         yaux.append(lagrang(x,y,i))
64     y1 = np.array(yaux)
65     return(y1)
66
67

```

---

### 8.2.3. Diferencias divididas.

Tanto el método de la matriz de Vandermonde como el de los polinomios de Lagrange, presentan el inconveniente de que si se añade un dato más  $(x_{n+1}, y_{n+1})$  a la colección de datos ya existentes, es preciso recalcular el polinomio de interpolación desde el principio.

El método de las diferencias divididas, permite obtener el polinomio de interpolación en un número menor de operaciones que en el caso del polinomio de Lagrange y además, el cálculo se hace escalonadamente, aprovechando todos los resultados anteriores cuando se añade al polinomio la contribución de un nuevo dato.

El polinomio de orden  $n$  de diferencias divididas se construye de la siguiente manera,

$$p_n(x) = a_0 + (x - x_0) \cdot a_1 + (x - x_0) \cdot (x - x_1) \cdot a_2 + \cdots + (x - x_0) \cdot (x - x_1) \cdots (x - x_{n-2}) \cdot (x - x_{n-1}) \cdot a_n$$

Donde,  $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$ , representan los datos para los que se quiere calcular el polinomio interpolador de grado  $n$ . Si sustituimos los datos en el polinomio, llegamos a un sistema de ecuaciones, triangular inferior,

### 8.2.3. Divided differences.

The two method we have studied so far, Vandermonde matrix and Lagrange polynomial, have a common drawback. If we add a new pair of data  $(x_{n+1}, y_{n+1})$  to the existing data collection, it is necessary to recalculate the interpolation polynomial from scratch.

The divided differences algorithm allows us to build the interpolation polynomial performing less operations than in the case of Lagrange polynomial. Besides, the computing is carried out stepwise, making use of all previous results when a new data pair is added to calculate the interpolation polynomial.

We build the divided differences polynomial of degree  $n$  as follows,

Where  $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$ , represent the dataset for which we want to calculate the interpolation polynomial of degree  $n$ . If we evaluate the polynomial using the dataset we arrive to a lower triangular system of

en el que las incógnitas son los coeficientes del polinomio.

linear equation, where the polynomial coefficients are the unknowns.

$$\begin{aligned}
 a_0 &= y_0 \\
 a_0 + (x_1 - x_0)a_1 &= y_1 \\
 a_0 + (x_2 - x_0)a_1 + (x_2 - x_0)(x_2 - x_1)a_2 &= y_2 \\
 \dots \\
 a_0 + (x_n - x_0)a_1 + \dots + (x_n - x_0)(x_n - x_1) \dots (x_n - x_{n-2})(x_n - x_{n-1})a_n &= y_n
 \end{aligned}$$

Este sistema se resuelve explícitamente empleando un esquema de diferencias divididas.

La diferencia dividida de primer orden entre dos puntos  $(x_0, y_0)$  y  $(x_1, y_1)$  se define como,

$$f[x_0, x_1] = \frac{y_1 - y_0}{x_1 - x_0}$$

Para tres puntos,  $(x_0, y_0)$ ,  $(x_1, y_1)$  y  $(x_2, y_2)$ , se define la diferencia dividida de segundo orden como,

$$f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}$$

y, en general definiremos la diferencia dividida de orden  $i$  para  $i + 1$  puntos como,

$$f[x_0, x_1, \dots, x_i] = \frac{f[x_1, x_2, \dots, x_i] - f[x_0, x_1, \dots, x_{i-1}]}{x_i - x_0}$$

Si despejamos por sustitución progresiva los coeficientes del polinomio de interpolación del sistema triangular inferior obtenido, cada coeficiente puede asociarse a una diferencia dividida,

This system can be solved using the divided differences algorithm.

We define the first-order two-point divided difference  $(x_0, y_0)$  y  $(x_1, y_1)$  as,

For three point,  $(x_0, y_0)$ ,  $(x_1, y_1)$  y  $(x_2, y_2)$ , we define the second-order divided difference as,

and eventually, we can define the  $i$ -order divide difference for  $i + 1$  points as,

We can now get the polynomial coefficients applying progressive substitutions to the lower triangular system defined above, then, each coefficient may be related with a divided difference,

$$\begin{aligned}
 a_0 &= f[x_0] = y_0 \\
 a_1 &= f[x_0, x_1] \\
 &\vdots \\
 a_i &= f[x_0, x_1, \dots, x_i] \\
 &\vdots \\
 a_n &= f[x_0, x_1, \dots, x_n]
 \end{aligned}$$

Por tanto, podemos obtener directamente los coeficientes del polinomio calculando las diferencias divididas. Veamos un ejemplo em-

Therefore we can get the polynomial coefficients straightforwardly if we compute the divided differences. Let's see an example using

pleando el siguiente conjunto de cuatro datos, | the following four data set.

x	0	1	3	4
y	1	-1	2	3

Habitualmente, se construye a partir de los datos una tabla, como la 8.2, de diferencias divididas. Las primera columna contiene los valores de la variable  $x$ , la siguiente los valores de las diferencias divididas de orden cero (valores de  $y$ ). A partir de la segunda, las siguientes columnas contienen las diferencias divididas de los elementos de la columna anterior, calculados entre los elementos que ocupan filas consecutivas. La tabla va perdiendo cada vez una fila, hasta llegar a la diferencia dividida de orden  $n - 1$  de todos los datos iniciales.

Usually, we build the divided difference polynomial, using a table such as table 8.2. the first column contains the values of variable  $x$  the second the values of the cero-order divided differences (values of variable  $y$ ). From the second one on, the following columns contain the divided differences of the elements held in the previous column. These differences are calculated using the elements located on consecutive rows. The table lost a row each time we advance a column. When we arrive to the  $n - 1$ -order divided difference, we have a single value that depends on all initial data.

Tabla 8.2: Tabla de diferencia divididas para cuatro datos  
Table 8.2: four data divided difference table

$x_i$	$y_i$	$f[x_i, x_{i+1}]$	$f[x_i, x_{i+1}, x_{i+2}]$	$f[x_i, x_{i+1}, x_{i+2}, x_{i+3}]$
$x_0 = 0$	$y_0 = 1$	$f[x_0, x_1] = -2$	$f[x_0, x_1, x_2] = 7/6$	$f[x_0, x_1, x_2, x_3] = -1/3$
$x_1 = 1$	$y_1 = -1$	$f[x_1, x_2] = 3/2$	$f[x_1, x_2, x_3] = -1/6$	
$x_2 = 3$	$y_2 = 2$	$f[x_2, x_3] = 1$		
$x_3 = 4$	$y_3 = 3$			

Los coeficientes del polinomio de diferencias divididas se corresponden con los elementos de la primera fila de la tabla. Por lo que en nuestro ejemplo el polinomio resultante sería,

$$p_3(x) = 1 - 2x + \frac{7}{6}x(x-1) - \frac{1}{3}x(x-1)(x-3)$$

Es importante hacer notar que el polinomio de interpolación obtenido por diferencias divididas siempre aparece representado como suma de productos de binomios  $(x - x_0)(x - x_1) \dots$  y los coeficientes obtenidos corresponden a esta representación y no a la representación habitual de un polinomio como suma de potencias de la variable  $x$ .

El siguiente código permite calcular el polinomio de diferencias divididas a partir de un conjunto de  $n$  datos. Como el polinomio de diferencias divididas toma una forma especial, es preciso tenerlo en cuenta a la hora de calcular su valor en un punto  $x$  determinado. La función `difdiv` permite obtener los coefi-

The divided differences polynomial coefficients are the elements held in the first row of the table. So, for our example, we obtain the following interpolation polynomial,

Notice that the interpolation polynomial, built using divided differences, it always represented as a sum of binomials products  $(x - x_0)(x - x_1) \dots$  and the coefficients computed belong to this representation a not to the standard polynomial representation as a sum of variable  $x$  powers.

The following code implement the divided differences polynomial from a set of  $n$  data. We must take into account the special form of the polynomial to calculate the value ot takes in an specific  $x$  point. Function `difdiv` computes the polynomiañ coefficients departing from a set  $x, y$  of data. Function `evdif` calculates the value of the polynomial in a point

cientes del polinomio de diferencias divididas a partir de una colección de datos  $x, y$ . La función `evdif` permite calcular el valor que toma el polinomio en un punto cualquiera  $x_i$ , a partir de los coeficientes calculados y los valores  $x$  de la colección de datos.

whatsoever, using the computed coefficients and the  $x$  values of the data set.

---

dif.div.py

---

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Created on Wed Jul 24 15:22:01 2024
5  This file define two funtion. one to calculate the divided differences
6  polynomial coefficients and the second one to evaluate the divided differences
7  polynomial any point
8  @author: juan
9  """
10 import numpy as np
11 import matplotlib.pyplot as pl
12 def difdiv(x,y):
13     """
14         This functions computes the coefficient of a the divided differences
15         polynomial using thw data arrays x,y
16
17     Parameters
18     -----
19     x : TYPE double
20         DESCRIPTION. numpy array with data x
21     y : TYPE double
22         DESCRIPTION. numpy array with data y
23
24     Returns
25     -----
26     a : TYPE double
27         DESCRIPTION: coefficientes of the polynomial
28          $p_n(x) = a_0 + (x-x_0)a_1 + (x-x_0)(x-x_1)a_2 + \dots$ 
29          $\dots + (x-x_0)(x-x_1)\dots(x-x_{n-2})(x-x_{n-1})a_n$ 
30     """
31     n = x.shape[0] #number of data
32     #we use the y variable to inisialise the vector of coefficients
33     a = y.copy() #warning I need to copy to avoid overwrite the values of y
34     #we must compute n differences
35     for j in range(1,n): #start in 1 we already have the order zero differences
36         for i in range(j,n):
37             a[i] = (a[i]-y[i-1])/(x[i]-x[i-j])
38     y = a.copy()
39     return(a)
40
41 def evdif(a,x,xi):
42     """
43         Evaluates the divided differences polynomial from points x and coefficients
44         y.

```

```

45
46     Parameters
47     -----
48     y : TYPE double
49         DESCRIPTION. numpy array with divided differences polynomial coefficients
50     x : TYPE double
51         DESCRIPTION. numpy array with x data from the table to be interpolated
52     xi : TYPE double
53         DESCRIPTION. point to calculate the polynomial at. It can be also an array
54
55     Returns
56     -----
57     y1: TYPE: double
58         DESCRIPTION: computed value of the polynomial at xi
59     """
60     n = a.shape[0]
61     yi = a[0] #copy the first coefficient into the result
62     for k in range(1,n):
63         #product of binomials to be multiplied for the coefficients
64         binprod = 1
65         for j in range(k):
66             binprod = binprod*(xi-x[j])
67         yi = yi +a[k]*binprod
68     return(yi)

```

---

#### 8.2.4. El polinomio de Newton-Gregory

Supone una simplificación al cálculo del polinomio de diferencias divididas para el caso particular en que los datos se encuentran equiespaciados y dispuestos en orden creciente con respecto a los valores de la coordenada  $x$ .

En este caso, calcular los valores de las diferencias es mucho más sencillo. Si pensamos en las diferencias de primer orden, los denominadores de todas ellas son iguales, puesto que los datos están equiespaciados,

#### 8.2.4. The Newton-Gregory polynomial

This polynomial is a simplification of the divided difference polynomial for the case in which the  $x$  data of the data set are equispaced and increasingly ordered.

For this case, it is much easier to compute the values of the differences. Think, for instance, in the first-order differences, as far as the data are equispaced all denominators are equal,

$$\Delta x \equiv x_i - x_{i-1} = h$$

En cuanto a los numeradores, se calcularían de modo análogo al de las diferencias divididas normales,

$$\Delta y_0 = y_1 - y_0, \Delta y_1 = y_2 - y_1, \dots, \Delta y_i = y_{i+1} - y_i, \dots, \Delta y_{n-1} = y_n - y_{n-1}$$

Las diferencias de orden superior para los numeradores se pueden obtener de modo re-

Concerning the numerators, they are computed as in the standard case of divided differences,

Higher order differences can be computed recursively from the first-order differences be-

cursivo, a partir de las de orden uno, puesto que los denominadores de todas ellas  $h$ , son iguales.

$$\Delta^2 y_0 = \Delta(\Delta y_0) = (y_2 - y_1) - (y_1 - y_0) = (y_2 - 2y_1 + y_0)$$

En este caso, el denominador de la diferencia sería  $x_2 - x_0 = 2h$ , y la diferencia tomaría la forma,

$$f[x_0, x_1, x_2] = \frac{\Delta^2 y_0}{2h^2}$$

En general, para la diferencias de orden  $n$  tendríamos,

$$\Delta^n y_0 = y_n - \binom{n}{1} \cdot y_{n-1} + \binom{n}{2} \cdot y_{n-2} - \cdots + (-1)^n \cdot y_0$$

Donde se ha hecho uso de la expresión binomial,

$$\binom{k}{l} = \frac{k!}{l! \cdot (k-l)!}$$

Para obtener la diferencia dividida de orden  $n$ , bastaría ahora dividir por  $n! \cdot h^n$ .

$$f[x_0, x_1, \dots, x_n] = \frac{\Delta^n y_0}{n! \cdot h^n}$$

A partir de las diferencias, podemos representar el polinomio de diferencias divididas resultante como,

$$p_n(x) = y_0 + \frac{x - x_0}{h} \Delta y_0 + \frac{(x - x_1) \cdot (x - x_0)}{2 \cdot h^2} \Delta^2 y_0 + \cdots + \frac{(x - x_{n-1}) \cdots (x - x_1) \cdot (x - x_0)}{n! \cdot h^n} \Delta^n y_0$$

Este polinomio se conoce como el polinomio de Newton-Gregory, y podría considerarse como una aproximación numérica al polinomio de Taylor de orden  $n$  de la posible función asociada a los datos empleados.

En este caso, podríamos construir la tabla para obtener los coeficientes del polinomio, calculando en cada columna simplemente las diferencias de los elementos de la columna anterior. Por ejemplo,

Una vez calculadas las diferencias, basta dividir por  $n! \cdot h^n$  los elementos de la primera fila de la tabla,

$$a_0 = 1, a_1 = \frac{-2}{1}, a_2 = \frac{5}{2 \cdot 1^2}, a_3 = \frac{-7}{6 \cdot 1^3}$$

cause the denominator of them all,  $h$  are equal.

In this case, the difference denominator would be  $x_2 - x_0 = 2h$ , and the difference would take the form,

In genral, for the order-n differences we obtain,

Where we have used the binomial expression,

And, eventually, we obtain the order-n divided difference just dividing by  $n! \cdot h^n$ .

Once we have got the differences we can write the divided differences polynomial as,

This polynomial is known as the Newton-Gregory polynomial, and it could be considered as a numerical approximation to the  $n$ -degree Taylor polynomial of the (possible) function associated to the dataset.

In this case, we can build the table to obtain the polynomial coefficients, computing in each column just the differences of the previous column. For example,

Once we have computed the differences, it is enough to divide by  $n! \cdot h^n$  the elements of the table first row,

Tabla 8.3: Tabla de diferencias para el polinomio de Newton-Gregory de cuatro datos  
 Table 8.3: Table of differences for a Newton-Gregory polynomial of four data

$x_i$	$y_i$	$\Delta y_i$	$\Delta^2 y_i$	$\Delta^3 y_i$
$x_0 = 0$	$y_0 = 1$	-2	5	-7
$x_1 = 1$	$y_1 = -1$	3	-2	
$x_2 = 2$	$y_2 = 2$	1		
$x_3 = 3$	$y_3 = 3$			

El siguiente código muestra un ejemplo de implementación en Python del polinomio de Newton-Gregory

The following code shows an example of Python implementation for the Newton-Gregory polynomial

---

newton\_gregory.py

---

```

1  # -*- coding: utf-8 -*-
2  """
3  Created on Thu Jul 25 20:12:49 2024
4  A gruby version of Newton-Gregory polynomial
5  @author: abierto
6  """
7
8  import numpy as np
9  from dif_div import evdif
10 def newgre(x,y,x1=0):
11     """
12
13     Parameters
14     -----
15     x : TYPE numpy array of data
16         DESCRIPTION.
17     y : TYPE numpy array of data
18         DESCRIPTION.
19     x1 : TYPE a point to calculate the value the polynomial
20         takes
21         DESCRIPTION.
22         This function takes a data set, represented by the
23         x and y array and obtain the corresponding
24         newton-gregory polynomial.Besides it calculates the
25         value of the polypolimial at the point x1. if no x1
26         value is supplied then it takes x1 =0 by default.
27         The program returns the polynomial coeficients and
28         the value calculated at point x1
29
30     Returns
31     -----
32     a : TYPE numpy array of polynomial coeficients
33         DESCRIPTION.
34     y1: TYPE real value
35         DESCRIPTION. The value the polynomial takes at x1
36
37     """
38     n = x.shape[0]
```

```

39  a = y.copy() #we start the coefficient with the 0-order differences, i.e.
40      #y values. remember use a copy to avoid overwrite the arrays
41  h = x[1] -x[0]
42  #here start the loop to calculate the differences
43  for j in range(1,n):
44      #for each iteration we calculate the differences
45      #of higher order. But we only need the first difference. So we restart
46      #the inner loop in the value j of the outer one
47      for i in range(j,n):
48          #now it is enough to divide for the distance h between x point
49          #multiplied by order j of the difference
50          a[i] = (a[i]-y[i-1])/(j*h)
51  y = a.copy()
52  #now we calculate the value(s) of the polynomial using the
53  #function built to evaluate divided differences polynomials
54  # if type(x1) != np.ndarray:
55  #     x1 = np.array([x1])
56  # y1 = []
57  # for i in x1:
58  #     y1.append(evdif(a,x,i))
59  # y1 = np.array(y1)
60  y1 = evdif(a,x,x1)
61  return(a,y1)

```

---

### 8.3. Interpolación por intervalos.

Hasta ahora, hemos visto cómo interpolar un conjunto de  $n + 1$  datos mediante un polinomio de grado  $n$ . En muchos casos, especialmente cuando el número de datos es suficientemente alto, los resultados de dicha interpolación pueden no ser satisfactorios. La razón es que el grado del polinomio de interpolación crece linealmente con el número de puntos a interpolar, así por ejemplo para interpolar 11 datos necesitamos un polinomio de grado 10. Desde un punto de vista numérico, este tipo de polinomios pueden dar grandes errores debido al redondeo. Por otro lado, y dependiendo de la disposición de los datos para los que se realiza la interpolación, puede resultar que el polinomio obtenido tome una forma demasiado complicada para los valores comprendidos entre los datos interpolados..

La figura 8.3 muestra el polinomio de interpolación de grado nueve para un conjunto de 10 datos. Es fácil darse cuenta, simplemen-

### 8.3. Piecewise interpolation

So far, we have seen how to interpolate a set of  $n + 1$  data using a degree  $n$  polynomial. In many case, in particular when the number of data is high, the results of such interpolation could be very poor. The problem comes from the linear increasing of the polynomial degree with the number of data. So, if we want to build the interpolation polynomial for a 11 data, we come up with a 10-degree polynomial. From a numerical point of view, this kinda polynomial are prone to cast large errors due to the rounding process. On the other hand, depending on how data used to compute the interpolation are distributed, the polynomial could take a too complicate shape that hardly can be related with the information held in the dataset.

Figure 8.3 shows a interpolating polynomial of degree nine obtained from a set of ten data. It is easy to realise, just for simple inspection, that there is not reason to justify the polynomial curvature between points 1 and 2 or between the points 9 and 10.

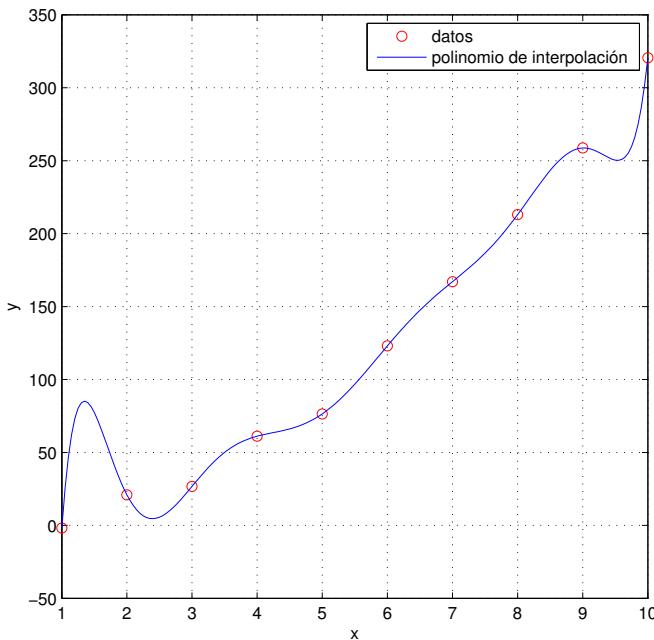


Figura 8.3: Polinomio de interpolación de grado nueve obtenido a partir de un conjunto de diez datos

Figure 8.3: Nine degree interpolating polynomial obtained using a set of ten data

te observando los datos, que no hay ninguna razón que justifique las curvas que traza el polinomio entre los puntos 1 y 2 o los puntos 9 y 10, por ejemplo.

En muchos casos es preferible no emplear todos los datos disponibles para obtener un único polinomio de interpolación. En su lugar, lo que se hace es dividir el conjunto de datos en varios grupos —normalmente se agrupan formando intervalos de datos consecutivos— y obtener varios polinomios de menor grado, de modo que cada uno interpole los datos de un grupo distinto.

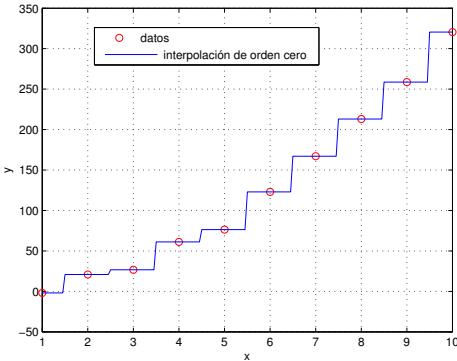
El grado de los polinomios empleados deberá estar, en principio, relacionado con los datos contenidos en cada tramo.

**interpolación de orden cero** si hacemos que cada intervalo contenga un solo dato, obtendríamos polinomios de interpolación de grado cero,  $a_{0i} = y_i$ . El resultado, es un conjunto

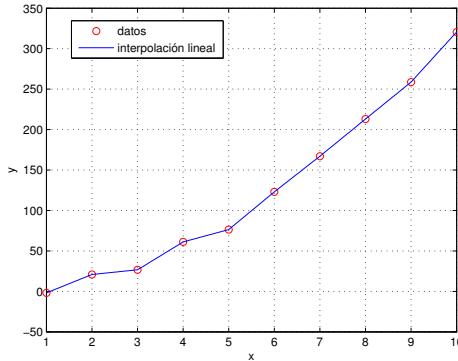
For these reasons, in many cases it is better not to use the whole dataset to build a single interpolation polynomial of maximum degree. Instead, a common practice is to divide the dataset in several groups of data —usually they are gathered using interval of consecutive data— and build several polynomial of lower degree, each one interpolating the data of a different group.

The degree of the polynomials should be related with the number of data allocated in each interval.

**Zero-order interpolation.** If we get interval which only hold a single data pair  $X, y$ , then we get zero-order interpolating polynomials,  $a_{0i} = y_i$ . The result is a stepwise inter-



(a) Interpolación de orden cero / Zero-order interpolation



(b) Interpolación lineal Linear interpolation

Figura 8.4: Interpolaciones de orden cero y lineal para los datos de la figura 8.3

Figure 8.4: Zero-order and linear interpolation for the figure data

de escalones cuya valor varía de un intervalo a otro de acuerdo con el dato representativo contenido en cada tramo. La figura 8.4(a) muestra el resultado de la interpolación de orden cero para los mismos diez datos de la figura 8.3.

**interpolación lineal.** En este caso, se dividen los datos en grupos de dos. Cada par de datos consecutivos se interpola calculando la recta que pasa por ellos. La interpolación lineal se emplea en muchas aplicaciones debido a su sencillez de cálculo. La figura 8.4(b), muestra el resultado de aproximar linealmente los mismos datos contenidos en los ejemplos anteriores.

Siguiendo el mismo procedimiento, aumentando el número de datos contenidos en cada intervalo, podríamos definir una interpolación cuadrática, con polinomios de segundo grado, tomando intervalos que contengan tres puntos, una interpolación cúbica, para intervalos de cuatro puntos etc.

### 8.3.1. Interpolación mediante splines cúbicos

Hemos descrito antes cómo el polinomio interpolador de orden  $n$  para un conjunto de  $n + 1$  datos puede presentar el inconveniente

polation which values changes from one interval to the next, taken the data value defined for each interval according to the dataset. Figure 8.4(a) shows the zero-order interpolation result for the same ten data of figure 8.3.

**Linear interpolation.** In this case, we divide the dataset in groups of two data. Each two consecutive data are interpolated calculating the line that pass through them. The linear interpolation is very commonly used due to its computing simplicity. Figure 8.4(b) shows the result of interpolating the same data utilised in previous examples, using linear interpolation.

Following the same procedure, we can increase the number of data include in each interval and define quadratic interpolation, using second-degree polynomials and taking three points in each interval; cubic interpolation, using third-degree polynomial and four point intervals. etc.

#### 8.3.1. Cubic spline interpolation

WE have seen how the  $n$ -degree interpolating polynomial for a set of  $n + 1$  data, could present a quite complex shape that does not represent well the information supplied by the dataset. The piecewise interpolation that we

de complicar excesivamente la forma de la curva obtenida entre los puntos interpolados. La interpolación a tramos que acabamos de describir, simplifica la forma de la curva entre los puntos pero presenta el problemas de la continuidad en las uniones entre tramos sucesivos. Sería deseable encontrar métodos de interpolación que fueran capaces de solucionar ambos problemas simultáneamente. Una buena aproximación a dicha solución la proporcionan los *splines*.

Una función *spline* está formada por un conjunto de polinomios, cada uno definido en un intervalo, que se unen entre sí obedeciendo a ciertas condiciones de continuidad.

Supongamos que tenemos una tabla de datos cualquiera,

x	$x_0$	$x_1$	...	$x_n$
y	$y_0$	$y_1$	...	$y_n$

Para construir una función *spline*  $S$  de orden  $m$ , que interpole los datos de la tabla, se definen intervalos tomando como extremos dos puntos consecutivos de la tabla y un polinomio de grado  $m$  para cada uno de los intervalos,

$$S = \begin{cases} S_0(x), & x \in [x_0, x_1] \\ S_1(x), & x \in [x_1, x_2] \\ \vdots \\ S_i(x), & x \in [x_i, x_{i+1}] \\ \vdots \\ S_{n-1}(x), & x \in [x_{n-1}, x_n] \end{cases}$$

Para que  $S$  sea una función Spline de orden  $m$  debe cumplir que sea continua y tenga  $m-1$  derivadas continuas en el intervalo  $[x_0, x_n]$  en que se desean interpolar los datos.

Para asegurar la continuidad, los polinomios que forman  $S$  deben cumplir las siguientes condiciones en sus extremos;

have described so far, simplifies the shape of the interpolating curve but have in turn the problem that it loses the continuity in the union between consecutive intervals. It could be valuable to find interpolation methods that may be able to cope with both problems simultaneously. A good approach to solve these problem is supplied by *spline* interpolation.

A *spline* is function built using a set of polynomials, any one of them defined in an interval. The spline polynomial are connected in the ends of the intervals meeting certain continuity conditions.

Suposse we have a data table whatsoever,

To build a  $m$  order *spline* function  $S$  for interpolating the table data, we define intervals taking two consecutive point onn the table as limits. Then, we define an  $m$  degree polynomial for each interval.

For  $s$  to be a order  $m$  spline function it should be continue and it should have  $m-1$  continue derivatives in the interval  $[x_0, x_n]$  in which we want to interpolate the data.

To ensure the continuity of the polynomials that build  $S$  they must satisfy the following condition at their ends,

$$\begin{aligned} S_i(x_{i+1}) &= S_{i+1}(x_{i+1}), \quad (1 \leq i \leq n-1) \\ S'_i(x_{i+1}) &= S'_{i+1}(x_{i+1}), \quad (1 \leq i \leq n-1) \\ S''_i(x_{i+1}) &= S''_{i+1}(x_{i+1}), \quad (1 \leq i \leq n-1) \\ &\vdots \\ S_i^{m-1}(x_{i+1}) &= S_{i+1}^{m-1}(x_{i+1}), \quad (1 \leq i \leq n-1) \end{aligned}$$

Es decir, dos polinomios consecutivos del spline y sus  $m - 1$  primeras derivadas, deben tomar los mismos valores en el extremo común.

Una consecuencia inmediata de las condiciones de continuidad exigidas a los splines es que sus derivadas sucesivas,  $S'$ ,  $S''$ , ... son a su vez funciones spline de orden  $m - 1$ ,  $m - 2$ , ... Por otro lado, las condiciones de continuidad suministran  $(n - 1) \cdot m$  ecuaciones que, unidas a las  $n + 1$  condiciones de interpolación —cada polinomio debe pasar por los datos que constituyen los extremos de su intervalo de definición—, suministran un total de  $n \cdot (m + 1) - (m - 1)$  ecuaciones. Este número es insuficiente para determinar los  $(m + 1) \cdot n$  parámetros correspondientes a los  $n$  polinomios de grado  $m$  empleados en la interpolación. Las  $m - 1$  ecuaciones que faltan se obtienen imponiendo a los splines condiciones adicionales.

**Splines cúbicos.** Los splines más empleados son los formados por polinomios de tercer grado. En total, tendremos que determinar  $(m + 1) \cdot n = 4 \cdot n$  coeficientes para obtener todos los polinomios que componen el spline. Las condiciones de continuidad más la de interpolación suministran en total  $3 \cdot (n - 1) + n + 1 = 4 \cdot n - 2$  ecuaciones. Necesitamos imponer al spline dos condiciones más. Algunas típicas son,

1. Splines naturales  $S''(x_0) = S''(x_n) = 0$
2. Splines con valor conocido en la primera derivada de los extremos  $S'(x_0) = y'_0, S'(x_n) = y'_n$
3. Splines periódicos,

$$\begin{cases} S(x_0) = S(x_n) \\ S'(x_0) = S'(x_n) \\ S''(x_0) = S''(x_n) \end{cases}$$

Intentar construir un sistema de ecuaciones para obtener a la vez todos los coeficientes de todos los polinomios es una tarea excesivamente compleja porque hay demasiados parámetros. Para abordar el problema partimos del hecho de que  $S''(x)$  es también un

That is, two consecutive polynomial belonging to the spline and their  $m - 1$  first derivatives must take the same values in the common end.

A straightforward consequence of the continuity conditions impose to spline functions is that their successive derivatives  $S'$ ,  $S''$ , ... are in turn order  $m - 1$ ,  $m - 2$ , ... spline functions too. Besides, the continuity conditions supply  $(n - 1) \cdot m$  equations that, together with the interpolation conditions —each polynomial should pass through the two data point that defines the ends of its definition interval—, sum up  $n \cdot (m + 1) - (m - 1)$  equations. This number is insufficient for obtaining the  $(m + 1) \cdot n$  parameters belonging to the the  $n$  polynomial of degree  $m$  used in the spline interpolation. We need  $m - 1$  equations more that are defined imposing to the polynomials additional conditions.

**Cubic Spline.** Probably, the most used splines are those composed of third-degree polynomials. We must need to determine  $(m + 1) \cdot n = 4 \cdot n$  coefficients to obtain all the polynomials that compose the spline. The continuity plus the interpolation conditions supply  $3 \cdot (n - 1) + n + 1 = 4 \cdot n - 2$  equations. We need to impose two more conditions to the spline. Some frequently used conditions are,

1. natural Spline  $S''(x_0) = S''(x_n) = 0$
2. Spline with a known value for the derivatives at the ends of the interval  $S'(x_0) = y'_0, S'(x_n) = y'_n$
3. Periodic Spline,

Try to build a system of equations to obtain all the coefficients of all the polynomial at a time, is an arduous task. There are too many parameters. We can address the problem starting with  $S''(x)$ , which it is also a order-1 spline for the points we want to interpolate. If we

spline de orden 1 para los puntos interpolados. Si los definimos como,

$$S_i''(x) = -M_i \frac{x - x_{i+1}}{h_i} + M_{i+1} \frac{x - x_i}{h_i}, \quad i = 0, \dots, n-1$$

donde  $h_i = x_{i+1} - x_i$  representa el ancho de cada intervalo y donde cada valor  $M_i = S''(x_i)$  será una de las incógnitas que debaremos resolver.

Si integramos dos veces la expresión anterior,

define it as,

Where  $h_i = x_{i+1} - x_i$  stand for each interval width and where the value  $M_i = S''(x_i)$  will be the unknown we have to solve.

Now, we integrate two times this expression,  $S_i''(x)$ , to obtain,

$$\begin{aligned} S_i'(x) &= -M_i \frac{(x - x_{i+1})^2}{2 \cdot h_i} + M_{i+1} \frac{(x - x_i)^2}{2 \cdot h_i} + A_i, \quad i = 0, \dots, n-1 \\ S_i(x) &= -M_i \frac{(x - x_{i+1})^3}{6 \cdot h_i} + M_{i+1} \frac{(x - x_i)^3}{6 \cdot h_i} + A_i(x - x_i) + B_i, \quad i = 0, \dots, n-1 \end{aligned}$$

Empezamos por imponer las condiciones de interpolación: el polinomio  $S_i$  debe pasar por el punto  $(x_i, y_i)$ ,

Let's start imposing the interpolation conditions: the polynomial  $S_i$  must pass through the point  $(x_i, y_i)$ ,

$$S_i(x_i) = -M_i \frac{(x_i - x_{i+1})^3}{6 \cdot h_i} + B_i = y_i \Rightarrow B_i = y_i - \frac{M_i \cdot h_i^2}{6}, \quad i = 0, \dots, n-1$$

A continuación imponemos continuidad del spline en los nodos comunes: El polinomio  $S_{i-1}$  también debe pasar por el punto  $(x_i, y_i)$ ,

Then, we impose the continuity of the spline in the points common to two polynomials: Polynomial  $S_{i-1}$  must also pass through the point  $(x_i, y_i)$ ,

$$\begin{aligned} S_{i-1}(x_i) &= M_i \frac{(x_i - x_{i-1})^3}{6 \cdot h_i} + A_{i-1}(x_i - x_{i-1}) + y_{i-1} - \overbrace{\frac{M_{i-1} \cdot h_{i-1}^2}{6}}^{B_{i-1}} = y_i \Rightarrow \\ &\Rightarrow A_{i-1} = \frac{y_i - y_{i-1}}{h_{i-1}} - \frac{M_i - M_{i-1}}{6} \cdot h_{i-1}, \quad i = 1, \dots, n \end{aligned}$$

Y por tanto,

and thus,

$$A_i = \frac{y_{i+1} - y_i}{h_i} - \frac{M_{i+1} - M_i}{6} \cdot h_i, \quad i = 0, \dots, n-1$$

En tercer lugar imponemos la condición de que las derivadas también sean continuas en los nodos comunes,

Thirdly, we impose that continuity condition to the derivatives, in the common end of two consecutive polynomials,

$$\begin{aligned} S'_i(x_i) &= -M_i \frac{(x_i - x_{i+1})^2}{2 \cdot h_i} + M_{i+1} \frac{(x_i - x_i)^2}{2 \cdot h_i} + \frac{y_{i+1} - y_i}{h_i} - \frac{M_{i+1} - M_i}{6} \cdot h_i, \quad i = 0, \dots, n-1 \\ S'_{i-1}(x_i) &= -M_{i-1} \frac{(x_i - x_i)^2}{2 \cdot h_{i-1}} + M_i \frac{(x_i - x_{i-1})^2}{2 \cdot h_{i-1}} + \frac{y_i - y_{i-1}}{h_{i-1}} - \frac{M_i - M_{i-1}}{6} \cdot h_{i-1}, \quad i = 1, \dots, n \\ S'_i(x_i) &= S'_{i-1}(x_i), \quad i = 1, \dots, n-1 \Rightarrow \\ &\Rightarrow -M_i \frac{h_i}{2} + \frac{y_{i+1} - y_i}{h_i} - \frac{M_{i+1} - M_i}{6} \cdot h_i = M_i \frac{h_{i-1}}{2} + \frac{y_i - y_{i-1}}{h_{i-1}} - \frac{M_i - M_{i-1}}{6} \cdot h_{i-1} \end{aligned}$$

Si agrupamos a un lado los valores  $M_{i-1}, M_i, M_{i+1}$ , If we group at one side the values  $M_{i-1}, M_i, M_{i+1}$ ,

$$h_{i-1} \cdot M_{i-1} + 2 \cdot (h_{i-1} + h_i) \cdot M_i + h_i \cdot M_{i+1} = 6 \cdot \left( \frac{y_{i+1} - y_i}{h_i} - \frac{y_i - y_{i-1}}{h_{i-1}} \right)$$

$$i = 1, \dots, n-1$$

En total tenemos  $M_0, \dots, M_n, n+1$  incógnitas y la expresión anterior, solo nos suministra  $n-1$  ecuaciones. Necesitamos dos ecuaciones más. Si imponemos la condición de splines naturales, para el extremo de la izquierda del primer polinomio y para el extremo de la derecha del último,

We have in total  $M_0, \dots, M_n, n+1$  unknowns and the above expression only supply  $n-1$  equations. We need two more equations. If we impose the conditions for a natural spline for the left end of the first polynomial and for the right end of the last one,

$$M_0 = S''(x_0) = 0$$

$$M_n = S''(x_n) = 0$$

Con estas condiciones y la expresión obtenida para el resto de los  $M_i$ , podemos construir un sistema de ecuaciones tridiagonal

With these last conditions and the expression obtained for the remaining  $M_i$ , we can build a tridiagonal system of equations

$$\begin{pmatrix} 2(h_0 + h_1) & h_1 & 0 & 0 & \cdots & 0 & 0 \\ h_1 & 2(h_1 + h_2) & h_2 & 0 & \cdots & 0 & 0 \\ 0 & h_2 & 2(h_2 + h_3) & h_3 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 2(h_{n-3} + h_{n-2}) & h_{n-2} \\ 0 & 0 & 0 & 0 & \cdots & h_{n-2} & 2(h_{n-2} + h_{n-1}) \end{pmatrix} \cdot \begin{pmatrix} M_1 \\ M_2 \\ M_3 \\ \vdots \\ M_{n-1} \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{n-1} \end{pmatrix}$$

Donde hemos hecho,

Where,

$$b_i = 6 \cdot \left( \frac{y_{i+1} - y_i}{h_i} - \frac{y_i - y_{i-1}}{h_{i-1}} \right)$$

Tenemos un sistema de ecuaciones en el que la matriz de coeficientes es tridiagonal y además diagonal dominante, por lo que podríamos emplear cualquiera de los métodos vistos en el capítulo ???. Una vez resuelto el sistema y obtenidos los valores de  $M_i$ , obtenemos los valores de  $A_i$  y  $B_i$  a Partir de las ecuaciones obtenidas más arriba.

Por último, la forma habitual de definir el polinomio de grado 3  $S_i$ , empleado para interpolar los valores del intervalo  $[x_i, x_{i+1}]$ , mediante splines cúbicos se define como,

So, we have a system of equations for which the coefficients matrix is tridiagonal and, besides, it is a dominant diagonal matrix. For this reason, we can solve it with anyone of the methods described in chapter ???. Once the system is solved, we get the values of  $M_i$  and the values of  $A_i$  and  $B_i$  using the equations described above.

A last remark: we usually define the 3-degree polynomial,  $S_i$  used for interpolating the values inside the interval  $[x_i, x_{i+1}]$  as follows,

$$S_i(x) = \alpha_i + \beta_i(x - x_i) + \gamma_i(x - x_i)^2 + \delta_i(x - x_i)^3, \quad x \in [x_i, x_{i+1}], \quad (i = 0, 1, \dots, n-1)$$

Donde,

Where,

$$\begin{aligned}\alpha_i &= y_i \\ \beta_i &= \frac{y_{i+1} - y_i}{h_i} - \frac{M_i \cdot h_i}{3} - \frac{M_{i+1} \cdot h_i}{6} \\ \gamma_i &= \frac{M_i}{2} \\ \delta_i &= \frac{M_{i+1} - M_i}{6 \cdot h_i}\end{aligned}$$

La siguiente función permite obtener los coeficientes y el resultado de interpolar un conjunto de puntos mediante splines cúbicos,

The following Python code computes the coefficients of the cubic spline defined using a dataset and the value taken by the spline at any point.

---

cubic\_spline.py

```

1  # -*- coding: utf-8 -*-
2  """
3  Created on Wed Jul 31 11:01:10 2024
4  This script implements some functions to compute Cubic spline
5  @author: abierto
6  """
7  import numpy as np
8
9  def spcubic(x,y):
10     """
11         Function to calculate the spline coefficients
12         Parameters
13         -----
14         x : TYPE real np array
15             DESCRIPTION. Table Values x to be interpolated
16         y : TYPE real np array
17             DESCRIPTION. Table values y to be interpolated
18         Returns
19         -----
20         h: Type array of differences between x consecutive data
21         M,A,B: arrays with coefficients for the polynomial that
22             that compose the spline expressed as differences
23             (see manual)
24         C : TYPE numpy array size [I,4]
25             I->intervals i.e number of data - 1
26             4->Each 3-degree polynomial has 4 coefficients
27             DESCRIPTION. spline coefficients in standard polynomial representation
28     """
29     l = x.shape[0] #number of data
30     dy = np.zeros(l-1)
31     h = dy.copy()
32
33     for i in range(0,l-1):
34         h[i] = x[i+1] - x[i]
35         dy[i] = y[i+1]-y[i]
```

```

36     CSPp = np.zeros(l-2)
37     b = CSPp.copy()
38     for i in range(l-2):
39         CSPp[i] = 2*(h[i+1]+h[i]) #ppal diagonal
40         b[i] = 6*(dy[i+1]/h[i+1]-dy[i]/h[i])
41     #building the system matrix (a pro. would use a sparse matrix here)
42     CSP = np.diag(CSPp)+np.diag(h[1:-1],1)+np.diag(h[1:-1],-1)
43     M = np.linalg.solve(CSP,b)
44     M = np.insert(M,0,0)
45     M= np.append(M,0)
46     A = np.zeros(l-1)
47     B = np.zeros(l-1)
48     for i in range(l-1):
49         A[i] = dy[i]/h[i] - (M[i+1]-M[i])*h[i]/6
50         B[i]=y[i]-M[i]*h[i]**2/6
51     C = np.zeros([l-1,4])
52     for i in range(l-1):
53         C[i,0]=y[i];
54         C[i,1]=dy[i]/h[i]-M[i]*h[i]/3-M[i+1]*h[i]/6;
55         C[i,2]=M[i]/2;
56         C[i,3]=(M[i+1]-M[i])/(6*h[i]);
57     return M,A,B,C,h
58
59 def evspc(M,A,B,x,xi):
60     """
61     This function calculates the value of a interpolating spline in point xi
62
63     Parameters
64     -----
65     M : TYPE numpy array
66         DESCRIPTION. Coefficients M of the spline
67     A : TYPE numpy array cofficients A of the spline
68         DESCRIPTION.
69     B : TYPE
70         DESCRIPTION.
71     x : TYPE numpy array
72         DESCRIPTION. x data of the datatable interpolated
73     xi : TYPE double
74         DESCRIPTION. point to calculate the spline value
75
76     Returns
77     -----
78     yi : TYPE double
79         DESCRIPTION. value calculate for the spline yi =s(xi)
80
81     """
82     l = x.shape[0]
83     h = np.zeros(l-1)
84     for i in range(0,l-1):
85         h[i] = x[i+1] - x[i]
86     j = 0
87     while xi > x[j]:

```

```

88         j = j+1
89     if j > l - 1:
90         j = l - 1
91     elif j < 1:
92         j = 1
93     yi=-M[j-1]*(xi-x[j])**3/(6*h[j-1])+ M[j]*(xi-x[j-1])**3/(6*h[j-1])\
94         +A[j-1]*(xi-x[j-1])+B[j-1]
95     return yi

```

---

La figura, 8.5 muestra el resultado de interpolar mediante un spline cúbico, los datos contenidos en la figura 8.3. Es fácil observar cómo ahora los polinomios de interpolación dan como resultado una curva suave en los datos interpolados y en la que además las curvas son también suaves, sin presentar variaciones extrañas, para los puntos contenidos en cada intervalo entre dos datos.

### 8.3.2. Funciones propias de Python para interpolación por intervalos

Para realizar una interpolación por intervalos mediante cualquiera de los procedimientos descritos, podemos emplear el subpaquete de `scipy Scipy.interpolate`. Este paquete incorpora múltiples métodos de interpolación; solo veremos dos:

La función `interp1d`. Esta función admite como variables de entrada dos arrays con los valores de las coordenadas  $x$  e  $y$  de los datos que se desea interpolar. Además, admite como variable de entrada una cadena de caracteres que indica el método con el que se quiere realizar la interpolación. Dicha variable puede tomar los valores:

1. '`nearest`'. Interpola el intervalo empleando el valor  $y_i$  correspondiente al valor  $x_i$  más cercano al punto que se quiere interpolar. El resultado es una interpolación a escalones. del conjunto de datos que se desea interpolar.
2. '`next`'. Interpola el intervalo empleando el valor de  $y_i$  correspondiente al punto  $x_i$  de los datos que cierra el intervalo.
3. '`previous`' Interpola el intervalo empleando el valor de  $y_i$  correspondiente

Figure 8.5 shows the result of interpolating the same data of figure 8.3 using a cubic spline. It is easy to check how the interpolation polynomials which compose the spline cast a smooth result in the points interpolated and how the curves shapes are reasonable smooth also in the intervals between the data. They do not present variations difficult to explain using the data.

### 8.3.2. Python functions for piecewise interpolation

To carry out a piecewise interpolation using anyone of the method above described, we can use the `scipy` sub-package `scipy.interpolate`. This package includes many interpolation methods; we will only describe two of them:

The `interp1d` function. This function takes two arrays as input variables with the coordinates  $x$  and  $y$  values belonging to the data we want to interpolate. In addition, the function may take also a character string, as an input variable which defines the method we want to use to carry out the interpolation. This last variable can take the following values:

1. '`nearest`'. It interpolates the interval using the value  $y_i$  corresponding to the value  $x_i$  nearest to the point at which we want to compute the interpolation. The result is stepwise interpolation.
2. '`next`'. It interpolates the interval using the value  $y_i$  corresponding to the value  $x_i$  which closes the interval in which we want to compute the interpolation.
3. '`previous`' It interpolates the interval using the value  $y_i$  corresponding to the

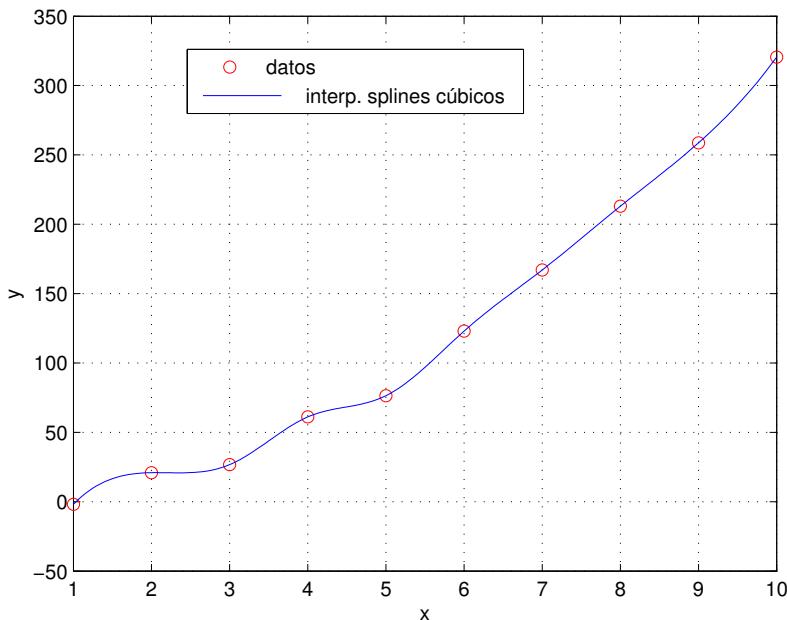


Figura 8.5: Interpolación mediante spline cúbico de los datos de la figura 8.3

al punto  $x_i$  de los datos que abre el intervalo.

4. 'linear' realiza una interpolación lineal entre los extremos del intervalo que se desea intepolar. Esta es la opción por defecto.

**interp1d** Devuelve una función que admite como variable de entrada un array con los puntos para para los que se quiere calcular el valor de la interpolación y devuelve como salida un array con los resultados obtenidos. El siguiente código muestra el modo de usar el comando **interp1d**. Para probarlo se han creado dos array **x** e **y** que contienen el conjunto de datos que se empleará para calcular la interpolación. Además, se ha creado otro vector **xi** que contiene los puntos para los que se quiere calcular el resultado de la interpolación.

value  $x_i$  which opens the interval in which we want to compute the interpolation.

4. 'linear' It calculates a linear interpolation between the points at the ends of the interval we want to interpolate. This is the default option.

**interp1d** returns a function that, in turn, takes as input variable an array with the points we want to calculate the interpolation at, and returns an array with the computed results. The following code shows how to use the function **interp1d**. To try it we have created two arrays **x** and **y** which contain the dataset we want to interpolate. Besides, we have created another array **xi** with the point we want to compute the interpolation at.

---

ejemplo\_interp1d.py —

```
1 # -*- coding: utf-8 -*-
```

```

2 """
3 Created on Fri Aug  2 16:55:04 2024
4 Ejemplo de uso de la función interp1d de Scipy
5 @author: abierto
6 """
7 import numpy as np
8 from scipy.interpolate import interp1d
9 from matplotlib import pyplot as pl
10 x = np.array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
11 y = np.array([-1.8143451, 20.914356, 26.714303, 61.129501,\n
12             76.414728, 123.00032, 167.06809, 212.97832,\n
13             258.67911, 320.53422])
14 #dibujamos los puntos
15 pl.plot(x,y,'o')
16 #Creamos unos puntos sobre los que interpolar
17 xi = np.arange(1.,10.,0.01)
18 #Creamos la función interpoladora empleando interp1d
19 flin = interp1d(x,y,'linear')#puede omitirse linear el la opcion por defecto
20 #empleamos la funcion para obtener los valores de los puntos interpolados
21 yi = flin(xi)
22 pl.plot(xi,yi)
23
24 fnear = interp1d(x,y,'nearest')
25 yi = fnear(xi)
26 pl.plot(xi,yi)
27
28 fnext = interp1d(x,y,'next')
29 yi = fnext(xi)
30 pl.plot(xi,yi)
31 #and so on and so forth...

```

---

La segunda función interpoladora de `scipy.interpolate` que vamos a describir se llama `CubicSpline`. Permite interpolar empleando Splines cúbicos. En este caso, además de los datos `x` e `y` necesarios para definir el spline, la función admite el parámetro `bc_type`, que permite determinar las condiciones de contorno en los extremos del spline. Las opciones posibles son:

`bc_type='not_knot'`. Es la opción por defecto. Hace que el polinomio empleado en el primer y segundo segmento en los extremos del spline sea el mismo.

`bc_type='natural'`. Spline natural.

`bc_type='periodic'`. Spline periódico.

Hay más opciones; los interesados pueden encontrarlas en la documentación de `scipy`. El uso es similar al descrito para `interp1d`. A partir de los datos se crea una función inter-

The second interpolating function, from `scipy.interpolate` we are going to describe is `CubicSpline`. This function, as can be expected, uses cubic splines to perform the interpolation of a dataset. In this cases, in addition to the `x` and `y` data needed to define the spline, the function defines an input parameter `bc_type`, which allows us to establish the boundary conditions in the ends of the spline. Some of the available options are:

`bc_type='not_knot'`. This is the default option. The first and second segments at an end of the spline uses the same polynomial.

`bc_type='natural'`. Natural spline.

`bc_type='periodic'`. Periodic spline.

There are more options. Those interested can find them in `scipy` documentation. Its use is similar to the use of `interp1d`. From the data we want to interpolate `CubicSpline` genera-

poladora que permite calcular el valor de la imagen en los puntos que se deseé. Por ejemplo, para splines naturales sería,

```
spnat = CubicSpline(x,y,'natural')
yi = spnat(xi)
```

## 8.4. Ajuste polinómico por el método de mínimos cuadrados

Los métodos de interpolación que hemos descrito hasta ahora, pretenden encontrar un polinomio o una función definida a partir de polinomios que pase por un conjunto de datos. En el caso del ajuste por mínimos cuadrados, lo que se pretende es buscar el polinomio, de un grado dado, que mejor se aproxime a un conjunto de datos.

Supongamos que tenemos un conjunto de  $m$  datos,

x	$x_1$	$x_2$	...	$x_m$
y	$y_1$	$y_2$	...	$y_m$

Queremos construir un polinomio  $p(x)$  de grado  $n < m - 1$ , de modo que los valores que toma el polinomio para los datos  $p(x_i)$  sean lo más cercanos posibles a los correspondientes valores  $y_i$ .

En primer lugar, necesitamos clarificar qué entendemos por *lo más cercano posible*. Una posibilidad, es medir la diferencia,  $y_i - p(x_i)$  para cada par de datos del conjunto. Sin embargo, es más frecuente emplear el cuadrado de dicha diferencia,  $(y_i - p(x_i))^2$ . Esta cantidad tiene, entre otras, la ventaja de que su valor es siempre positivo con independencia de que la diferencia sea positiva o negativa. Además, representa el cuadrado de la distancia entre  $p(x_i)$  e  $y_i$ . Podemos tomar la suma de dichas distancias al cuadrado, obtenidas por el polinomio para todos los pares de puntos,

$$\sum_{i=1}^m (y_i - p(x_i))^2$$

tes an interpolating function which allows us to compute the spline value in the point we wish. For instance, for natural splines we write,

## 8.4. Least squared error method for polynomial data fitting

The interpolation methods we have described so far, try to find a polynomial of a function defined using polynomials that passes through a set of data. In the case of Least squared error fitting, the aim is to find the polynomial of a specific degree that better approximates a dataset.

Suppose we have a dataset holding  $m$  data,

We want to build a polynomial  $p(x)$  with degree  $n < m - 1$  so that the values it takes for the data  $p(x_i)$  are the nearest possible to the values  $y_i$  of the data table.

First, we must clarify what means *the nearest possible*. One option is to measure the difference  $y_i - p(x_i)$  for each pair of data in the dataset. However, using the square of such difference  $(y_i - p(x_i))^2$  is more common. This quantity has the advantage, among others, of being always positive, no matter whether the difference is positive or negative. Besides, it represents the square distance between  $p(x_i)$  and  $y_i$ . We can take the sum of these differences for all pairs in the dataset,

como una medida de la distancia del polinomio a los datos. De este modo, el polinomio *lo más cercano posible* a los datos sería aquel que minimice la suma de diferencias al cuadrado que acabamos de definir. De ahí el nombre del método.

En muchos casos, los datos a los que se pretende ajustar un polinomio por mínimos cuadrados son datos experimentales. En función del entorno experimental y del método con que se han adquirido los datos, puede resultar que algunos resulten más fiables que otros. En este caso, sería deseable hacer que el polinomio se aproxime más a los datos más fiables. Una forma de hacerlo es añadir unos *pesos*,  $\omega_i$ , a las diferencias al cuadrado en función de la confianza que nos merece cada dato,

$$\sum_{i=1}^m \omega_i (y_i - p(x_i))^2$$

Los datos fiables se multiplican por valores de  $\omega$  grandes y los poco fiables por valores pequeños.

Para ver cómo obtener los coeficientes de un polinomio de mínimos cuadrados, empezaremos con el caso más sencillo; un polinomio de grado 0. En este caso, el polinomio es una constante, definida por su término independiente  $p(x) = a_0$ . El objetivo a minimizar sería entonces,

$$g(a_0) = \sum_{i=1}^m \omega_i (y_i - a_0)^2$$

El valor mínimo de esta función debe cumplir que su derivada primera  $g'(a_0) = 0$  y que su derivada segunda  $g''(a_0) \geq 0$ ,

$$g'(a_0) = -2 \sum_{i=1}^m \omega_i (y_i - a_0) = 0 \Rightarrow a_0 = \frac{\sum_{i=1}^m \omega_i \cdot y_i}{\sum_{i=1}^m \omega_i}$$

$$g''(a_0) = 2 \sum_{i=1}^m \omega_i \Rightarrow g''(a_0) \geq 0$$

El resultado obtenido para el valor de  $a_0$  es una media, ponderada con los pesos  $w_i$  de los datos. Si hacemos  $w_i = 1 \forall w_i$  obtendríamos exactamente la media de los datos. Este resul-

as a measure of the distance from the polynomial to the data. In this way, the *nearest possible to the data* would be the polynomial that minimised the sum of square differences we have just defined. Indeed, this is the origin of the method's name.

The data we often want to fit a polynomial using the least square method are experimental data. Depending on the environment and the method used for data acquisition, some data may be more reliable than others. In this case, it is interesting that the polynomial passes closest to the more reliable data. One way to achieve it is to add some *weights*,  $\omega_i$  to the square differences according to the confidence that each data deserves,

We will multiply more reliable data by large values of  $\omega$  and the less reliable by small values.

To see how to obtain the coefficients of a least square polynomial, we will start with the simplest case; a 0-degree polynomial. In this case the polynomial is a simple constant value, defined by its constant term  $p(x) = a_0$ . Then the function to be minimised would be,

The minimum of this function must meet that its first derivative  $g'(a_0) = 0$  and its second derivative  $g''(a_0) \geq 0$ ,

We obtain that the value for  $a_0$  is a mean of the data, weighted by the values  $\omega_i$  assigned to the data. If we take  $w_i = 1 \forall w_i$ , we obtain the mean value of the data. This re-

tado resulta bastante razonable. Aproximar un conjunto de valores por un polinomio de grado cero, es tanto como suponer que la variable  $y$  permanece constante para cualquier valor de  $x$ . Las diferencias observadas deberían deberse entonces a errores aleatorios experimentales, y la mejor estima del valor de  $y$  será precisamente el valor medio de los valores disponibles. La figura 8.6 muestra el resultado de calcular el polinomio de mínimos cuadrados de grado cero para un conjunto de datos.

sult is quite reasonable. Approximating a set of values with a 0-degree polynomial is equivalent to considering that variable  $y$  remains constant for every value of  $x$ . In this case, we may attribute the observed differences among the values of the table to experimental random errors. Consequently, the best estimation of  $y$  would be the mean value of the available data.

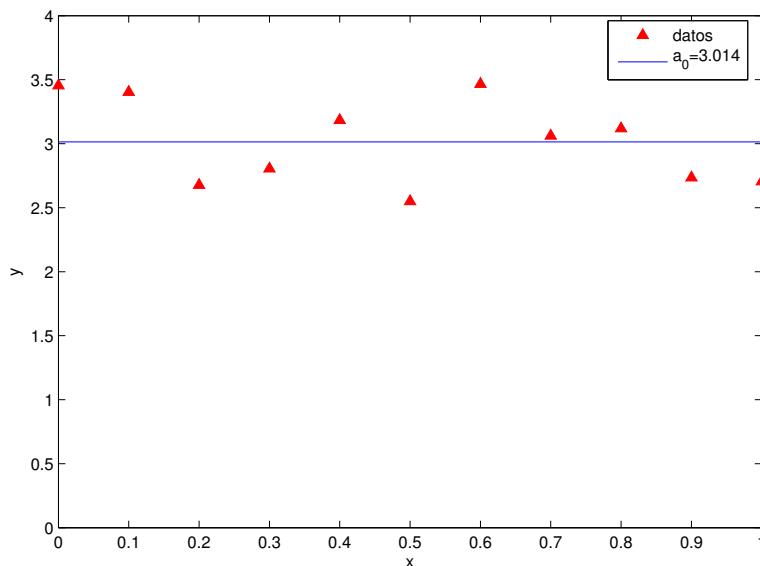


Figura 8.6: Polinomio de mínimos cuadrados de grado 0  
Figure 8.6: 0-degree least squared error polynomial

El siguiente paso en dificultad sería tratar de aproximar un conjunto de datos por un polinomio de grado 1, es decir, por una linea recta,  $p(x) = a_0 + a_1x$ . En este caso, la suma de diferencias al cuadrado toma la forma,

The next step would be to approximate the data using a 1-degree polynomial, i.e., a straight line,  $p(x) = a_0 + a_1x$ . In this case, the sum of squared differences takes the form,

$$g(a_0, a_1) = \sum_{i=1}^m \omega_i (y_i - a_0 - a_1 x_i)^2$$

En este caso, tenemos dos coeficientes sobre los que calcular el mínimo. Éste se obtiene cuando las derivadas parciales de  $g(a_0, a_1)$  respecto a ambos coeficientes son iguales a cero.

We have now two coefficient for computing the minimum. We get the minimum making the function  $g(a_0, a_1)$  partial derivatives equal to zero.

$$\begin{aligned}\frac{\partial g}{\partial a_0} &= -2 \sum_{i=1}^m \omega_i (y_i - a_0 - a_1 x_i) = 0 \\ \frac{\partial g}{\partial a_1} &= -2 \sum_{i=1}^m \omega_i x_i (y_i - a_0 - a_1 x_i) = 0\end{aligned}$$

Si reordenamos las ecuaciones anteriores,

And after rearranging the previous equations,

$$\begin{aligned} \left( \sum_{i=1}^m \omega_i \right) a_0 + \left( \sum_{i=1}^m \omega_i x_i \right) a_1 &= \sum_{i=1}^m \omega_i y_i \\ \left( \sum_{i=1}^m \omega_i x_i \right) a_0 + \left( \sum_{i=1}^m \omega_i x_i^2 \right) a_1 &= \sum_{i=1}^m \omega_i x_i y_i \end{aligned}$$

Obtenemos un sistema de dos ecuaciones lineales cuyas incógnitas son precisamente los coeficientes de la recta de mínimos cuadrados.

Podemos ahora generalizar el resultado para un polinomio de grado  $n$ ,  $p(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_n x^n$ . La función  $g$  toma la forma,

We get a linear system with two equations and two unknowns, which are the coefficients of the least squares line.

We can now generalised this result for a  $n$ -degree polynomial  $p(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_n x^n$ . Function  $g$  would be now,

$$g(a_0, a_1, \dots, a_n) = \sum_{i=1}^m \omega_i (a_0 + a_1 x_i + \dots + a_n x_i^n - y_i)^2$$

De nuevo, para obtener los coeficientes del polinomio igualamos las derivadas parciales a cero,

Again, we can get the polynomial coefficients making the partial derivatives equal to zero,

$$\frac{\partial g(a_0, a_1, \dots, a_n)}{\partial a_j} = 0 \Rightarrow \sum_{i=1}^m \omega_i x_i^j (a_0 + a_1 x_i + \dots + a_n x_i^n - y_i) = 0, \quad j = 0, 1, \dots, n$$

Si reordenamos las expresiones anteriores, llegamos a un sistema de  $n+1$  ecuaciones lineales, cuyas incógnitas son los coeficientes del polinomio de mínimos cuadrados,

If we rearrange the previous equation, we arrive to a linear system of  $n+1$  equations whose unknowns are the coefficients of the least squares polynomial,

$$\begin{pmatrix} s_0 & s_1 & \dots & s_n \\ s_1 & s_2 & \dots & s_{n+1} \\ \vdots & \vdots & \ddots & \\ s_n & s_{n+1} & \dots & s_{2n} \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{pmatrix}$$

Donde hemos definido  $s_j$  y  $c_j$  como,

Where we have defined  $s_j$  and  $c_j$  as,

$$\begin{aligned} s_j &= \sum_{i=1}^m \omega_i x_i^j \\ c_j &= \sum_{i=1}^m \omega_i x_i^j y_i \end{aligned}$$

El siguiente código permite obtener el polinomio de mínimos cuadrados que aproxima un conjunto de  $n$  datos,

The following code calculates the coefficients of the least squares polynomial that approximate a set of  $n$  data.

---

minimos\_cuadrados.py

---

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Sat Aug 3 20:15:18 2024
4 Function to obtain the least squared error polynomial from a dataset
5 @author: abierto
6 """
7 import numpy as np
8 def lse(x,y,n,w='none'):
9     """
10     This function compute the least squared error polynomial of
11     degree n from a set of data x,y. Optionally it can use a
12     set of weights to perform the computation
13     Parameters
14     -----
15     x : TYPE numpy array
16         DESCRIPTION. x data to be fitted
17     y : TYPE numpy array
18         DESCRIPTION. y data to be fitted
19     n : TYPE integer
20         DESCRIPTION. Polynomial degree
21     w : TYPE numpy array with the same length than x and y
22         DESCRIPTION.
23
24     Returns
25     -----
26     a: TYPE numpy array
27         DESCRIPTION. polynomial coefficients in increasing powers
28         a[0]+a[1]x+a[2]x**2+...+a[n]x**n
29
30     """
31     m = x.shape[0]
32     #first we check that we have enough points
33     if m < n:
34         raise Warning('the degree is greater than the number of data')
35     #if there is no weight array generate an arry of ones
36     if type(w)==str:
37         w = np.ones(m)
38     #we build the s elements of the system coefficient matrix
39     n = n+1 # for degree n I need n+1 coeficientes
40     s = np.zeros(2*n)
41     for j in range(2*n):
42         for i in range(m):
43             s[j] = s[j] + w[i]*x[i]**j
44     c = np.zeros(n)
45     for j in range(n):
46         for i in range(m):
47             c[j] = c[j] + w[i]*x[i]**j*y[i]
48     A = np.zeros([n,n])
49     for i in range(n):
50         for j in range(n):
51             A[i,j] = s[i+j]
```

---

```

52     a =np.linalg.solve(A,c)
53     return(a)
54

```

---

Una última observación importante es que si intentamos calcular el polinomio de mínimos cuadrados de grado  $m - 1$  que aproxima un conjunto de  $m$  datos, lo que obtendremos será el polinomio de interpolación. En general, cuanto mayor sea el grado del polinomio más posibilidades hay de que la matriz de sistema empleado para obtener los coeficientes del polinomio esté mal condicionada.

#### 8.4.1. Mínimos cuadrados en Python.

Hay varias formas de llevar a cabo un ajuste polinómico por mínimos cuadrados empleando Python. Solo vamos a describir uno de ellos. El subpaquete de `numpy`, `numpy.polynomial` incluye una clase llamada `Polynomial` (con mayúscula) permite, entre otras cosas, crear polinomios como objetos de programación. Una de las formas de crear un polinomio, es como resultado del ajuste por mínimos cuadrados a un conjunto de datos. Para ello, `Polynomial`, cuenta con la función `fit`. Este comando admite como entradas un vector de coordenadas  $x$  y otro de coordenadas  $y$ , de los datos que se quieren aproximar, y una tercera variable con el grado del polinomio. Opcionalmente se puede añadir un vector de pesos, si se quiere ponderar la importancia que damos a cada dato.

La función `Polynomial.fit` devuelve como resultado un 'objeto' polinomio. Este objeto contiene toda la información sobre el polinomio creado y puede usarse directamente como una función para calcular el valor del polinomio en un punto. El siguiente script de Python crea un par de vectores y muestra como manejar el comando,

A last important remark: If we try to compute the least squared error polynomial of degree  $m - 1$  to approximate a set of  $m$  data, we will get the interpolating polynomial. In general, if we increase the degree of the least squared polynomial, we increase the probability that the system matrix we use to obtain the polynomial coefficients will be poorly conditioned.

#### 8.4.1. Least squares in Python.

There are several ways to compute the Least squares polynomial for a set of data using Python. We will describe only one of them. The `numpy` sub-package `numpy.polynomial` includes a class called `Polynomial` (Mean the first capital letter!) allows us, among many other things, to create polynomial as programming objects. One way to create a polynomial is to compute the least squares polynomial from a dataset. The class `Polynomial` has a special function `fit` to accomplish it. This command takes as inputs two arrays of data, one with the coordinates  $x$  and the other with the coordinates  $y$  of the value we want to interpolate, plus a third variable for the polynomial degree. Optionally, we can add an array of weights to balance the importance we give to each data.

Function `Polynomial.fit` returns a polynomial 'object'. This object contains all the information on the polynomial it represents and it can be used straightforwardly as a function to compute the value of the polynomial in any point. The following Python's script generates a pair of vectors and shows how to deal with the `fit` command.

---

```

ejemplo.ajuste.py
1  # -*- coding: utf-8 -*-
2  """
3  Created on Sun Aug  4 16:00:12 2024
4  Example of least squares fit using the numpy.polynomial class Polynomial
5  @author: abierto

```

---

## 380CAPÍTULO/CHAPTER 8. INTERP. Y AJUST. FUNCIONES. \*INTERP. &amp; FUNCT FITTING

```
6  """
7  import numpy as np
8  from numpy.polynomial import Polynomial as py
9  from matplotlib import pyplot as pl
10 #we import the function included above to compare the result
11 #with the numpy.polynomial function
12 from minimos_cuadrados import lse
13
14 #define a data set to play with
15 x = np.array([1., 2., 3., 4., 5., 6., 7., 8., 9., 10.])
16 y = np.array([-1.8143451, 20.914356, 26.714303, 61.129501,\n              350.414728, 123.00032, 167.06809, 212.97832,\n              258.67911, 320.53422])
17
18
19
20 p3 = py.fit(x,y,3) #we are not including weight
21 #we get the polynomial coefficients just to see them
22 a = p3.convert().coef
23
24 #a set of point to evaluate the polynomial
25 xi = np.arange(1.,10.,0.01)
26
27 yi = p3(xi)
28
29 #point y[4] (see graphics bellow) apears to be an outlayer we can
30 #use weights to attenuate its influence
31 w = np.ones(x.shape[0])
32 w[4] = 0.5 #we reduce the weight of the outlayer
33 p3w = py.fit(x,y,3,w=w)
34 yiw = p3w(xi)
35
36 #####
37 #we repeate the calculation using our own funtion lse
38 #####
39 an = lse(x,y,3)
40 p = py(an) #we create a polynomial object using the coeficients
41 yin = p(xi)
42 #repeat the computing using the weight
43 aw = lse(x,y,3,w)
44 pw = py(aw)
45 yinw = pw(xi)
46
47 #drawing the results to compare
48 ax1 = pl.subplot(2,1,1)
49 pl.plot(x,y,'o')
50 pl.plot(xi,yi)
51 pl.plot(xi,yiw)
52 ax1.legend(['Data','Polynomial.fit',\n            'Polynomial.fit (weights)'])
53 ax2 = pl.subplot(2,1,2)
54 pl.plot(x,y,'o')
55 pl.plot(xi,yin)
56 pl.plot(xi,yinw)
```

```
58 ax2.legend(['Data', 'lse', 'lse (weights)'])
```

Vamos a revisar algunos puntos importantes de este código. En la línea 8, importamos el objeto `Polynomial`. Lo hacemos siguiendo el mismo método con que habríamos importado un submódulo cualquiera de `numpy`. Le asignamos el alias `py`. En la línea 12 hemos importado la función `lse`. Se trata de la función que hemos creado nosotros para realizar ajustes por mínimos cuadrados y cuyo código hemos incluido mas arriba. En las líneas 15 y 16 creamos un conjunto de datos.

En la línea 20 creamos un polinomio de mínimos cuadrados de grado 3, `p3`, que ajusta los datos anteriores. `p3`, contiene toda la información del polinomio creado. En la línea 22, empleamos el comando `convert` para obtener un array con los coeficientes del polinomio, ordenados de menor a mayor grado.  $p(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$ . En la línea 27 empleamos directamente `p3` como una función, pasándole los valores del array `xi`, para evaluar el polinomio en dichos puntos.

En las líneas 29-34 repetimos el cálculo, pero ahora empleando pesos. Damos a todos los pesos un valor 1 excepto para el quinto dato, al que damos un peso menor.

Las líneas 39 a 45 repiten los mismos cálculos, pero ahora empleando la función `lse`, creada por nosotros.

Por último, dibujamos los resultados para compararlos, tal y como se muestran en la figura 8.7

La figura muestra claramente el efecto de los pesos en la regresión. Al atenuar la influencia del quinto punto, que claramente parece fuera de rango, la curva se acerca más al resto de los puntos.

Si miramos con atención ambos gráficos, es fácil darse cuenta que la función de Python acerca más el polinomio a los puntos que la que hemos escrito nosotros. La razón está en que hemos usado un modo distinto de introducir los pesos. La función `fit`, aplica el peso directamente a la diferencia entre el dato y el valor del polinomio  $\omega_i|y_i - p(x_i)|$  mientras que nosotros aplicamos los pesos al cuadrado de dicha diferencia  $\omega_i(y_i - p(x_i))^2$ . Si compa-

We are going to review some important points of the previous code. In line 8, we import the object `Polynomial`. We do it in the same way that we import whatever `numpy` submodule. We have assigned to `Polynomial` the alias `py`. In line 12 we import the function `lse`. This is the function we previously created to perform least squares fits and whose code we included above. In lines 15 and 16 we create a dataset.

In line 20, we create a 3-degree least squares polynomial, `p3`, which fits the previous data. `p3` holds all the information on the polynomial we have created. In line 22, we use the command `convert` to obtain an array with the polynomial coefficients, ordered in ascending degree.  $p(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$ . In line 27, we use `p3` straightforwardly, as a function, to obtain the values the polynomial takes at the points held by the array `x_i`.

We repeat the same computation in lines 29-34 but now use weights. We assign a value of 1 to every weight except for the fifth data point, to which we assign a smaller weight.

In lines 39 to 45 we repeat the the same computation but this time using our own function, `lse`

Lastly, we draw the results to compare, as can be seen in figure 8.7.

The figure clearly shows the effect the weights have on the regression results. When we attenuate the influence of the fifth point, which appears to be quite an outlier, the curve approaches more to the remaining points.

If we look at both graphics attentively, we will see that the Python function approximates the polynomial more to the points than the function we have written. The difference comes from how we introduce the weights, which are not the same as those used by Python. Function `fit`, applies the weights directly to the difference between the data an the value taken by the polynomial  $\omega_i|y_i - p(x_i)|$ , while our function applies the weights to the squa-

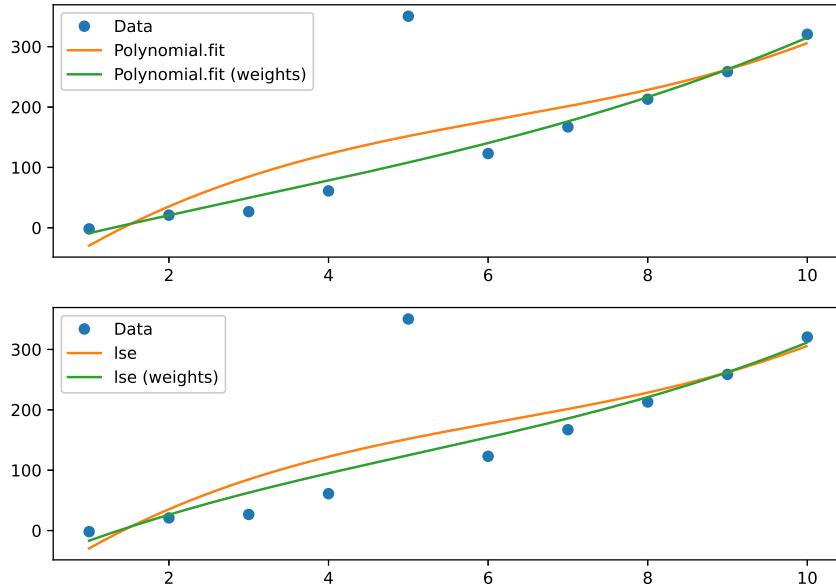


Figura 8.7: Ejemplo de cálculo de ajuste por mínimos cuadrados empleando la función fit de Polynomial y nuestra función lse.

Figure 8.7: An example of least squares polynomial fit, using the function fit from Polynomial and our function lse.

ramos los coeficientes de los polinomios obtenidos,

red difference  $\omega_i(y_i - p(x_i))^2$ . If we compare the coefficients of the polynomials computed by both methods,

```
In [47]: p3w.convert().coef
Out[47]: array([-40.71774313,  33.22583298, -1.61930914,  0.1854822])
```

```
In [48]: aw
Out[48]: array([-68.26797522,  56.50545799, -5.31193931,  0.34603134])
```

vemos que no coinciden. Para obtener el mismo resultado, deberíamos introducir el cuadrado del valor de los pesos cuando usamos nuestra función,

We can see they are different. To get the same result, we must introduce the squared values of our weight when we use our function,

```
In [49]: aw2 = lse(x,y,3,w**2)
```

```
In [50]: aw2
Out[50]: array([-40.71774313,  33.22583298, -1.61930914,  0.1854822])
```

Esto nos enseña una lección muy importante: no se deben emplear nunca paquetes de cálculo numérico sin saber qué están calculando exactamente.

#### 8.4.2. Análisis de la bondad de un ajuste por mínimos cuadrados.

Supongamos que tenemos un conjunto de datos obtenidos como resultado de un experimento. En muchos casos la finalidad de un ajuste por mínimos cuadrados, es encontrar una ley que nos permita relacionar los datos de la variable independiente con la variable dependiente. Por ejemplo si aplicamos distintas fuerzas a un muelle y medimos la elongación sufrida por el muelle, esperamos obtener, en primera aproximación, una relación lineal:  $\Delta x \propto F$ . (Ley de Hooke).

Sin embargo, los resultados de un experimento rara vez se ajustan a una ley debido a errores aleatorios que no es posible corregir.

Cuando realizamos un ajuste por mínimos cuadrados de  $m$  datos, podemos emplear cualquier polinomio desde grado 0 hasta grado  $m-1$ . Desde el punto de vista del error cometido con respecto a los datos disponibles el mejor polinomio sería precisamente el de grado  $m-1$  que da error cero para todos los datos, por tratarse del polinomio de interpolación. Sin embargo, si los datos son experimentales estamos incluyendo los errores experimentales en el ajuste.

Por ello, para datos experimentales y suponiendo que los datos solo contienen errores aleatorios, el mejor ajuste lo dará el polinomio de menor grado para el cual las diferencias entre los datos y el polinomio  $y_i - p(x_i)$  se distribuyan aleatoriamente. Estas diferencias reciben habitualmente el nombre de residuos.

La figura 8.8 muestra un ejemplo de ajuste por mínimos cuadrados empleando cada vez un polinomio de mayor grado.

En la figura 8.8(a) se observa claramente que el ajusto no es bueno, la recta de mínimos cuadrados no es capaz de adaptarse a la forma que presentan los datos. Los residuos

This result teaches us a fundamental lesson: never use a numerical computing software package without understanding its operations.

#### 8.4.2. Analyzing goodness of fit using least squares method.

Suppose we have a dataset we have obtained as a result of an experiment. In many cases, least squares data fitting aims to find a law that allows us to establish a mathematical relationship between the independent and dependent variables. For instance, if we apply different stresses to a spring and measure the elongation acquired by the spring, we expect to get a linear relationship between stress and elongation as a first approximation:  $\Delta x \propto F$ . (Hooke's law).

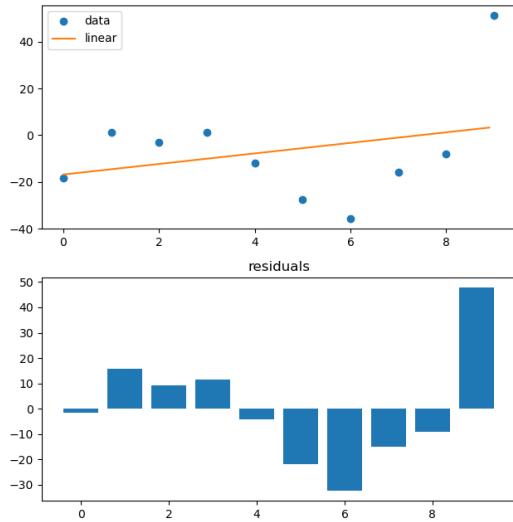
However, the results of an experiment hardly ever fit precisely into any law due to random errors, which we cannot amend.

When we carry out a least squares fitting, we may use any polynomial of degrees 0 to  $m-1$ . From the point of view of the error we make, considering only the available data, the best polynomial would be the  $m-1$ -degree polynomial because this is the interpolation polynomial, and, so, it makes a zero error for every data. Nevertheless, if we are trying to fit experimental data, we are including the experimental errors in the fitting.

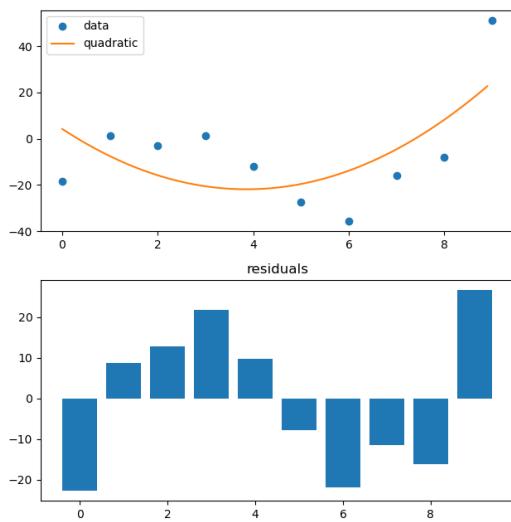
As a consequence, for experimental data and assuming that the data only are affected by random errors, we get the best fitting using the polynomial of least degree for which the differences between data and polynomial value  $y_i - p(x_i)$  are randomly distributed. Such differences are usually called residuals.

Figure 8.8 shows a least square fitting example, using polynomials of increasing degrees.

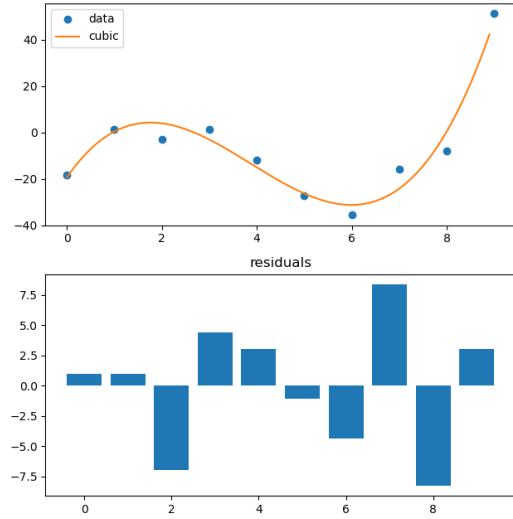
The residual also shows the data shape; they are not randomly distributed. In figure 8.8(b), we can see how a parabola better approximates the data but cannot follow the data up and down. The residuals clearly show



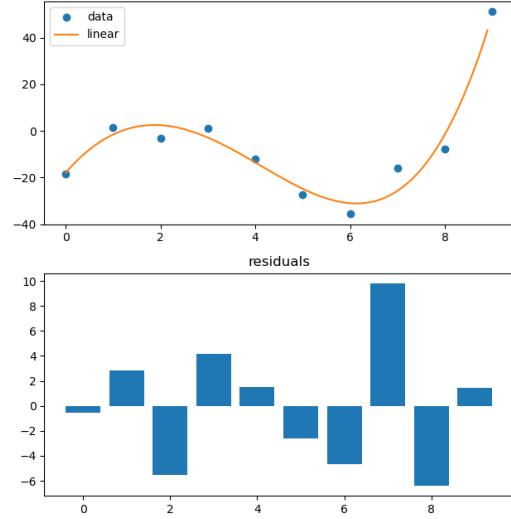
(a) Recta de mínimos cuadrados y residuos obtenidos / Least squares line and residuals.



(b) Parábola de mínimos cuadrados y residuos obtenidos / Least squares parabola and residuals.



(c) polinomio de tercer grado de mínimos cuadrados y residuos obtenidos / 3-degree polynomial and residuals

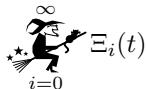


(d) polinomio de cuarto grado de mínimos cuadrados y residuos obtenidos / 4-degree polynomial and residuals

Figura 8.8: Comparación entre los residuos obtenidos para los ajustes de mínimos cuadrados de un conjunto de datos empleando polinomios de grados 1 a 4.

Figure 8.8: Comparison among the residuals resulting from a dataset least squares fitting, using polynomials of degrees 1 to 4.

muestran claramente esta tendencia: no están distribuidos de forma aleatoria. En la figura 8.8(b), la parábola aproxima mejor el conjunto de datos. Sin embargo, los residuos presentan un máximo y un mínimo suaves, no son realmente aleatorio. En la figura 8.8(c), los residuos están distribuidos de forma aleatoria. Si comparamos estos resultados con los de la figura 8.8(d) vemos que en este último caso los residuos son más pequeños, pero conservan esencialmente la misma distribución aleatoria que en la figura anterior. La aproximación de los datos empleando un polinomio de cuarto grado no añade información sobre la forma de la función que siguen los datos, y ha empezado a incluir en el ajuste los errores de los datos.



## 8.5. Curvas de Bézier

Las curvas de Bézier permiten unir puntos mediante curvas de forma arbitraria. Una curva de Bézier viene definida por un conjunto de  $n + 1$  puntos,  $\{\vec{p}_0, \vec{p}_2, \dots, \vec{p}_n\}$ , que reciben el nombre de puntos de control. El orden en que se presentan los puntos de control es importante para la definición de la curva de Bézier correspondiente. Ésta pasa siempre por los puntos  $\vec{p}_0$  y  $\vec{p}_n$ . El resto de los puntos de control permiten dar forma a la curva, que tiene siempre la propiedad de ser tangente en el punto  $\vec{p}_0$  a la recta que une  $\vec{p}_0$  con  $\vec{p}_1$  y tangente en el punto  $\vec{p}_n$  a la recta que une  $\vec{p}_{n-1}$  con  $\vec{p}_n$ .

Para definir la curva, se asocia a cada punto de control  $\vec{p}_i$  una función conocida con el nombre de función de fusión. En el caso de las curvas de Bézier, las funciones de fusión empleadas son los polinomios de Bernstein,

$$B_i^n(t) = \binom{n}{i} (1-t)^{n-i} t^i, \quad i = 0, 1, \dots, n$$

El grado de los polinomios de Bernstein empleados depende del número de puntos de control; para un conjunto de  $n + 1$  puntos, los

a smooth maximum and minimum, but they are not actually random. In figure 8.8(c), the polynomial fits better the data, and the residuals are randomly distributed. If we compare them with the residuals presented in figure 8.8(d), we see these last are smaller but present essentially the same random distribution. We may conclude that the data approximation using a fourth-degree polynomial does not add any useful information on the function shape the data follow, and, worse than this, we are starting to include the data errors in the fitting.

## 8.5. Bézier Curves

Bézier's curves can connect points using curves with arbitrary shape. We define a Bézier's curve using a set of  $n + 1$  points,  $\{\vec{p}_0, \vec{p}_2, \dots, \vec{p}_n\}$  which are known as control points. The order of the control points play a key role in the definition of the corresponding Bézier's curve. The curve pass through the end points  $\vec{p}_0$  and  $\vec{p}_n$ . The remaining control points allow us to define the curve shape, which has the interesting properties of being always tangent in point  $\vec{p}_0$  to the line that connects  $\vec{p}_0$  with  $\vec{p}_1$  and tangent in point  $\vec{p}_n$  to the line that connect  $\vec{p}_{n-1}$  with  $\vec{p}_n$ .

To define the curve, we associate to the control point  $\vec{p}_i$ , a function known as fusion function. In the case of Bezier's curves, we use as fusion function Berstein's polynomials,

The Berstein's polynomials degree we use depends on the number of control points; for a set of  $n + 1$ , we need polynomials of degree  $n$ .

polinomios son de grado  $n$ . La variable  $t$  es un parámetro que varía entre 0 y 1.

La ecuación de la curva de Bézier que utiliza un conjunto de  $n + 1$  puntos de control,  $\{\vec{p}_0, \vec{p}_2, \dots, \vec{p}_n\}$ , se define a partir de los polinomios de Bernstein como,

$$\vec{p}(t) = \sum_{i=0}^n B_i^n(t) \cdot \vec{p}_i = \sum_{i=0}^n \binom{n}{i} (1-t)^{n-i} t^i \cdot \vec{p}_i$$

La expresión anterior da como resultado  $\vec{p}_0$  si hacemos  $t = 0$  y  $\vec{p}_n$  si hacemos  $t = 1$ . para los valores comprendidos entre  $t = 0$  y  $t = 1$ , los puntos  $\vec{p}(t)$  trazarán una curva entre  $\vec{p}_0$  y  $\vec{p}_n$ .

Veamos un ejemplo sencillo. Supongamos que queremos unir los puntos  $\vec{p}_0 = (0, 1)$  y  $\vec{p}_n = (3, 1)$  mediante curvas de Bézier. Si no añadimos ningún punto más de control,  $\{\vec{p}_0 = (0, 1), \vec{p}_1 = (3, 1)\}$ , la curva de Bézier que obtendremos será una recta que unirá los dos puntos,

$$\vec{p}(t) = \binom{1}{0} (1-t)^1 t^0 \cdot (0, 1) + \binom{1}{1} (1-t)^0 t^1 \cdot (3, 1) = (1-t) \cdot (0, 1) + t \cdot (3, 1)$$

Si sepáramos las componentes  $x$  e  $y$  del vector  $\vec{p}(t)$ ,

$$\begin{aligned} x &= 3t \\ y &= 1 \end{aligned}$$

Se trata de la ecuación del segmento horizontal que une los puntos  $\vec{p}_0 = (0, 1)$  y  $\vec{p}_n = (3, 1)$

Si añadimos un punto más de control, por ejemplo:  $\vec{p}_1 = (1, 2) \rightarrow \{\vec{p}_0 = (0, 1), \vec{p}_1 = (1, 2), \vec{p}_2 = (3, 1)\}$ , la curva resultante será ahora un segmento de un polinomio de segundo grado en la variable  $t$ ,

$$\begin{aligned} \vec{p}(t) &= \binom{2}{0} (1-t)^2 t^0 \cdot (0, 1) + \binom{2}{1} (1-t) t^1 \cdot (1, 2) + \binom{2}{2} (1-t)^0 t^2 \cdot (3, 1) = \\ &= (1-t)^2 \cdot (0, 1) + 2(1-t)t \cdot (1, 2) + t^2 \cdot (3, 1) \end{aligned}$$

Según vamos aumentando el número de los puntos de control, iremos empleando polinomios de mayor grado. El segmento de polinomio empleado en cada caso para unir los

Variable  $t$  is a parameter which varies between 0 and 1.

The Bezier's curve for a set of  $n+1$  control points,  $\{\vec{p}_0, \vec{p}_2, \dots, \vec{p}_n\}$ , can be defined using Bernstein's polynomials, according to the following equation,

The above expression yields  $\vec{p}_0$  as a result, if we take  $t = 0$  and  $\vec{p}_n$  for  $t = 1$ . For values of  $t$  between  $t = 0$  and  $t = 1$ , points  $\vec{p}(t)$  will describe a curve between  $\vec{p}_0$  y  $\vec{p}_n$ .

Let's see a basic example. Suppose we want to connect point  $\vec{p}_0 = (0, 1)$  with point  $\vec{p}_1$  using a Bezier's curve. If we do not define any other control point,  $\{\vec{p}_0 = (0, 1), \vec{p}_1 = (3, 1)\}$ , The Bezier's curve we will obtain will be a line connection both points.

If we split the components  $x$  and  $y$  of vector  $\vec{p}(t)$ ,

It is the equation of the horizontal segment that joints the points  $\vec{p}_0 = (0, 1)$  y  $\vec{p}_n = (3, 1)$

If we add a control point more, for instance:  $\vec{p}_1 = (1, 2) \rightarrow \{\vec{p}_0 = (0, 1), \vec{p}_1 = (1, 2), \vec{p}_2 = (3, 1)\}$ , the resulting curve will be now a segment of a second degree polynomial on the variable  $t$ ,

As we increase the number of control points, we also increase de degree of the polynomial used to connect the first and last control points. The shape of this polynomial will change de-

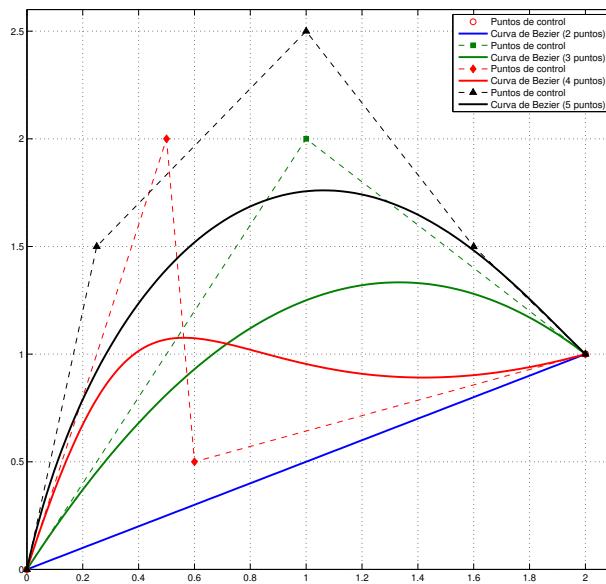


Figura 8.9: Curvas de Bézier trazadas entre los puntos  $P_0 = (0, 0)$  y  $P_n = (2, 1)$ , variando el número y posición de los puntos de control.

Figure 8.9: Bézier curves traced between points  $P_0 = (0, 0)$  y  $P_n = (2, 1)$ , variating the control points number and positions

puntos de control inicial y final variará dependiendo de los puntos de control intermedios empleados.

La figura 8.9 muestra un ejemplo en el que se han construido varias curvas de Bézier sobre los mismos puntos de paso inicial y final. Es fácil observar cómo la forma de la curva depende del número y la posición de los puntos de control. Si unimos éstos por orden mediante segmentos rectos, obtenemos un polígono que recibe el nombre de polígono de control. En la figura 8.9 se han representado los polígonos de control mediante líneas de puntos.

El siguiente código permite calcular y dibujar una curva de Bézier a partir de sus puntos de control, empleando las funciones `bezier` y `bezier_sc`

pending on the intermediate control point we use.

Figure 8.9 shows an example in which we have built several Bézier's curves using the same initial and final controls points. Notice how the shape of the curve depends on the number and position of the control points. If we orderly connect the control points using straight segments we get a polygon know as control polygon. In figure 8.9 the control polygons have been represented using dashed lines.

The following code includes functions `bezier` and `bezier_sc` to compute and draw Bezier's curves define from their control points.

---

bezier.py

---

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Thu Aug  8 19:41:24 2024

```

```

4  This module defines a functions to generate Bezier Polynomials
5  bezier returns the value of the curve as a function of t
6  bezier_rc uses function bezier to obtain the values of the curve
7  y an array of point a draw the result
8  @author: abierto
9  """
10 import numpy as np
11 import matplotlib.pyplot as pl
12
13
14 def bezier(p,t):
15     """
16     Calculates de value of a bezier curve with control points
17     in the array p at value t
18
19     Parameters
20     -----
21     p : TYPE numpy array
22         DESCRIPTION. Numpy array of dimension 2Xn where n is the
23         number of control points. first row contains x coordinate
24         and second row contain y coordinate, the curve pass through
25         the first a last control points
26     t : TYPE
27         DESCRIPTION. Berstein's polynomial parameter t belongs
28         to [0,1]
29
30     Returns
31     -----
32     ber : TYPE np.array
33         DESCRIPTION. ber[0] x coordinate of the bezier's curve
34         at t. ber[1] y coordinate of the bezier's curve at t
35
36     """
37     n = p.shape[1]
38     num = np.arange(1,n).prod()
39     #we firt calculate all factorial needed
40     ftal = np.insert(np.arange(1.,n).cumprod(),0,1)
41     ber = np.zeros(2)
42     for i in range(n):
43         f = num/ftal[i]/ftal[n-1-i]
44         ber = ber + f*p[:,i]*(1-t)**(n-1-i)*t**i
45     return(ber)
46
47 def bezier_sc(p,t):
48     """
49     This function use funtion bezier to reproduce the bezier curve
50     with control point containing in p.
51     Parameters
52     -----
53     p : TYPE numpy array
54         DESCRIPTION. Numpy array of dimension 2Xn where n is the
55         number of control points. first row contains x coordinate

```

```

56         and second row contain y coordinate, the curve pass through
57         the first a last control points
58     t : TYPE numpy array
59         DESCRIPTION. It should be an array of ordered values for
60         parameter t. To reproduce de whole Bezier's curve use
61         t = numpy.arange(0,1+step,step) the smaller step the smoother
62         the result
63
64     Returns
65     -----
66     pt : TYPE numpy array
67         DESCRIPTION. a 2xm array with m the number of data in t
68         first row contains the x coordinates at the values of t
69         and second row the y coordinates at the values of t
70
71     """
72     pt = np.array([bezier(p,i) for i in t]).T
73     pl.plot(p[0],p[1],'or')
74     pl.plot(p[0],p[1],'-.')
75     pl.plot(pt[0],pt[1])
76     return(pt)
77
78 def bzeq(p,n,step):
79     """
80     Computes Bezier's curves of higher degree, equivalent to a
81     given Bezier's curve
82
83     Parameters
84     -----
85     p : TYPE numpy array
86         DESCRIPTION. 2xn array containing the control points of
87         the Bezier's curve to be derivate. (each column a point)
88     n : TYPE integer
89         DESCRIPTION. Degree of the equivalent Bezier's, it should
90         be greater than the number of points in p.
91     step : TYPE double
92         DESCRIPTION. a step to generate an ordered, array which
93         equispaciate values in [0,1].
94
95     Returns
96     -----
97     p : TYPE numpy array
98         DESCRIPTION. 2x(n-1) array containing the control points
99         of the equivalent Bezier's curve
100
101    """
102    c = p.shape[1]
103    t = np.arange(0,1+step,step)
104    bezier_sc(p, t) #draw the Bz curve
105    if c < n:
106        #adding one more control point and recalculating
107        # the control points

```

## 390CAPÍTULO/CHAPTER 8. INTERP. Y AJUST. FUNCIONES. \*INTERP. & FUNCT FITTING

```

108     p = np.append(p, [[0], [0]], axis=1)
109     pp = p.copy()
110     for i in range(1,c+1):
111         pp[:,i] = p[:,i-1]*i/(c) + (1-i/(c))*p[:,i]
112         #the function calls itself till the number of points
113         #equals n
114     p = bzeq(pp,n,step)
115     # and it returns the control points. In the meanwhile it
116     #has drawn all the equivalent curves with degrees between
117     # c and n
118     return(p)
119
120 def dbbez(p):
121     """
122     Compute a Bezier's . Curve derivative, defined by control
123     points in p. It also draw de odograph and the derivatives
124     at some selected points and the Bezier's curve and
125     the derivative, as tangents vector, as some selected points.
126
127     Parameters
128     -----
129     p : TYPE numpy array
130         DESCRIPTION. Control points of the Bezier's curve we
131         wanna derivate.
132
133     Returns
134     -----
135     d : TYPE numpy array
136         DESCRIPTION. Controls points of the derivative.
137
138     """
139     grado = p.shape[1]-1
140     d = np.zeros([2,grado])
141     #compute the control points of the derivativa
142     for i in range(grado):
143         d[:,i] = (grado-1)*(p[:,i+1]-p[:,i])
144     t = np.arange(0,1+0.01,0.01)
145     #plot de odograph
146     pl.figure(1)
147     v = bezier_sc(d, t)
148     pl.quiver(np.zeros(12),np.zeros(12),\
149               v[0,:,:9],v[1,:,:9],\
150               angles='xy', scale_units='xy', scale=1)
151
152     #plot the curve and its derivative. (the escale of the
153     #vector is arbitrary)
154     pl.figure(2)
155     ptos = bezier_sc(p,t)
156     pl.quiver(ptos[0,:,:9],ptos[1,:,:9],\
157               v[0,:,:9],v[1,:,:9])
158     pl.axis('equal')
159     return(d)

```

**Curvas equivalentes de grado superior.**

Dada una curva de Bézier, representada por un polinomio de Bernstein de grado  $n$ , es posible encontrar polinomios de grado superior que representan la misma curva de Bézier. Lógicamente esto supone un aumento del número de puntos de control.

Supongamos que tenemos una curva de Bézier con cuatro puntos de control,

$$\vec{p}(t) = \vec{p}_0 B_0^3(t) + \vec{p}_1 B_1^3(t) + \vec{p}_2 B_2^3(t) + \vec{p}_3 B_3^3(t)$$

Si multiplicamos este polinomio por  $t + (1 - t) \equiv 1$ , el polinomio no varía y por tanto representa la misma curva de Bézier. Sin embargo, tendríamos ahora un polinomio un grado superior al inicial. Si después de la multiplicación agrupamos términos de la forma  $(1 - t)^{n-1} t^i$ , Podríamos obtener el valor de los nuevos puntos de control,

**Upper degree equivalent curves.** Given a Bezier's curve, represented by a Bernstein's polynomial of degree  $n$ , it is possible to find Bernstein's polynomials of higher degree, which also describes this same Beezier's curve. Indeed, this means an increase in the number of the control points.

Suposse we have a Bezier's curve defined by four control points,

If we multiply this polynomial by  $t + (1 - t) \equiv 1$ , the polynomial remains the same an so it represent the same Bezier's curve too. However, we have now a polynomial one degree higher than the initial one. If after multiplying we group terms like  $(1 - t)^{n-1} t^i$ , we can compute the value of the new control points,

$$\begin{aligned}\vec{p}_0^+ &= \vec{p}_0 \\ \vec{p}_1^+ &= \frac{1}{4} \vec{p}_0 + \frac{3}{4} \vec{p}_1 \\ \vec{p}_2^+ &= \frac{2}{4} \vec{p}_2 + \frac{2}{4} \vec{p}_3 \\ \vec{p}_3^+ &= \frac{3}{4} \vec{p}_2 + \frac{1}{4} \vec{p}_3 \\ \vec{p}_4^+ &= \vec{p}_3\end{aligned}$$

y, en general, los puntos de control de la curva de Bézier de grado  $n + 1$  equivalente a una dada de grado  $n$  puede obtenerse como,

And, in general, we can compute the control points for the Bezier's curve of degree  $n+1$  equivalent to a previous defined curve of degree  $n$  as,

$$\vec{p}_i^+ = \alpha_i \vec{p}_{i-1} + (1 - \alpha_i) \vec{p}_i, \quad \alpha_i = \frac{i}{n+1}$$

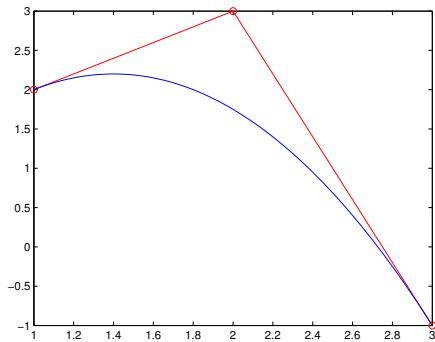
Mediante la ecuación anterior, es posible obtener iterativamente los puntos de control de la curva de Bézier equivalente a una dada de cualquier grado. Una propiedad interesante es que según aumentamos el número de puntos de control empleados, estos y el polígono de control correspondiente, van convergiendo a la curva de Bézier.

Using the above equation, we can iteratively obtain the control points of a Bezier's curve equivalent to any other given of whatever degree. An interesting property is that, as we increase the number of control points, these and the control polygon tend to converge to the Bezier's curve.

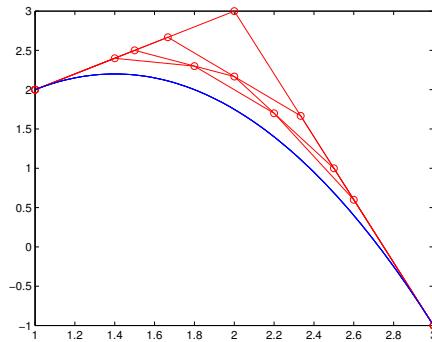
Figure 8.10 shows an example using a Bézier's

La figura 8.10 muestra un ejemplo para una curva de Bézier construida a partir de tres puntos de control. Es fácil ver cómo a pesar de aumentar el número de puntos de control, la curva resultante es siempre la misma. También es fácil ver en la figura 8.10(d) la convergencia de los polígonos de control hacia la curva.

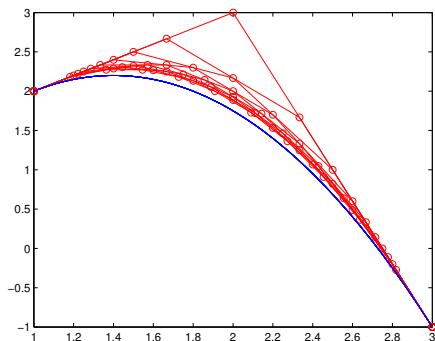
curve built from three control points. As can be seen, we always obtain the same curve, although we have increased the number of control points. It is also easy to see in figure 8.10(d) that the control polygons also converge to the Bezier's curve.



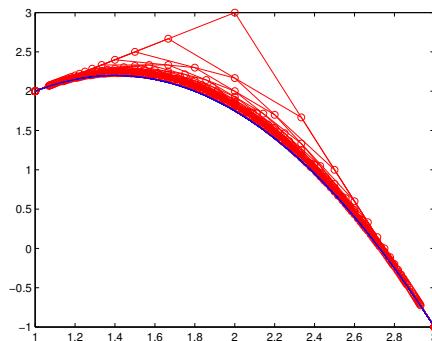
(a) Curva original (3 puntos) / Original curve (3 points)



(b) Curvas equivalentes de 4 a 6 puntos / 4 to 6 points equivalent curves



(c) Curvas equivalentes de 4 a 12 puntos / 4 to 12 points Equivalent curves



(d) Curvas equivalentes de 4 a 30 puntos / 4 to 30 points equivalent curve

Figura 8.10: Curvas de Bézier equivalentes, construidas a partir de una curva con tres puntos de control.

Figure 8.10: Equivalent curves, built from a three control point Bezier's curve.

La función `bzeq` del módulo `bezier.py`, incluido más arriba permite calcular la curva equivalente de Bézier a una dada, para cualquier número de puntos de control que se desee.

Function `bzeq` in `bezier.py` module above include, allows us to calculate the Bezier's curve equivalent to any other given, using any number of control points.

**Derivadas.** Las derivadas de una curva de Bézier con respecto al parámetro  $t$  son particularmente fáciles de obtener a partir de los puntos de control. Si tenemos una curva de Bézier de grado  $n$ , definida mediante puntos de control  $\vec{p}_i$ . Su derivada primera con respecto a  $t$  será una curva de Bézier de grado  $n - 1$ , cuyos puntos de control  $\vec{d}_i$  puede obtenerse como:

$$\vec{d}_i = n(\vec{p}_{i+1} - \vec{p}_i)$$

La nueva curva de Bézier obtenida de este modo, es una hodógrafa; representa el extremo de un vector tangente en cada punto a la curva de Bézier original y guarda una relación directa con la velocidad a la que se recorrería dicha curva.

La figura 8.11(a), muestra una curva de Bézier sobre la que se ha trazado el vector derivada para algunos puntos. La figura 8.11(b) muestra la hodógrafa correspondiente y de nuevo los mismos vectores derivada de la figura 8.11(a).

La función `dbez`, incluida en el módulo `bezier.py` permite calcular los puntos de control de la derivada de una curva de Bézier.

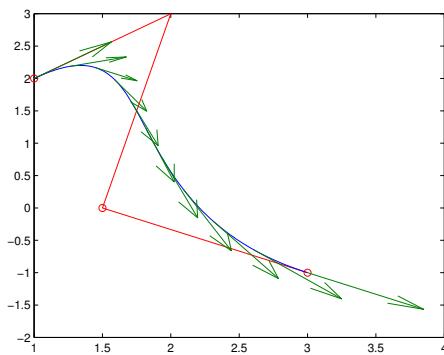
**Derivatives.** To compute the derivatives of a Bezier's curve with respect to parameter  $t$  is particularly easy, using the control points. Suppose we have a degree  $n$  Bezier's curve, defined from control points  $\vec{p}_i$ . Its first derivative with respect to  $t$  will be a Bezier's curve of degree  $n - 1$ , whose control point we can compute as:

$$\vec{d}_i = n(\vec{p}_{i+1} - \vec{p}_i)$$

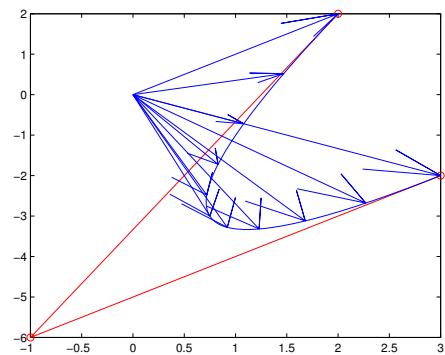
The new Bezier curve we obtain is a hodograph. It represents the position of the tips of vectors tangent in each point to the original Bezier curve and has a direct relationship with the speed at which you can travel across the curve at any point.

Figure 8.11(a) shows a Bezier's curve and the derivative vector at some points of the curve. Figure 8.11(b) shows the hodograph corresponding to and the same derivative vectors of the curve presented in figure 8.11(a).

Function `dbez`, belonging to the module `bezier.py` included above, allows us to compute the derivative of any Bezier's curve.



(a) Curva de Bézier (4 puntos) / Bézier's curve (4 points)



(b) Derivada (hodógrafa) / Derivative (hodograph)

Figura 8.11: Curva de Bézier y su derivada con respecto al parámetro del polinomio de Bernstein que la define:  $t \in [0, 1]$

Figure 8.11: Bézier's curve and its derivative with respect to the Bernstein's polynomial parameter which defines the curve  $t \in [0, 1]$

**Interpolación con curvas de Bézier** Podemos emplear curvas de Bézier para interpolar un conjunto de puntos  $\{\vec{p}_0, \dots, \vec{p}_m\}$ . Si empleamos un curva para interpolar cada par de puntos,  $\vec{p}_i, \vec{p}_{i+1}$ ,  $i = 1, \dots, m - 1$  tenemos asegurada la continuidad en los puntos interpolados puesto que las curvas tienen que pasar por ellos. Como en el caso de la interpolación mediante splines, podemos imponer continuidad en las derivadas para conseguir una curva de interpolación suave. En el caso de las curvas de Bézier esto es particularmente simple. Si llamamos  $B$  a la curva de Bézier de grado  $n$  construida entre los puntos  $\vec{p}_{i-1}, \vec{p}_i$  con puntos de control,  $\vec{p}_{i-1}, \vec{b}_1, \vec{b}_2, \dots, \vec{b}_{n-1}, \vec{p}_i$ , y  $C$  a la curva de Bézier de grado  $s$  construida entre los puntos  $\vec{p}_i, \vec{p}_{i+1}$  con puntos de control,  $\vec{p}_i, \vec{c}_1, \vec{c}_2, \dots, \vec{c}_{s-1}, \vec{p}_{i+1}$ . Para asegurar la continuidad en la primera derivada en el punto  $\vec{p}_i$  basta imponer,

$$n \cdot (\vec{p}_i - \vec{b}_{n-1}) = s \cdot (\vec{c}_1 - \vec{p}_i)$$

Esta condición impone una relación entre el penúltimo punto de control de la curva  $B$  y el segundo punto de control de la curva  $C$ . Pero deja completa libertad sobre el resto de los puntos de control elegidos para construir las curvas.

Podemos, por ejemplo, elegir libremente todos los puntos de control de la curva  $B$  y obtener a partir de ella el punto  $\vec{c}_1$ ,

$$\vec{c}_1 = \frac{n+s}{s} \vec{p}_i - \frac{n}{s} \vec{b}_{n-1}$$

La figura 8.12 muestra un ejemplo de interpolación en la que se ha aplicado la condición de continuidad en la derivada que acabamos de describir.

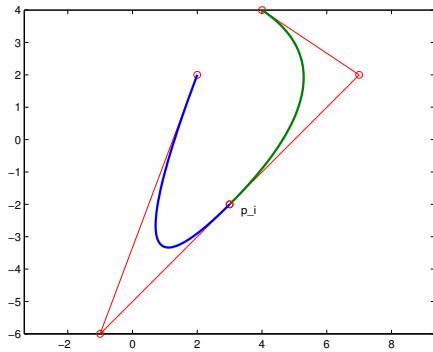
**Interpolation using Bézier's curves** We can use Bézier's curve for interpolating a set of points,  $\{\vec{p}_0, \dots, \vec{p}_m\}$ . If we use a curve to interpolate every pair of consecutive points,  $\vec{p}_i, \vec{p}_{i+1}$ ,  $i = 1, \dots, m - 1$  we assure the continuity in the interpolated points because the curves have necessarily to pass through them. As in the case of spline interpolation, we can also impose continuity to the derivatives to get a smooth interpolating curve. In the case of Bézier's curves, this is especially simple. If we call  $B$  to the  $n$ -degree Bezier's curve built between the points  $\vec{p}_{i-1}, \vec{p}_i$  with control points,  $\vec{p}_{i-1}, \vec{b}_1, \vec{b}_2, \dots, \vec{b}_{n-1}, \vec{p}_i$ , and  $C$  to the  $s$ -degree Bezier's curve built between the points  $\vec{p}_i, \vec{p}_{i+1}$  with control points,  $\vec{p}_i, \vec{c}_1, \vec{c}_2, \dots, \vec{c}_{s-1}, \vec{p}_{i+1}$ ; we can assure the continuity of the first derivative in the point  $\vec{p}_i$  just imposing that,

This condition imposes a link between the second-last control point of the curve  $B$  and the second control point of the curve  $C$ . However, it allows free election of the remaining control points to build the curves as desired.

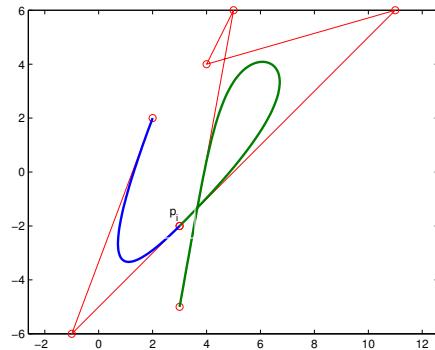
For instance we can freely choose all control points of curve  $B$  and use it to compute the point  $\vec{c}_1$ ,

Figure 8.12 shows an example of interpolation in which we have applied the derivative continuity condition we have described.





(a) Interpolación mediante curvas de Bézier de 3 puntos / Interpolation using three control points Bezier's curves)



(b) Interpolación mediante curvas de Bézier de 3 y 4 puntos / iNterpolation using 3 and 4 control points Bezziers's curves

Figura 8.12: Interpolación de tres puntos mediante dos curvas de Bézier  
Figure 8.12: interpolating three point using two Bezier's curves

## 8.6. ejercicios

1. Carga en python los datos del fichero **datos.txt**<sup>1</sup> y realiza las siguientes tareas:

- a) Crea una función que a partir de dos vectores de datos  $x, y$  de igual longitud  $n+1$ , calcula la matriz de Vandermonde necesaria para obtener el polinomio de interpolación asociado a los puntos. Empleando la primera columna de datos contenida en **datos.txt** como datos  $x$  y la segunda como datos  $y$ , genera el polinomio de interpolación,

$$p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$$

Calcula el valor que toma el polinomio de interpolación en 100 puntos equiespaciados entre los valores  $x_0$  y  $x_n$  de los datos del fichero. Dibuja en una misma gráfica los resultados obtenidos, empleando una línea continua, y los valores del fichero, mediante puntos separados

## 8.6. exercises

1. Load in Python the date held in file **datos.txt**<sup>1</sup> an carry out the following tasks:

- a) Build a function that taking two data vector,  $x, y$  of the same length,  $n+1$ , computes the Vandermonde's matrix needed for obtaining the interpolating polynomial associated to the points. Using the first data column in **data.txt** as  $x$  and the second column as  $y$  compute the the interpolating polynomial,

$$p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$$

Compute the values of the interpolating polynomial in 100 equispaced points between values  $x_0$  and  $x_n$  of the data file. In the same graphic, draw the results obtained using a continuous line and the file data as single points, using the symbol

<sup>1</sup>disponible en <https://github.com/UCM-237/LCC/tree/master/datos>

<sup>1</sup>available in <https://github.com/UCM-237/LCC/tree/master/datos>

- empleando el símbolo que prefieras. Comprueba que el polinomio pasa por los puntos contenidos en el fichero.
- b) Reproduce en python la función 'Lagrange' de la sección 8.2.2 para calcular el polinomio interpolador de Lagrange. Emplea la función que acabas de crear para recalcular los valores del polinomio de interpolación realizado en el ejercicio anterior y comprueba que los resultados obtenidos son los mismos que empleando la matriz de Vandermonde.
- c) A partir de los ejemplos de la sección sección 8.2.3, crea un programa que calcule los coeficientes del polinomio interpolador de diferencias divididas a partir de dos vectores de datos  $x, y$  de igual longitud  $n + 1$  y un segundo programa que calcule el valor del polinomio en un punto cualquiera a partir de los coeficientes obtenidos con el primer programa. Vuelve a calcular, empleando ahora el polinomio de diferencias divididas, los valores del polinomio de interpolación sobre los mismos datos empleados en los ejercicios anteriores y comprueba que da los mismos resultados.
- d) Por último, crea una función que calcule el polinomio de interpolación de Newton-Gregory (sección 8.2.4). ¿Es posible usarlos para interpolar los datos del archivo `datos.txt`? Si la respuesta es afirmativa, repite el cálculo del polinomio de interpolación, empleando Newton-Gregory y comprueba si coincide con lo obtenido en los ejercicios anteriores.
- e) Usa la función de numpy `interp1d` y emplea de nuevo los datos del fichero `datos.txt`, para obtener el resultado de interpolar los valores en 100 puntos equiespaciados entre los valores  $x_0$  y  $x_n$  del fiche-
- you prefer. Check that the polynomial passes through the data point included in the file.
- b) Write the 'Lagrange' function for computing the Interpolating Lagrange polynomial in Python, included in section 8.2.2. Use the function you just created to recompute the values of the interpolating polynomial obtained in the previous exercise and check that you get the same results as using Vandermonde's matrix.
- c) Departing from the examples in section 8.2.3, build a program for computing the divided differences polynomial coefficients, using two data arrays  $x$  and  $y$  with the same length,  $n + 1$ . Build also a second program for computing the value of the polynomial in any point, using the coefficients computed with the previous program. Recompute one more time, now using the divided differences polynomial the values of the interpolation polynomial, of the same data used in previous exercises. Check that you get the same results.
- d) Lastly, build a function that compute the Newton-Gregory's interpolating polynomials (section 8.2.4). It is possible to interpolate the data held in the file `datos.txt` using the Newton-Gregory's polynomial? If the answer is 'yes', use your function just written to repeat the computing of the interpolation polynomial carried out in previous section and check that the results meet those obtained in previous sections.
- e) Using the Numpy's function `interp1d` and the data contained in file `datos.txt`, compute the result of interpolating in 100 equispaced points between values  $x_0$  and  $x_n$  of the file dataset. Do it using the method '`nearest`' and '`linear`'. Repeat the computing but now using

- ro. Emplea para ello los métodos '`nearest`' y '`linear`'. Repite el cálculo empleando ahora la función de numpy '`CubicSpline`'. Dibuja los resultados en la misma gráfica empleada en el ejercicio 1a)
2. Construye a partir del código de ejemplo de la sección 8.4, una función que calcule el polinomio de grado  $n$  que ajusta por mínimos cuadrados un conjunto de pares de datos  $(x, y)$ . Pruebalo sobre los datos del fichero `datos.txt`, sin emplear pesos. Compara los coeficientes del polinomio obtenido con los que se obtienen empleando la función `Polynomial.fit` de Numpy. ¿Qué conclusión sacas?
  3. Añade el código necesario al programa anterior para que, una vez obtenidos los coeficientes del polinomio de mínimos cuadrados, la función calcule  $y$  devuelva un vector  $r$  con los valores de los residuos,  $r = y - p(x)$ , donde  $x$  e  $y$  son los vectores del conjunto de datos para los que se ha obtenido el polinomio de mínimos cuadrados y  $p$  los valores obtenidos aplicando el polinomio a los valores  $x$  de la colección de datos.

## 8.7. Test del curso 2020/21

**Problema 1.** Una cuerda de escalada aumenta su longitud cuando está sometida a una tensión estacionaria. En particular, el aumento de longitud sigue la siguiente ley

$$T = b \tanh(ax), \quad (8.1)$$

donde  $T \in \mathbb{R}^+$  es la tensión aplicada en Newtons,  $x \in \mathbb{R}^+$  es la elongación de la cuerda en metros y  $a, b \in \mathbb{R}^+$  son parámetros constantes que dependen de las características de la cuerda.

En función del comportamiento físico de la cuerda, podemos distinguir tres regímenes:

- *Comportamiento elástico:* Se dice que el comportamiento de la cuerda es elástico si al desaparecer la tensión la cuerda recupera su longitud original. Esto

the Numpy function '`CubicSpline`'. Draw the results in the same graphic where you drew exercise 1a) results.

2. create a function that calculates the  $n$ -degree polynomial using the least squared method to fit a set of data pairs  $(x, y)$ , using the code in section 8.4 as a reference. Test your function with the data in the file `datos.txt` without employing weights. Afterward, compare the coefficients produced by your program with those obtained using the Numpy function `Polynomial.fit`. What conclusion can you draw from the comparison?
3. Add the necessary code to the previous program so that, once the coefficients of the least square polynomials has been obtained, the function calculates and return a vector  $r$  with the values of the residuals,  $r = y - p(x)$ , where  $x$  and  $y$  are the data set for which the function has computed the least square polynomial and  $p$  the values obtained applying the polynomial to the  $x$  values of the data set.

## 8.7. Course 2020/21 test

**Question 1.** A Climbing rope stretches when it suffers a stationary strain. Specifically, the increase in length follows the following law,

$$T = b \tanh(ax), \quad (8.1)$$

Where  $T \in \mathbb{R}^+$  is the applied strain in Newtons,  $x \in \mathbb{R}^+$  is the rope stretching in meters and  $a, b \in \mathbb{R}^+$  are constant parameter which depend on the rope characteristics.

According to the rope physical behaviour, we can observe three regimes.

- *elastic behaviour:* The rope follows an elastic behaviour if it recovers its original length once we withdraw the strain. This is the normal behaviour under small

se cumple para tensiones estacionarias pequeñas, y las elongaciones están acotadas por un valor positivo  $x_{le}$ , esto es,  $0 \leq x \leq x_{le}$ . En régimen elástico, la ecuación (8.7) puede aproximarse por la siguiente expresión

$$T = \kappa x - \gamma x^3, \quad (8.2)$$

donde  $\kappa, \gamma \in \mathbb{R}^+$  son también constantes.

- *Comportamiento plástico:* Se dice que el comportamiento de la cuerda es plástico si al desaparecer la tensión la cuerda se deforma y no recupera su longitud original. Esto ocurre para tensiones medias, y la elongación alcanzada en este régimen está también acotada por  $x_{le} < x \leq x_{max}$ .
- *Rotura:* Para tensiones grandes la cuerda no admite elongaciones mayores que  $x_{max}$  y se rompe.

La fábrica de cuerdas de escalada *Pa'bennos Matao S.L.* ha realizado un estudio sobre un nuevo modelo de cuerda. En dicho estudio se fijó un extremo de la cuerda a una mesa abatible suficientemente grande (que hará la función de un plano inclinado), y se ató el otro extremo de la cuerda a una pesa. De manera secuencial se fue incrementando el ángulo formado por la mesa con la horizontal. En particular, se empezó con cero grados y aumentando el ángulo hasta que finalmente la cuerda alcanzó  $x_{max}$  y se rompió.

Del estudio se pudo registrar la elongación sufrida por la cuerda por los distintos  $i \in \{1, \dots, 52\}$  ángulos de inclinación. Estos datos están en el archivo: `cuerda.txt`<sup>2</sup>, en donde:

- La primera columna corresponde a los ángulos  $\theta_i$  medidas en radianes.
- La segunda columna corresponde a las elongaciones  $x_i$  medidas en metros.

---

<sup>2</sup>disponible en <https://github.com/UCM-237/LCC/tree/master/datos>

stationary strains. In this case, the stretching is upper bounded by a positive value  $x_{le}$ , that is,  $0 \leq x \leq x_{le}$ . In the elastic regime equation can be approximated by the following expression,

$$T = \kappa x - \gamma x^3, \quad (8.2)$$

where  $\kappa, \gamma \in \mathbb{R}^+$  are also constant.

- *Plastic behaviour:* The rope behaviour is considered plastic if the rope deforms after removing the strain and does not recover its original length. This behaviour takes place for intermediate strains and, in this plastic regime, the rope stretching is also bounded between two limits  $x_{le} < x \leq x_{max}$ .
- *Breaking off:* The rope does not admit a bigger stretching and breaks off for larger strains.

The climbing ropes company *Slink Yer Hook, ltd.* conducted a study on a new brand of rope. They attached one end of the rope to a tilting table, which was used as an inclined plane, and the other to a weight for the study. The angle between the table and the horizontal was gradually increased. The test started with a zero-degree angle, and the angle was increased until the rope reached the length  $x_{max}$  and broke.

During the study, the stretching suffered by the rope for different tilting angles  $i \in \{1, \dots, 52\}$  was registered. These data are available in the file `cuerda.txt`<sup>2</sup>, where:

- The first column holds the angles  $\theta_i$  measured in radians.
  - The second column holds the elongations  $x_i$  measured in meters.
1. **(1 point)** Estimate the stretching limit value  $x_{max}$  beyond which the rope breaks.
  2. **(1 point)** compute the strain exerted to the rope for the weight at any table tilting angle  $\theta_i$ , that is,

$$T_i = mg \sin(\theta_i), \quad (8.3)$$

---

<sup>2</sup>Available in <https://github.com/UCM-237/LCC/tree/master/datos>

1. **(1 punto)** Estima el valor de la elongación  $x_{max}$  para el cual se produce la rotura de la cuerda.
2. **(1 punto)** Obtén la tensión ejercida por la pesa sobre la cuerda para cada ángulo de inclinación  $\theta_i$  de la mesa, esto es

$$T_i = mg \sin(\theta_i), \quad (8.3)$$

donde  $m = 1000$  Kg y  $g = 9.8$  m/s<sup>2</sup>. Representa gráficamente los datos:  $T_i$  frente a  $x_i$ .

3. A partir de los pares de datos  $T_i$  y  $x_i$  podemos estimar la ecuación (8.2).

- a) **(2 puntos)** Ajusta los datos por mínimos cuadrados a un polinomio de grado tres. Dado que dicha aproximación solo es válida para el régimen de comportamiento elástico, es imprescindible realizar el ajuste del polinomio asignando pesos a cada par de datos. Para dar más valor a las elongaciones pequeñas y menos a las grandes, utiliza la siguiente expresión para definir los pesos

$$\omega_i = e^{-10x_i^2}. \quad (8.4)$$

- b) **(1 punto)** Representa, sobre la gráfica dibujada en el apartado 2a, el polinomio obtenido en el apartado 3a. Según tu criterio, ¿es razonable el ajuste realizado?
4. **(1 punto)** Los coeficientes de las ecuaciones (8.7) y (8.2) están relacionados por las siguientes expresiones

$$\kappa = b \cdot a, \quad \gamma = \frac{b \cdot a^3}{3}.$$

Calcula los valores de  $a$  y  $b$  a partir de los coeficientes del polinomio obtenido en el apartado 3a. Representa, en la misma gráfica de los apartados anteriores, la función  $T(x) = b \tanh(ax)$ . Explica, a la vista del gráfico, si los resultados obtenidos son razonables o no.

where  $m = 1000$  Kg and  $g = 9.8$  m/s<sup>2</sup>. Draw a  $T_i$  vs  $x_i$  graph.

3. From data pairs  $T_i$  and  $x_i$  we can estimate equation (8.2).

- a) **(2 points)** To fit the data, utilize a three-degree least squares polynomial. Because this method is only suitable for elastic behavior, it's important to assign weights to the data when calculating the polynomial. Use the following expression to define the weights, which will enhance the influence of small elongations and reduce the impact of larger ones.

$$\omega_i = e^{-10x_i^2}. \quad (8.4)$$

- b) **(1 point)** Plot on top the graph drew in question 2a, the polynomial you have computed in question 3b. Is it reasonable the fitting carried out? explain why.

4. **(1 point)** Coefficient in equations (8.7) y (8.2) are related by the following expressions,

$$\kappa = b \cdot a, \quad \gamma = \frac{b \cdot a^3}{3}.$$

Compute the values of parameters  $a$  and  $b$  departing from the coefficients of the polynomial obtained in question 3a. Plot the function  $T(x) = b \tanh(ax)$  in the same graphic used in the previous questions. Explain, according to the graphic, if the results achieved are reasonable or not.

5. **(1 point)** Compute the value of residuals  $r_i = T_i - P_3(x_i)$  where  $P_3(x)$  is the polynomial obtained in question 3a. Suppose that the limit  $x_{le}$  for the rope elastic behaviour is reached when  $r_i \approx 500N$ , Find an approximated value for  $x_{le}$ .

**Note:** It is not enough to draw the residuals and estimate  $x_{le}$  by simple inspection. You must use code to do it.

5. **(1 punto)** Calcula el valor de los residuos  $r_i = T_i - P_3(x_i)$ , donde  $P_3(x)$  es el polinomio de grado tres obtenido en el apartado 3a. Si la cota  $x_{le}$  para el comportamiento elástico de la cuerda viene definida cuando  $r_i \approx 500N$ . Encuentra un valor aproximado para  $x_{le}$ .

**Nota:** Hay que buscar  $x_{le}$  empleando código. No vale dibujar los residuos y estimarlo a vista.

**Problema 2.** Dados los siguiente valores de la función  $f(x)$ :

$$f(0) = 1, \quad f\left(\frac{\pi}{4}\right) = \frac{\sqrt{2}}{2}, \quad f\left(\frac{\pi}{2}\right) = 0, \quad f\left(\frac{3\pi}{4}\right) = -\frac{\sqrt{2}}{2}, \quad f(\pi) = -1$$

1. **(1.5 puntos)** Utilizando el comando de Numpy correspondiente, obtener mediante interpolación con splines cúbicos los valores de la función  $f(x)$  sobre cien puntos equiespaciados en el intervalo  $[0, \pi]$ .
2. **(1.5 puntos)** Dibuja mediante un diagrama de barras las diferencias entre los  $f(x)$  y la función  $\cos(x)$  de Python en los mismos puntos del intervalo anterior. Considera que no necesitamos más de dos decimales para la precisión el cálculo de la función coseno; ¿podríamos utilizar la interpolación para  $f(x)$ ?

**Problem 2.** Given the following values belonging to function  $f(x)$ :

1. **(1.5 points)** Using an appropriate Numpy function, compute using cubic splines interpolation the values of function  $f(c)$  at 100 equispaced points into the interval  $[0, \pi]$ .
2. **(1.5 points)** Use a bar plot to draw the differences between  $f(x)$  and the function  $\cos(x)$  in the same points interpolated in the previous question. Suppose we only need up to two decimal precisions to calculate the cosine function. Could we then use the interpolation calculated for  $f(x)$ ?

## Capítulo/Chapter 9

# Diferenciación e Integración numérica

## Numerical differentiation and integration

You never really understand a person until you consider things from his point of view...until you climb into his skin and walk around in it.

Harper Lee, To kill a mokingbird

La diferenciación, y sobre todo la integración son operaciones habituales en cálculo numérico. En muchos casos obtener la expresión analítica de la derivada o la integral de una función puede ser muy complicado o incluso imposible. Además en ocasiones no disponemos de una expresión analítica para la función que necesitamos integrar o derivar, sino tan solo de un conjunto de valores numéricos de la misma. Este es el caso, por ejemplo, cuando estamos trabajando con datos experimentales. Si solo disponemos de valores numéricos, entonces solo podemos calcular la integral o la derivada numéricamente.

Los sistemas físicos se describen generalmente mediante ecuaciones diferenciales. La mayoría de las ecuaciones diferenciales no poseen una solución analítica, siendo posible únicamente obtener soluciones numéricas.

Differentiation and, above all, integration are very common operations in numerical computing. In many cases, obtaining the analytical expression for a function derivative or integral can be very complex or even impossible. Besides, sometimes we have not an analytical expression for the function we need to derive or integrate, but only a set of numerical values of it. This is the case, for example, when we are working with experimental data. If we only have numerical values of the function, we can only compute the derivative or integral numerically.

Physical systems are usually described by differential equations. Most differential equations do not have an analytical solution and thus, we can only obtain numerical solutions for them.

In general, numerical differentiation appro-

En términos generales la diferenciación numérica consiste en aproximar el valor que toma la derivada de una función en un punto. De modo análogo, la integración numérica approxima el valor que toma la integral de una función en un intervalo.

## 9.1. Diferenciación numérica.

Como punto de partida, supondremos que tenemos un conjunto de puntos  $\{x_i, y_i\}$ ,

x	$x_0$	$x_1$	...	$x_n$
y	$y_0$	$y_1$	...	$y_n$

Que pertenecen a una función  $y = f(x)$  que podemos o no conocer analíticamente. El objetivo de la diferenciación numérica es estimar el valor de la derivada  $f'(x)$  de la función, en alguno de los puntos  $x_i$  en los que el valor de  $f(x)$  es conocido.

En general existen dos formas de aproximar la derivada:

1. Derivando el polinomio de interpolación. De este modo, obtenemos un nuevo polinomio que aproxima la derivada.

$$f(x) \approx P_n(x) \Rightarrow f'(x) \approx P'_n(x)$$

2. Estimando la derivada como una fórmula de diferencias finitas obtenida a partir de la aproximación del polinomio de Taylor.

Si partimos de la definición de derivada,

$$f'(x_0) = \lim_{h \rightarrow 0} \frac{f(x_0 + h) - f(x_0)}{h} \approx \frac{f(x_0 + h) - f(x_0)}{h}$$

Podemos asociar esta aproximación con el polinomio de Taylor de primer orden de la función  $f(x)$ ,

$$f(x) \approx f(x_0) + f'(x_0) \cdot (x - x_0) \Rightarrow f'(x_0) \approx \frac{f(x) - f(x_0)}{x - x_0}$$

Si hacemos  $x - x_0 = h$ , ambas expresiones coinciden.

En general, los algoritmos de diferenciación numérica son inestables. Los errores iniciales que puedan contener los datos debido a factores experimentales o al redondeo del or-

ximates the value taken by the derivative of a function at a specific point. Similarly, numerical integration approximates the value taken by the integral of a function in an interval.

## 9.1. Numerical differentiation.

We start taking a set of pairs of points  $\{x_i, y_i\}$ ,

Which belong to a function  $y = f(x)$  whose analytical expression we may know or may not know. The goal of numerical differentiation is to estimate the value of the function derivative  $f'(x)$ , in any of the points  $x_i$  where the function value  $f(x)$  is known.

We can distinguish two different method to approximate the derivative:

1. WE can derivate the polynomial that interpolates the point. So, we get a new polynomial that approximate the derivative.

2. Estimating the derivativa using a finite differences quotient obtained from the Taylor polinomial expansion.

If we start from the definition of derivative,

We can associate this approach with the first-degree Taylor polynomial expansion of functions  $f(x)$

If we take  $x - x_0 = h$  both expressions meet.

In general, the numerical differentiation algorithms are unstable. Initial data errors due to experimental factors or computer rounding off increase with the differentiation process.

denador, aumentan en el proceso de diferenciación. Por eso no se pueden calcular derivadas de orden alto y, los resultados obtenidos de la diferenciación numérica deben tomarse siempre extremando la precaución.

### 9.1.1. Diferenciación numérica basada en el polinomio de interpolación.

El método consiste en derivar el polinomio  $P_n(x)$  de interpolación obtenido por alguno de los métodos estudiados en el capítulo 8 y evaluar el polinomio derivada  $P'_n(x)$  en el punto deseado.

Un ejemplo particularmente sencillo, para la expresión del polinomio derivada se obtiene en el caso de datos equidistantes interpolados mediante el polinomio de Newton-Gregory,

For this reason, it is not possible to compute high-order derivatives, and the results achieved from numerical differentiation must always be considered extremely carefully.

#### 9.1.1.1. Numerical differentiation base on the interpolation polynomial.

The method involves deriving the polynomial  $P_n(x)$  using any of the methods described in chapter 8 and then evaluating the derivative polynomial  $P'_n(x)$  at the desired point.

We can obtain a specially simple expression for the derivative polynomial, in the case of equispaced points interpolated by the newton-Gregory polynomial,

$$p_n(x) = y_0 + \frac{x - x_0}{h} \Delta y_0 + \frac{(x - x_1) \cdot (x - x_0)}{2 \cdot h^2} \Delta^2 y_0 + \cdots + \frac{(x - x_{n-1}) \cdots (x - x_1) \cdot (x - x_0)}{n! \cdot h^n} \Delta^n y_0$$

Si lo derivamos, obtenemos un nuevo polinomio,

And after deriving, we obtain a new polynomial,

$$\begin{aligned} p'_n(x) &= \frac{\Delta y_0}{h} + \frac{\Delta^2 y_0}{2 \cdot h^2} [(x - x_1) + (x - x_0)] + \\ &+ \frac{\Delta^3 y_0}{3! \cdot h^3} [(x - x_1)(x - x_2) + (x - x_0)(x - x_1) + (x - x_0)(x - x_2)] + \cdots + \\ &+ \frac{\Delta^n y_0}{n! \cdot h^n} \sum_{k=0}^{n-1} \frac{(x - x_0)(x - x_1) \cdots (x - x_{n-1})}{x - x_k} \end{aligned}$$

Este polinomio es especialmente simple de evaluar en el punto  $x_0$ ,

To evaluate this polynomial at point  $x_0$  is particularly easy,

$$\begin{aligned} p'_n(x_0) &= \frac{\Delta y_0}{h} + \frac{\Delta^2 y_0}{2 \cdot h^2} \overbrace{(x_0 - x_1)}^{-h} + \cdots + \frac{\Delta^n y_0}{n! \cdot h^n} [\overbrace{(x_0 - x_1)}^{-h} \overbrace{(x_0 - x_2)}^{-2h} \cdots \overbrace{(x_0 - x_{n-1})}^{-(n-1)h}] \\ p'_n(x_0) &= \frac{1}{h} \left( \Delta y_0 - \frac{\Delta^2 y_0}{2} + \frac{\Delta^3 y_0}{3} + \cdots + \frac{\Delta^n y_0}{n} (-1)^{n-1} \right) \end{aligned}$$

Es interesante remarcar como en la expresión anterior, el valor de la derivada se va haciendo más preciso a medida que vamos añadiendo diferencias de orden superior. Si solo conocieramos dos datos,  $(x_0, y_0)$  y  $(x_1, y_1)$ , so-

It is worth notice that the previous expression represent de derivative with increasing precision as we add new higher-order differences. If we would only know two data,  $(x_0, y_0)$  y  $(x_1, y_1)$ , we could only calculate the first-order

lo podríamos calcular la diferencia dividida de primer orden. En este caso nuestro cálculo aproximado de la derivada de  $x_0$  sería,

$$p'_1(x_0) = \frac{1}{h} \Delta y_0$$

Si conocemos tres datos, podríamos calcular  $\Delta^2 y_0$  y añadir un segundo término a nuestra estimación de la derivada,

$$p'_2(x_0) = \frac{1}{h} \left( \Delta y_0 - \frac{\Delta^2 y_0}{2} \right)$$

y así sucesivamente, mejorando cada vez más la precisión.

Veamos como ejemplo el cálculo la derivada en el punto  $x_0 = 0.0$  a partir de la siguiente tabla de datos,

$x_i$	$y_i$	$\Delta y_i$	$\Delta^2 y_i$	$\Delta^3 y_i$	$\Delta^4 y_i$
0.0	0.000	0.203	0.017	0.024	0.020
0.2	0.203	0.220	0.041	0.044	
0.4	0.423	0.261	0.085		
0.6	0.684	0.346			
0.8	1.030				

divided difference. In which case, our approximate derivative computation at  $x_0$  would be,

If we have three data, we could compute  $\Delta^2 y_0$  and add second term to our derivative estimation.

And so on, precision is increasingly improving.

Let see the computing of the derivative at point  $x_0 = 0.0$  using the following data table,

1. Empleando los dos primeros puntos,

$$y'(0,0) = p_1^1(0.0) = \frac{1}{0.2} \cdot 0.203 = 1.015$$

2. Empleando los tres primeros puntos,

$$y'(0,0) = p_2^1(0.0) = \frac{1}{0.2} \left( 0.203 - \frac{0.017}{2} \right) = 0.9725$$

3. Empleando los cuatro primeros puntos,

$$y'(0,0) = p_3^1(0.0) = \frac{1}{0.2} \left( 0.203 - \frac{0.017}{2} + \frac{0.024}{3} \right) = 1.0125$$

4. Empleando los cinco puntos disponibles,

$$y'(0,0) = p_4^1(0.0) = \frac{1}{0.2} \left( 0.203 - \frac{0.017}{2} + \frac{0.024}{3} - \frac{0.020}{4} \right) = 0.9875$$

1. Using the two first points,

2. Using the three first points,

3. Using the four first points,

4. Using the four available points,

### 9.1.2. Diferenciación numérica basada en diferencias finitas

Como se explicó en la introducción, la idea es emplear el desarrollo de Taylor para aproximar la derivada de una función en punto. Si

### 9.1.2. Numerical differentiation based on finite differences

As we explained in the introduction to the chapter, the idea is to take the Taylor's expansion to approximate a function derivative

empezamos con el ejemplo más sencillo, podemos aproximar la derivada suprimiendo de su definición el *paso al límite*,

$$f'(x_k) = \lim_{h \rightarrow 0} \frac{f(x_k + h) - f(x_k)}{h} \approx \frac{f(x_k + h) - f(x_k)}{h}$$

La expresión obtenida, se conoce con el nombre de formula de diferenciación adelantada de dos puntos. El error cometido debido a la elección de un valor de  $h$  finito, se conoce con el nombre de error de truncamiento. Es evidente que desde un punto de vista puramente matemático, la aproximación es mejor cuanto menor es  $h$ . Sin embargo, desde un punto de vista numérico esto no es así. A medida que hacemos más pequeño el valor de  $h$ , aumenta el error de redondeo debido a la aritmética finita del computador. Por tanto, el error cometido es la suma de ambos errores,

at a specific point. If we start with the simplest case, we can approximate the derivative suppressing the *limit computation* in its definition

The expression we have achieved is know as two-point forward difference formula. The error we make due to the finite value we choose for  $h$  is called as truncation error. It is clear that from a purely mathematical point of view, the approximation is better the smaller  $h$  is. However, this is not true from a numerical point of view. As we make smaller the value of  $h$ , the rounding-off error increases due to the computer finite arithmetic. Then we have to take into account the sum of both errors,

$$f'(x) = \frac{f(x + h) - f(x)}{h} + \underbrace{\frac{C \cdot h}{h}}_{\substack{\text{truncating} \\ \text{truncamiento}}} + \underbrace{D \cdot \frac{1}{h}}_{\substack{\text{redondeo} \\ \text{rounding}}}, \quad C \gg D$$

El valor óptimo de  $h$  es aquel que hace mínima la suma del error de redondeo y el error de truncamiento. La figura 9.1 muestra de modo esquemático como domina un error u otro según hacemos crecer o decrecer el valor de  $h$  en torno a su valor óptimo.

Como vimos en la introducción a esta sección, partiendo de el desarrollo de Taylor de una función es posible obtener fórmulas de diferenciación numérica y poder estimar el error cometido. Así por ejemplo, a partir del polinomio de Taylor de primer orden,

The optimal value for  $h$  is that what minimised the rounding-off error plus the truncating error sum. figure 9.1 shows schematically how truncating or rounding-off error dominates as we do grow or decrease the value of  $h$  around its optimal value.

A we sow in the introducction to the present section, departing from a function Taylor's expansion it is possible to arrive to different numerical differentiation formulae and estimate the error made. So, for instance, departing from the fist order Taylor's polynomial,

$$f(x + h) = f(x) + hf'(x) + \frac{h^2}{2}f''(z) \Rightarrow f'(x) = \frac{f(x + h) - f(x)}{h} - \frac{h}{2}f''(z), \quad x < z < x + h$$

El error que se comete debido a la aproximación, es proporcional al tamaño del intervalo empleado  $h$ . La constante de proporcionalidad depende de la derivada segunda de la función,  $f''(z)$  en algún punto indeterminado  $z \in (x, x + h)$ . Para indicar esta relación lineal entre el error cometido y el valor de  $h$ , se dice

The error we make is proportional to the interval  $h$  that we use. The constant of proportionality depends on the second derivative of the function,  $f''(z)$ , at an undetermined point  $z \in (x, x + h)$ . We express this error as having a linear dependency on the value of  $h$ , stating that the error is of order  $h$ , and we represent

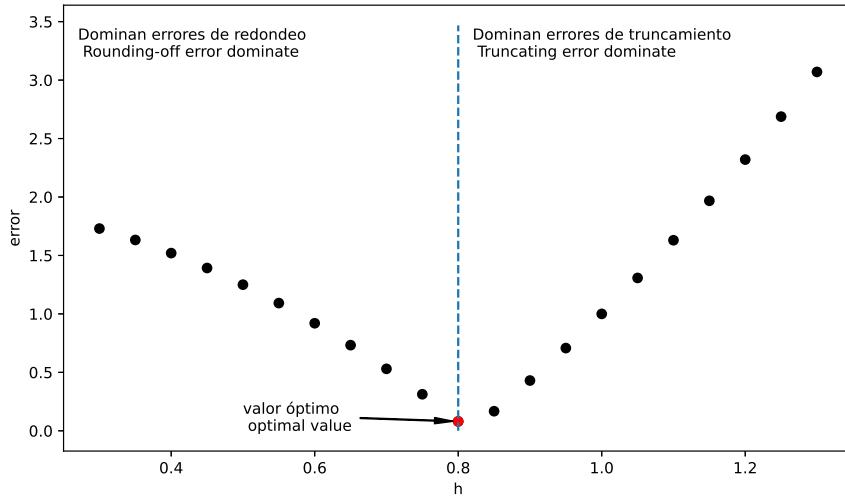


Figura 9.1: Variación del error cometido al aproximar la derivada de una función empleando una fórmula de diferenciación de dos puntos.

Figure 9.1: Function derivative approximation error, using a a two-points differentiation formula. Dependency on the difference  $h$  value

que el error es del *orden* de  $h$  y se representa como  $O(h)$ .

$$f'(x) = \frac{f(x+h) - f(x)}{h} + O(h)$$

Podemos mejorar la aproximación, calculando el valor de polinomio de Taylor de tercer orden para dos puntos equidistantes situados a la izquierda y la derecha del punto  $x$ , restando dichas expresiones y despejando la derivada primera del resultado,

$$\left. \begin{aligned} f(x+h) &= f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{3!}f'''(z) \\ f(x-h) &= f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{3!}f'''(z) \end{aligned} \right\} \Rightarrow f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{6}f'''(z)$$

En esta caso, el error es proporcional al cuadrado de  $h$ , por tanto,

$$f'(x_0) = \frac{f(x_1) - f(x_{-1})}{2h} + O(h^2)$$

Donde hemos hecho  $x \equiv x_0$ ,  $x+h \equiv x_1$  y  $x-h \equiv x_{-1}$ . Esta aproximación recibe el nombre de diferencia de dos puntos centrada. La figura 9.2(a) muestra una comparación entre

it as  $O(h)$ .

To enhance the accuracy, we can calculate the value of the third-order Taylor polynomials at two equally spaced points to the left and right of the point  $x$ . Then we find the difference between these values and solve for the first derivative.

In this case, we get an error proportional to  $h$  square, thus,

Where we have taken  $x \equiv x_0$ ,  $x+h \equiv x_1$  and  $x-h \equiv x_{-1}$ . This approximation is called two point central difference. Figure 9.2(a) shows a comparison between an actual fun-

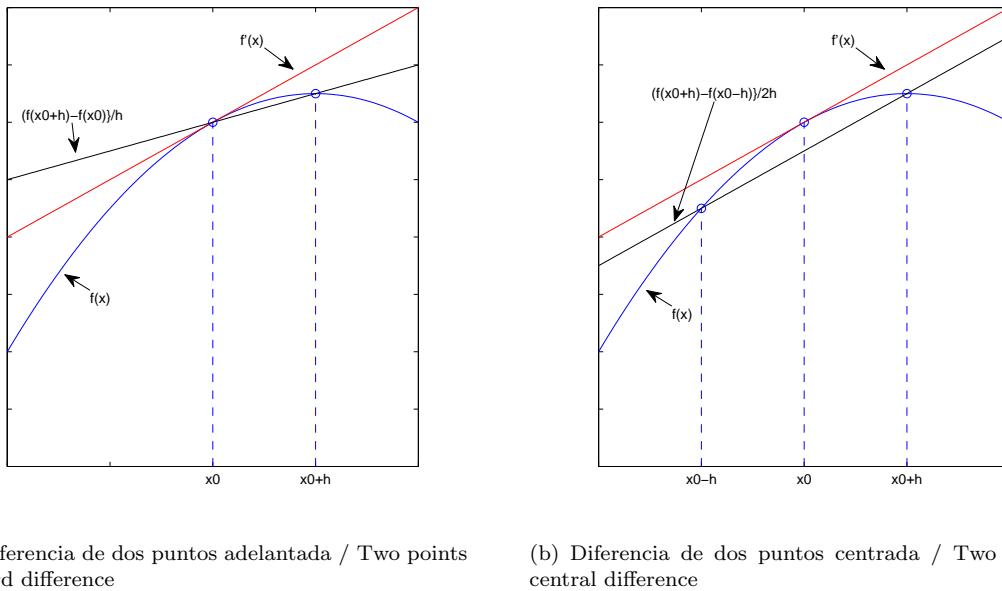


Figura 9.2: Comparación entre las aproximaciones a la derivada de una función obtenidas mediante las diferencias de dos puntos adelantada y centrada.

Figure 9.2: Comparison between the approximations achieved for a function derivative using two point forward difference and two point central difference.

la derivada real de una función y su aproximación mediante una diferencia adelantada de dos puntos. La figura 9.2(b) muestra la misma comparación empleando esta vez la aproximación de dos punto centrada. En este ejemplo es fácil ver como la aproximación centrada da un mejor resultado. No hay que olvidar que la bondad del resultado, para un valor de  $h$  dado, depende también del valor de las derivadas de orden superior de la función, por lo que no es posible asegurar que el resultado de la diferencia centrada sea siempre mejor.

Empleando el desarrollo de Taylor y tres puntos podemos aproximar la derivada por la diferencia de tres puntos adelantada,

ction derivative and its approximation using a two point forward difference formula. Figure 9.2(b) Compares, for the same function, the actual derivative with a two points central difference approximation. In this example, we can see how the central approximation yields better results. However, the accuracy of a result also depends on the higher-order derivative values of the function. Therefore, it is not possible to claim that the central difference will always result in better outcomes.

With three points and the Tylor's expansion we can approximate the derivative using a three points forward difference formula,

$$\left. \begin{aligned} & 4 \cdot \left( f(x_1) = f(x_0) + hf'(x_0) + \frac{h^2}{2}f''(x_0) + \frac{h^3}{3!}f'''(z) \right) \\ & f(x_2) = f(x_0) + 2hf'(x_0) + 2h^2f''(x_0) + \frac{4h^3}{3}f'''(z) \end{aligned} \right\} \Rightarrow$$

$$\Rightarrow f'(x_0) = \frac{-f(x_2) + 4f(x_1) - 3f(x_0)}{2h} - \frac{h^2}{3}f'''(z)$$

En este caso el error es también de orden  $h^2$  pero vale el doble que para la diferencia de dos puntos centrada.

A partir del desarrollo de Taylor y mediante el uso del número de puntos adecuado, es posible obtener aproximaciones a la derivada primera y a las sucesivas derivadas de una función, procediendo de modo análogo a como acaba de mostrarse para el caso de las diferencias de dos y tres puntos. La tabla 9.1 muestra algunas de las fórmulas de derivación mediante diferencias finitas más empleadas. Para simplificar la notación, en todos los casos se ha tomado  $y_i = f(x_i)$ ,  $y_i^{(j)} = f^{(j)}(x_i)$ .

In this case we also get a order  $h^2$  error but its two times the value achieved for the two point central difference.

From Taylor's expansion and using the appropriated number of points, it is possible to compute approximation for the first and successive derivatives of a function. The procedure is similar to the one we employed in the two and three points differences. Table 9.1 shows some derivative approximations using the most common finite differences formulae. To simplify notation, in every case we have taken  $y_i = f(x_i)$ ,  $y_i^{(j)} = f^{(j)}(x_i)$ .

Fórmulas primera derivada First derivative formulae	Fórmulas segunda derivada Second derivative formulae
$y'_0 = \frac{y_1 - y_0}{h} + O(h)$ $y'_0 = \frac{y_1 - y_{-1}}{2h} + O(h^2)$ $y'_0 = \frac{-y_2 + 4y_1 - 3y_0}{2h} + O(h^2)$ $y'_0 = \frac{-y_2 + 8y_1 - 8y_{-1} + y_{-2}}{12h} + O(h^4)$	$y''_0 = \frac{y_2 - 2y_1 + y_0}{h^2} + O(h)$ $y''_0 = \frac{y_1 - 2y_0 + y_{-1}}{h^2}$ $y''_0 = \frac{-y_3 + y_2 - 5y_1 + 2y_0}{h^2} + O(h^2)$ $y''_0 = \frac{-y_2 + 16y_1 - 30y_0 + 16y_{-1} - y_{-2}}{12h^2} + O(h^4)$
Fórmulas tercera derivada Third derivative formulae	Fórmulas cuarta derivada Fourth derivative formulae
$y'''_0 = \frac{y_3 - 3y_2 + 3y_1 - y_0}{h^3} + O(h)$ $y'''_0 = \frac{y_2 - 2y_1 + 2y_{-1} - y_{-2}}{2h^3} + O(h^2)$	$y^{iv}_0 = \frac{y_4 - 4y_3 + 6y_2 - 4y_1 + y_0}{h^4} + O(h)$ $y^{iv}_0 = \frac{y_2 - 4y_1 + 6y_0 - 4y_{-1} + y_{-2}}{h^4} + O(h^2)$

Tabla 9.1: Fórmulas de derivación basadas en diferencias finitas

Table 9.1: Derivative formulae based on finite differences

## 9.2. Integración numérica.

Dada una función arbitraria  $f(x)$  es en muchos casos posible obtener de modo analítico su primitiva  $F(x)$  de modo que  $f(x) = F'(x)$ . En estos casos, la integral definida de  $f(x)$  en un intervalo  $[a, b]$  puede obtenerse directamente a partir de su primitiva,

## 9.2. Numerical Integration.

Given an arbitrary function  $f(x)$  it is in many cases possible to obtain analytically its primitive  $F(x)$  such that,  $f(x) = F'(x)$ . In these cases, the function  $f(x)$  definite integral in an interval  $[a, b]$  can be obtained from its primitive.

$$I(f) = \int_a^b f(x)dx = F(x)|_a^b = F(b) - F(a)$$

Hay sin embargo muchos casos en los cuales se desconoce la función  $F(x)$  y otros en los que ni siquiera se conoce la expresión de la función  $f(x)$ , como por ejemplo, cuando solo se dispone de una tabla de valores  $\{x_i, y_i = f(x_i)\}$  para representar la función. En estos casos se puede aproximar la integral definida de la función  $f(x)$  en un intervalo  $[a, b]$ , a partir de los puntos disponibles, mediante lo que se conoce con el nombre de una fórmula de cuadratura,

$$I(f) = \int_a^b f(x)dx \approx \sum_{i=0}^n A_i f(x_i)$$

Una técnica habitual de obtener los coeficientes  $A_i$ , es hacerlo de modo implícito a partir de la integración de los polinomios de interpolación,

$$I(f) = \int_a^b f(x)dx \approx \int_a^b P_n(x)dx$$

Para ello, se identifican los extremos del intervalo de integración con el primer y el último de los datos disponibles,  $[a, b] \equiv [x_0, x_n]$ .

Así por ejemplo, a partir de los polinomios de Lagrange, definidos en la sección 8.2.2,

$$p(x) = \sum_{j=0}^n l_j(x) \cdot y_j$$

Podemos obtener los coeficientes  $A_i$  como,

$$I(f) = \int_a^b f(x)dx \approx \int_a^b P_n(x)dx = \int_{x_0}^{x_n} \left( \sum_{j=0}^n l_j(x) \cdot y_j \right) dx \Rightarrow A_j = \int_{x_0}^{x_n} l_j(x)dx$$

La familia de métodos de integración, conocidas como fórmulas de Newton-Cotes, pueden obtenerse a partir del polinomio de interpolación de Newton-Gregory descrito en la sección 8.2.4. Supongamos que tenemos la función a integrar definida a partir de un conjunto de puntos equiespaciados a lo largo del intervalo de integración  $\{(x_i, y_i)\}_{0, \dots, n}$ . Podemos aproximar la integral  $I(y)$  como,

$$\int_{x_0}^{x_n} ydx \approx \int_{x_0}^{x_n} \left( y_0 + \frac{x - x_0}{h} \Delta y_0 + \frac{(x - x_0)(x - x_1)}{2!h^2} \Delta^2 y_0 + \dots + \frac{(x - x_0) \cdots (x - x_{n-1})}{n!h^n} \Delta^n y_0 \right)$$

There are many cases for which we do not know function  $F(x)$ , and there are others where we do not even know the function  $f(x)$  as, for example, when we only have a data table  $\{x_i, y_i = f(x_i)\}$  that represents the function. In these cases, it is possible to approximate the definite integral of a function  $f(x)$  in an interval  $[a, b]$  using available points and methods known as quadrature formulae.

A common technique to obtain  $A_i$  coefficients is to use an implicit method by interpolating polynomial integration,

To do this, we first takes the ends of the interval as the first and last data available,  $[a, b] \equiv [x_0, x_n]$ .

So, for example, we using Lagrange's polynomials defined in section 8.2.2,

We can compute the  $A_i$  coefficients as,

The family of numerical integration methods known as Newton-Cotes formulae, can be obtained from the Newton-Gregory's interpolation polynomial described in section 8.2.4. Suppose we have a function defined as a set of  $n + 1$  equispaced points along an integrating interval  $\{(x_i, y_i)\}_{0, \dots, n}$ . We can approximate the integral  $I(y)$  as,

Las fórmulas de Newton-Cotes se asocian con el grado del polinomio de interpolación empleado en su obtención:

1. Para  $n = 1$ , se obtiene la regla del trapezio,

$$I(y) = \int_{x_0}^{x_1} y dx \approx \frac{h}{2}(y_0 + y_1)$$

2. Para  $n = 2$ , se obtiene la regla de Simpson

$$I(y) = \int_{x_0}^{x_2} y dx \approx \frac{h}{3}(y_0 + 4y_1 + y_2)$$

3. Para  $n = 3$ , se obtiene la regla de 3/8 de Simpson

$$I(y) = \int_{x_0}^{x_3} y dx \approx \frac{3h}{8}(y_0 + 3y_1 + 3y_2 + y_3)$$

No se suelen emplear polinomios de interpolación de mayor grado debido a los errores de redondeo y a las oscilaciones locales que dichos polinomios presentan.

### 9.2.1. La fórmula del trapecio.

La fórmula del trapecio emplea tan solo dos puntos para obtener la integral de la función en el intervalo definido por ellos.

$$I(y) = \int_{x_0}^{x_1} y dx \approx \int_{x_0}^{x_1} \left( y_0 + \frac{x - x_0}{h} \Delta y_0 \right) dx = y_0 x + \frac{\Delta y_0}{h} \frac{(x - x_0)^2}{2} \Big|_{x_0}^{x_1} = \frac{h}{2}(y_0 + y_1)$$

La figura 9.3 muestra gráficamente el resultado de aproximar la integral definida de una función  $y = f(x)$  mediante la fórmula del trapecio. Gráficamente la integral coincide con el área del *trapezio* formado por los puntos  $(x_0, 0)$ ,  $(x_0, y_0)$ ,  $(x_1, y_1)$  y  $(x_1, 0)$ . De ahí su nombre y la expresión matemática obtenida,

$$I(y) = \frac{h}{2}(y_0 + y_1),$$

que coincide con el área del trapecio mostrado en la figura.

Newton-Cotes formulae are associated with the degree of the interpolation polynomial employed to obtain the formula:

1. for  $n=1$ , we obtain the Trapezium rule.

2. for  $n = 2$ , we obtain the Simpson's rule.

3. for  $n = 3$ , we obtain the Simpson's 3/8 rule.

We do not usually use interpolation polynomials with larger degrees due to rounding-off errors and the local oscillations of such polynomials.

### 9.2.1. Trapezium formula

The trapezium formula uses only two points to obtain the function integral in the interval defined by the points.

Figure 9.3 shows graphically the results of approximating a function  $y = f(x)$  defined integral using the trapezium rule. The integral fits the area of the *trapezium* formed by points  $(x_0, 0)$ ,  $(x_0, y_0)$ ,  $(x_1, y_1)$  y  $(x_1, 0)$ . This is the reason why the method is known as trapezium formula or trapezium rule and the mathematical expression we get,

which coincides with the expression to compute the area of the trapezium showed in the figure

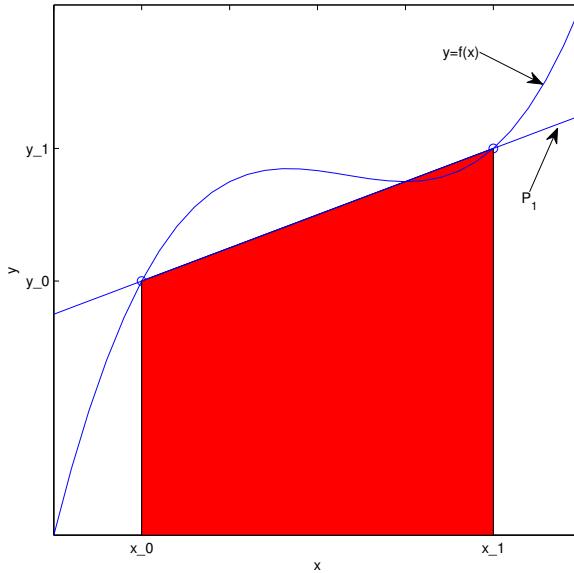


Figura 9.3: Interpretación gráfica de la fórmula del trapecio.

Figure 9.3: Graphical interpretation of trapezium rule.

**Formula extendida (o compuesta) del trapecio.** La figura 9.3 permite observar la diferencia entre el área calculada y el área comprendida entre la curva real y el eje  $x$ . Como se ha aproximado la curva en el intervalo de integración por un línea recta (polinomio de grado 1  $p_1$ ), el error será tanto mayor cuando mayor sea el intervalo de integración y/o la variación de la función en dicho intervalo.

Una solución a este problema, si se conoce la expresión analítica de la función que se desea integrar o se conocen suficientes puntos es subdividir el intervalo de integración en intervalos más pequeños y aplicar a cada uno de ellos la fórmula de trapecio,

**Extended (or composed) trapezium formula.** Figure 9.3 allows us to observe the difference between the area computed and the actual area comprised between the curve and the  $x$  axis. As we have approximated the curve by a straight line (1-degree interpolation polynomial  $p_1$ ) in the integration interval, the larger the integration interval and/or the variation of the function in the interval, the larger the error we make.

One solution to this problem, if we know the analytic expression for the function we wish to integrate or we have enough points, is to divide the integration interval into shorter intervals and apply the trapezium formula in any of them.

$$I(y) = \int_{x_0}^{x_n} y dx \approx \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} y(x) dx = \sum_{i=0}^{n-1} \frac{h}{2} (y_i + y_{i+1}) = \frac{h}{2} (y_0 + 2y_1 + 2y_2 + \dots + 2y_{n-1} + y_n)$$

La figura 9.4, muestra el resultado de aplicar la fórmula extendida del trapecio a la misma función de la figura 9.3. En este caso, se ha dividido el intervalo de integración en cuatro

Figure 9.4, shows the result obtained after applying the extended trapezium formula to the same function of figure 9.3. In the present case, we have divided the integration interval

subintervalos. Es inmediato observar a partir de la figura, que la aproximación mejorará progresivamente si se aumenta el número de subintervalos y se reduce el tamaño de los mismos.

into four subintervals. It is evident from the figure that the approximation will progress as the number of intervals increases and their size decreases.

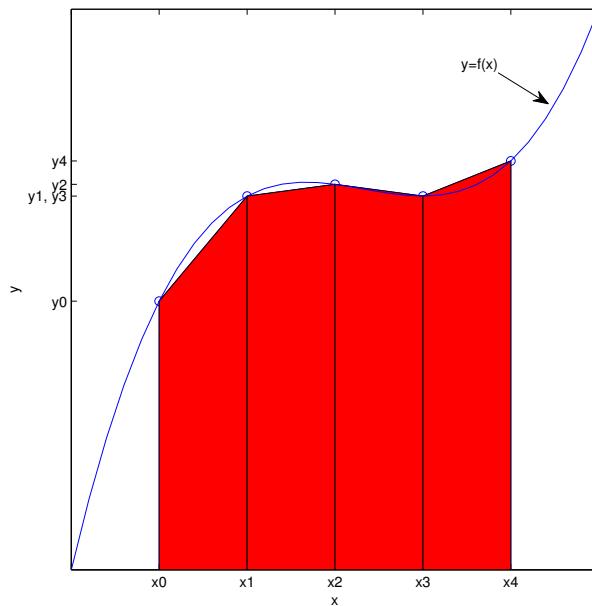


Figura 9.4: Interpretación gráfica de la fórmula extendida del trapecio.

Figure 9.4: Extended trapezium formula graphical interpretation

### 9.2.2. Las fórmulas de Simpson.

Se conocen con el nombre de fórmulas integrales de Simpson, a las aproximaciones a la integral definida obtenida a partir de los polinomios interpoladores de Newton-Gregory de grado dos (Simpson 1/3) y de grado tres (Simpson 3/8).

En el primer caso, es preciso conocer tres valores equiespaciados de la función en el intervalo de integración y en el segundo es preciso conocer cuatro puntos.

**Fórmula de Simpson 1/3.** La fórmula de Simpson, o Simpson 1/3, emplea un polinomio de interpolación de Newton-Gregory de grado dos para obtener la aproximación a la integral,

$$I(y) \approx \int_{x_0}^{x_2} P_2(x) dx = \int_{x_0}^{x_2} \left( y_0 + \frac{x - x_0}{h} \Delta y_0 + \frac{(x - x_0) \cdot (x - x_1)}{2h^2} \Delta^2 y_0 \right) dx = \frac{h}{3} (y_0 + 4y_1 + y_2)$$

### 9.2.2. Simpson's formulae

The integral Simpson's formulae (or rules) are two approximations to the definite integral of a function, obtained from the Newton-Gregory's interpolation polynomials of degree two (Simpson 1/3) and degree 3 (Simpson 3/8).

In the first case we need to know three equispaced values of the function in the integration interval. In the second case we need to know four points.

**Simpson's 1/3 formula.** The Simpson's rule or Simpson's 1/3 rule employs a 2-degree Newton-Gregory's interpolation polynomial to obtain the approximation of the integral,

La figura 9.5, muestra gráficamente el resultado de aplicar el método de Simpson a la misma función de los ejemplos anteriores. De nuevo, la bondad de la aproximación depende de lo que varíe la función en el intervalo. La diferencia fundamental con el método del trapecio es que ahora el área calculada esta limitada por el segmento de parábola definido por el polinomio de interpolación empleado.

Figure, 9.5 shows graphically the result of applying the Simpson's method to the same function used in previous examples. Again, the goodness of the approximation depends on the function variation in the integration interval. The main difference with the trapezium method is that the area is now bounded by the parabolic segment defined by the polynomial we have utilized.

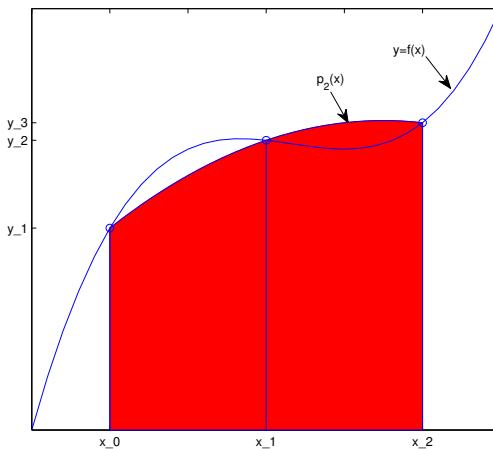


Figura 9.5: Interpretación gráfica de la fórmula 1/3 de Simpson.

Figure 9.5: Graphical interpretation of Simpson's 1/3 formula

**Fórmula de Simpson 3/8.** En este caso, se emplea un polinomio de Newton-Gregory de grado 3 para obtener la aproximación a la integral,

**Simpson's 3/8 Formula.** In this case, we use a Newton-Gregory's polynomial of degree 3 to obtain the approximation for the integral.

$$\begin{aligned}
 I(y) &\approx \int_{x_0}^{x_3} P_2(x) dx = \\
 &= \int_{x_0}^{x_3} \left( y_0 + \frac{x - x_0}{h} \Delta y_0 + \frac{(x - x_0) \cdot (x - x_1)}{2h^2} \Delta^2 y_0 + \frac{(x - x_0) \cdot (x - x_1) \cdot (x - x_2)}{3!h^3} \Delta^3 y_0 \right) dx \\
 &= \frac{3h}{8} (y_0 + 3y_1 + 3y_2 + y_3)
 \end{aligned}$$

La figura 9.6 muestra el resultado de aplicar la fórmula de Simpson 3/8 a la misma función de los ejemplos anteriores. En este caso, la integral sería exacta porque la función de ejemplo elegida es un polinomio de tercer gra-

Figure 9.6 shows the result of applying Simpson's 3/8 formula to the same function utilized in previous examples. In this case, the integral is exact because the example function is a three degree polynomial and fit exactly

do y coincide exactamente con el polinomio de interpolación construido para obtener la integral.

the interpolating polynomial we have built to compute the integral.

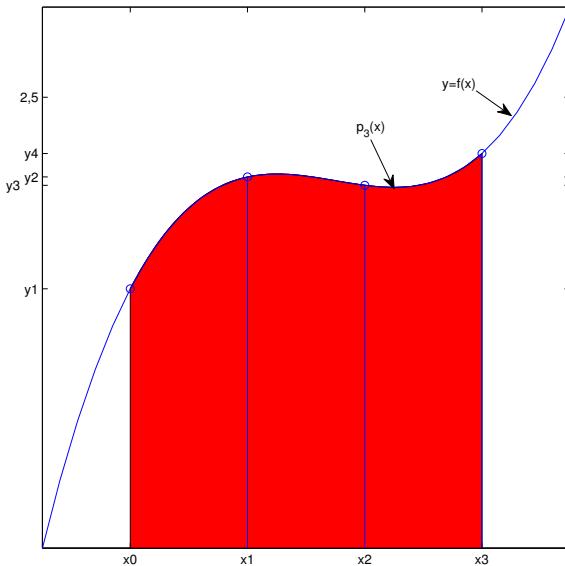


Figura 9.6: Interpretación gráfica de la fórmula 3/8 de Simpson.

Figure 9.6: Graphical interpretation of Simpson's 3/8 formula

Al igual que en el caso del método del trapecio, lo normal no es aplicar los métodos de Simpson a todo el intervalo de integración, sino dividirlo en subintervalos más pequeños y aplicar el método sobre dichos subintervalos. El resultado se conoce como métodos extendidos de Simpson. Al igual que sucede con la fórmula del trapecio, los métodos extendidos de Simpson mejoran la aproximación obtenida para la integral tanto más cuanto más pequeño es el tamaño de los subintervalos empleados.

Así, la fórmula extendida de Simpson 1/3 toma la forma,

Like the trapezium formula, we hardly ever apply Simpson's methods to the whole integration interval. Instead, we divide the integration interval into smaller subintervals and apply the method to such subintervals. The results are known as extended Simpson's methods. As in the case of the trapezium formula, the extended Simpson's methods improve the approximation computed for the integral as much as the smaller the size of the subinterval we use.

So the extended Simpson 1/3 formula takes the form,

$$\begin{aligned} I(y) &\approx \sum_{i=0}^{\frac{n}{2}-1} \int_{x_{2i}}^{x_{2i+2}} P_2(x) dx = \sum_{i=0}^{\frac{n}{2}-1} \frac{h}{3} (y_{2i} + 4y_{2i+1} + y_{2i+2}) \\ &= \frac{h}{3} (y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + 2y_{n-2} + 4y_{n-1} + y_n) \end{aligned}$$

Donde se ha dividido el intervalo de integración en  $n$  subintervalos, la fórmula de Simpson se ha calculado para cada dos subintervales y se han sumado los resultados.

Por último para la fórmula extendida de Simpson 3/8 se puede emplear la expresión,

$$\begin{aligned} I(y) &\approx \sum_{i=0}^{\frac{n}{3}-1} \int_{x_{3i}}^{x_{3i+3}} P_3(x) dx = \sum_{i=0}^{\frac{n}{3}-1} \frac{3h}{8}(y_{3i} + 3y_{3i+1} + 3y_{3i+2} + y_{3i+3}) \\ &= \frac{3h}{8}(y_0 + 3y_1 + 3y_2 + 2y_3 + 3y_4 + 3y_5 + 2y_6 + \cdots + 2y_{n-3} + 3y_{n-2} + 3y_{n-1} + y_n) \end{aligned}$$

En este caso también se divide el intervalo en  $n$  subintervalos pero ahora se ha aplicado la regla de Simpson 3/8 a cada tres subintervalos.

A continuación se incluye un código que permite permitir aproximar la integral definida de una función en un intervalo por cualquiera de los tres métodos descritos: Trapecio, Simpson o Simpson 3/8,

Where we have divided the integration interval in  $n$  subintervals, the Simpson's formula has been applied to every two consecutive subintervals and, eventually we have summed the results together.

In this case, we have also divided the interval in  $n$  subintervals but now we apply the Simpson's 3/8 rule to each group of three successive subintervals.

The code included below compute an approximation to de definite integral of a function using any one the three method we have described: Trapezium, Simpson's or Simpson's 3/8,

---

integrador.py

---

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Sun Aug 18 16:21:54 2024
4
5 @author: abierto
6 """
7 import numpy as np
8 from matplotlib import pyplot as pl
9 def integra(fun,met,inter,dib=False):
10 """
11     function numerical integration, using trapezium, Simpson's
12     1/3 or Simpson 3/8 methods
13
14     Parameters
15     -----
16     fun : TYPE function
17         DESCRIPTION. Function to be integrated
18     met : TYPE char array
19         DESCRIPTION. method should be one of 'trap' 'simpson'
20         or 'simpson38'
21     inter : TYPE array [a,b]
22         DESCRIPTION. integration interval limits
23     dib : TYPE logical
24         DESCRIPTION. yes : draw the results no: skip the drawing
25
26     Returns
27     -----

```

```

28     intg: TYPE double
29         DESCRIPTION. Integral value
30         f1 = fun(inter[1])
31         h = inter[1] - inter[0]
32
33         """
34         if met == 'trap':
35             f0 = fun(inter[0])
36             f1 = fun(inter[1])
37             h = inter[1] - inter[0]
38             intg = h*(f0+f1)/2
39             if dib:
40                 x = np.linspace(inter[0],inter[1],100)
41                 y = np.array([fun(i) for i in x])
42                 ypol = f0 + (x - inter[0])*(f1-f0)/h
43                 pl.fill_between(x,ypol,\n
44                                 where=(inter[0]<=x)&(x<=inter[1]))
45                 pl.plot(x,y,'b')
46             elif met == 'simpson':
47                 pmedio = (inter[1] + inter[0])/2
48                 f0 = fun(inter[0])
49                 f1 = fun(pmedio)
50                 f2 = fun(inter[1])
51                 h = (inter[1] - inter[0])/2
52                 intg = h*(f0+4*f1+f2)/3
53                 if dib:
54                     x = np.linspace(inter[0],inter[1],100)
55                     y = np.array([fun(i) for i in x])
56                     ypol = f0+(x-inter[0])*(f1-f0)/h\n
57                     +(x-inter[0])*(x-pmedio)*(f2-2*f1+f0)/(2*h**2)
58                     pl.fill_between(x,ypol,\n
59                                     where=(inter[0]<=x)&(x<=inter[1]))
60                     pl.plot(x,ypol,'k')
61                     pl.plot(x,y,'b')
62             elif met == 'simpson38':
63                 inter = np.linspace(inter[0],inter[1],4)
64                 f = np.array([fun(i) for i in inter])
65                 h = inter[1]-inter[0]
66                 intg = 3*h*(f[0]+3*f[1]+3*f[2]+f[3])/8
67                 if dib:
68                     x = np.linspace(inter[0],inter[3],100)
69                     y = np.array([fun(i) for i in x])
70                     ypol = f[0]+(x-inter[0])*(f[1]-f[0])/h\n
71                     +(x-inter[0])*(x-inter[1])\n
72                     +(f[2]-2*f[1]+f[0])/(2*h**2)\n
73                     +(x-inter[0])*(x-inter[1])*(x-inter[2])\n
74                     +(f[3]-3*f[2]+3*f[1]-f[0])/(6*h**3)
75                     pl.fill_between(x,ypol,\n
76                                     where=(inter[0]<=x)&(x<=inter[3]))
77                     pl.plot(x,ypol,'k')
78                     pl.plot(x,y,'b')
79             return(intg)

```

```

80
81 def trocea(fun,met,inter,div,dib=False):
82     """
83     This function divide a interval into the number of subintervals indicate
84     by div and calls function integra to obtain the integral in
85     any subinterval. Eventually sums all results to compute the
86     integral over the whole interval
87
88     Parameters
89     -----
90     fun : TYPE function
91         DESCRIPTION. function to be integrate
92     met : TYPE character string
93         DESCRIPTION. Method it should be one of 'trap' 'simpson'
94         or 'simpson38' can be 'trap',
95     inter : TYPE array or list
96         DESCRIPTION. limits for the integration interval [a,b]
97     div : TYPE integer
98         DESCRIPTION. Number of subintervals
99     dib : TYPE bool
100        DESCRIPTION. True draw the result False does not draw
101
102     Returns
103     -----
104     total : TYPE double
105         DESCRIPTION Definite integral value of fun in inter
106
107     """
108     tramos = np.linspace(inter[0],inter[1],div+1)
109     total = 0
110     for i in np.arange(div):
111         int = integra(fun,met,[tramos[i],tramos[i+1]],dib)
112         total = total + int
113     return(total)

```

---

Este programa contiene dos funciones. La primera `integra` aplica directamente el método deseado sobre el intervalo de integración. Para obtener los métodos extendidos, podemos emplear la segunda función `trocea` que divide el intervalo de integración inicial en el número de subintervalos que deseemos, aplica el programa anterior a cada subintervalo, y, por último, suma todo los resultados para obtener el valor de la integral en el intervalo deseado.

Por último, indicar que el modulo de Python, Scipy posee varias funciones para calcular la integral definida de una función. Se encuentran dentro del submódulo `integrate`. Para calcular la integral definida de una fun-

This program holds two functions. The first one `integra` applies the method directly over the integration interval. To obtain the extended methods, we can employ the second included function `trocea`, which divides the initial integration interval into the number of subintervals we wish, applies the function `integra` to any subinterval and, eventually, add all the results together to obtain the value of the integral in the wished interval.

Lastly, the Python module Scipy has several functions to compute a function's definite integral. It is located in the submodule `integrate`. The easiest way To compute the integral of a function is to use the function `integrate.quad`. This function takes a fun-

ción lo más sencillo es usar el comando `integrate.quad`. Este comando admite como variables de entrada el nombre de una función, y dos valores que representan los límites de integración. Como variable de salida devuelve el valor de la integral definida en el intervalo introducido y una estimación del error cometido. El siguiente código muestra un ejemplo de uso de la función `quad` y compara los resultados con la función `trocea` cuyo código hemos incluido más arriba.

```
In [56]: import scipy.integrate as integ

In [57]: def probando(x):
....:     """test function to be integrate"""
....:     return((x-1)*x*(x-2))+6
....:

In [58]: inte.quad(probando,-5,2.5)
Out[58]: (-260.8593749999994, 3.3240810849038596e-12)

In [59]: trocea(probando,'simpson',[ -5,2.5],5)
Out[59]: -260.859375
```

### 9.3. Problemas de valor inicial en ecuaciones diferenciales

Las leyes de la física están escritas en forma de ecuaciones diferenciales.

Una ecuación diferencial establece una relación matemática entre una variable y sus derivadas respecto a otra u otras variables de las que depende. El ejemplo más sencillo lo encontramos en las ecuaciones de la dinámica en una sola dimensión que relacionan la derivada segunda de la posición de un cuerpo con respecto al tiempo, con la fuerza que actúa sobre el mismo.

$$m \cdot \frac{d^2x}{dt^2} = F$$

Si la fuerza es constante, o conocemos explícitamente como varía con el tiempo, podemos integrar la ecuación anterior para obtener la derivada primera de la posición con respecto

tion name and two variables which represent the integration limits as inputs and returns the integral value in the interval introduced and an estimation of the committed error. The following code shows a use example of the function `quad` and compares the results with those of the function `trocea` whose code we have included above.

### 9.3. Differential equations. Initial value problems

The laws of physics are written as differential equations

Differential equations establish a mathematical relationship between a variable and its derivatives with respect to another variable or other variables the first variable depends upon. We can find the simplest example watching the one dimensional dynamics equations which link the second derivative of a body position with the force acting on the body.

If the force applied is constant or we know explicitly how it evolves on time, we can integrate the previous equation to straightforwardly obtain the position first derivative with

al tiempo —la velocidad— de una forma directa,

$$m \cdot \frac{d^2x}{dt^2} = F \rightarrow v(t) = \frac{dx}{dt} = \int \frac{F(t)}{m} dt + v(0)$$

Donde suponemos conocido el valor  $v(0)$  de la velocidad del cuerpo en el instante inicial.

Si volvemos a integrar ahora la expresión obtenida para la velocidad, obtendríamos la posición en función del tiempo,

$$v(t) = \frac{dx}{dt} = \int \frac{F(t)}{m} dt + v(0) \rightarrow x(t) = \int \left( \int \frac{F(t)}{m} dt + v(0) \right) dt + x(0)$$

Donde suponemos conocida la posición inicial  $x(0)$ .

Quizá el sistema físico idealizado más conocido y estudiado es el oscilador armónico. En este caso, el sistema está sometido a una fuerza que depende de la posición y, si existe disipación, a una fuerza que depende de la velocidad,

$$m \frac{d^2x}{dt^2} = -kx - \mu \frac{dx}{dt}$$

En este caso, la expresión obtenida constituye una ecuación diferencial ordinaria y ya no es tan sencillo obtener una expresión analítica para  $x(t)$ . Para obtener dicha expresión analítica, es preciso emplear métodos de resolución de ecuaciones diferenciales.

El problema del oscilador armónico, pertenece a una familia de problemas conocida como problemas de valores iniciales. En general, un problema de valores iniciales de primer orden consiste en obtener la función  $x(t)$ , que satisface la ecuación,

$$x'(t) \equiv \frac{dx}{dt} = f(x(t), t), \quad x(t_0)$$

Donde  $x(t_0)$  representa un valor inicial conocido de la función  $x(t)$ .

En muchos casos, las ecuaciones diferenciales que describen los fenómenos físicos no admiten una solución analítica, es decir no permiten obtener una función para  $x(t)$ . En estos casos, es posible obtener soluciones numéricas empleando un computador. El problema de valores iniciales se reduce entonces a encon-

respect to time —the velocity—,

When we consider we know the body velocity  $v(0)$  at the initial time.

If we now integrate the expression we have got for the velocity, we obtain the position as a function of time,

Were we suppose the initial position  $x(0)$  is known.

Perhaps the better known and most studied physical idealised system is the harmonic oscillator. In this case, the system suffers a force that depends on the position and, if the oscillator is damped, it suffer also a force that depends on the speed,

In this last case, the expression we obtain is an ordinary differential equation (ODE), and it is no longer so easy to solve and obtain an analytical expression for  $x(t)$ . To obtain such a expression we need to use specific methods to solve differential equations

The harmonic oscillator problem belongs to a problems family known as initial value problems. In general, to solve a first order initial value problem is to obtain the function  $x(t)$ , that satisfies the equation,

Where  $x(t_0)$  represents a known initial value of the function  $x(t)$ .

In many cases, differential equations describe physical phenomena that not have an analytical solution. That is, there is no analytical function  $x(t)$  that satisfies the differential equation. In these cases is still possible to compute numerical solutions using a computer. The initial value problems reduces then to

trar un aproximación discretizada de la función  $x(t)$ .

El desarrollo de técnicas de integración numérica de ecuaciones diferenciales constituye uno de los campos de trabajo más importantes de los métodos de computación científica. Aquí nos limitaremos a ver los más sencillos.

Esencialmente, los métodos que vamos a describir se basan en discretizar el dominio donde se quiere conocer el valor de la función  $x(t)$ . Así por ejemplo si se quiere conocer el valor que toma la función en el intervalo  $t \in [a, b]$ , se divide el intervalo en  $n$  subintervalos cada uno de tamaño  $h_i$ . Los métodos que vamos a estudiar nos proporcionan una aproximación de la función  $x(t)$ ,  $x_0, x_2 \dots x_n$  en los  $n + 1$  puntos  $t_0, t_1, \dots, t_n$ , donde  $t_0 = a$ ,  $t_n = b$ , y  $t_{i+1} - t_i = h_i$ . El valor de  $h_i$  recibe el nombre de paso de integración. Además se supone conocido el valor que toma la función  $x(t)$  en el extremo inicial  $a$ ,  $x(a) = x_a$ .

### 9.3.1. El método de Euler.

El método de Euler, puede obtenerse a partir del desarrollo de Taylor de la función  $x$ , entorno al valor conocido  $(a, x_a)$ . La idea es empezar en el valor conocido e ir obteniendo iterativamente el resto de los valores  $x_1, \dots$  hasta llegar al extremo  $b$  del intervalo en que queremos conocer el valor de la función  $x$ . En general podemos expresar la relación entre dos valores sucesivos a partir del desarrollo de Taylor como,

$$x(t_{i+1}) = x(t_i) + (t_{i+1} - t_i)x'(t_i) + \frac{(t_{i+1} - t_i)^2}{2}x''(t_i) + \dots + \frac{(t_{i+1} - t_i)^n}{n!}x^{(n)}(t_i) + \dots$$

Como se trata de un problema de condiciones iniciales, conocemos la derivada primera de la función  $x(t)$ , explícitamente,  $x'(t) = f(x(t), t)$ . Por tanto podemos sustituir las derivadas de  $x$  por la función  $f$  y sus derivadas con respecto a  $t$ ,

$$x(t_{i+1}) = x(t_i) + (t_{i+1} - t_i)f(t_i, x_i) + \frac{(t_{i+1} - t_i)^2}{2}f'(t_i, x_i) + \dots + \frac{(t_{i+1} - t_i)^n}{n!}f^{(n-1)}(t_i, x_i) + \dots$$

find a discretised version of the function  $x(t)$ .

Developing techniques to numerically solve differential equations is perhaps one of the most important fields of scientific computing. Here, we will describe only the most basic methods.

The methods we are going to introduce are essentially based on discretising the domain in which we want to know the value of the function  $x(t)$ . For instance, if we want to know the values of the function in the interval  $t \in [a, b]$ , we divide the interval in  $n$  subintervals, each one of size  $h_i$ . The methods we are going to study give us an approximation of the function  $x(t)$ ,  $x_0, x_2 \dots x_n$  in the  $n + 1$  points  $t_0, t_1, \dots, t_n$ , where  $t_0 = a$ ,  $t_n = b$ , and  $t_{i+1} - t_i = h_i$ . The value  $h_i$  is known as the integration step. Besides, we suppose that we know the value the function  $x(t)$  takes in the interval initial side  $a$ ,  $x(a) = x_a$ .

#### 9.3.1.1. The Euler's method.

The Euler's method can be obtained from the Taylor's expansion of the function  $x$ , around the known value  $(a, x_a)$ . The idea is to start at the known value and compute iteratively the remaining values  $x_1, \dots$  until we arrive to the end of the interval  $b$ , at which we want to know the value function  $x$  takes.

We can obtain the relationship between two consecutive values of the function from the Taylor's expansion as,

As we are solving an initial condition problem, we know first the derivative of function  $x(t)$  at the initial point  $x'(t) = f(x(t), t)$ . Therefore, we may substitute the derivatives of  $x$  by the function  $f$  and its derivatives with respect to  $t$ .

Donde  $x_i \equiv x(t_i)$ . Si truncamos el polinomio de Taylor, quedándonos solo con el término de primer grado, y hacemos que el paso de integración sea fijo,  $h_i \equiv h = \text{cte}$  obtenemos el método de Euler,

$$x_{i+1} = x_i + h \cdot f(t_i, x_i)$$

A partir de un valor inicial,  $x_0$  es posible obtener valores sucesivos mediante un algoritmo iterativo simple,

Where  $x_i \equiv x(t_i)$ . If we truncate the Taylor's polynomial, leaving only the first order term and set the integration step to a constant value,  $h_i \equiv h = \text{cte}$  we arrive to the Euler's method,

Departing from a known initial value,  $x_0$ , we can obtain the successive values using a simple iterative algorithm,

$$\begin{aligned} x_0 &= x(a) \\ x_1 &= x_0 + hf(a, x_0) \\ x_2 &= x_1 + hf(a + h, x_1) \\ &\vdots \\ x_{i+1} &= x_i + hf(a + ih, x_i) \end{aligned}$$

El siguiente código implementa el método de Euler para resolver un problema de condiciones iniciales de primer orden a partir de una función  $f(t)$  y un valor inicial  $x_0$ .

The following code implements the Euler's method using a function  $f(t)$  and an initial value  $x_0$  as inputs,

---

Euler\_method.py

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Wed Aug 21 11:08:42 2024
4 Euler's method for solving initial value problems
5 @author: abierto
6 """
7 import numpy as np
8 import matplotlib.pyplot as pl
9
10 def euler(fun,xa,a,b,h):
11     """
12         This function solve numerically the problem dx/dt = f(t,x)
13         form the intial condition xa along the interval [a,b] and
14         using an integration step h
15
16     Parameters
17     -----
18     fun : TYPE function
19         DESCRIPTION. it is the funtion that describe the differential
20         equation to be solved it should be a funtion of t and x
21         f(t,x) also is there is not explicit dependence on t.
22     xa : TYPE numpy array
23         DESCRIPTION. initial condition
24     a : TYPE double
25         DESCRIPTION. initial value of t
26     b : TYPE double

```

```

27     DESCRIPTION. final value of t
28 h : TYPE double
29     DESCRIPTION. integration step
30
31 Returns
32 -----
33 x : TYPE numpy array
34     DESCRIPTION. initial value  $dx/dt = f(x)$  solution at points
35     a, a+h a+2h ... b
36 t. TYPE numpy array
37     DESCRIPTION. times x has been computed at
38 """
39 if a >= b:
40     raise ValueError('I need an increasing interval')
41
42 #to arrive to the final condition, we modify slightly the
43 #integration step. First we calculate the approximate number
44 #of intervals between a and b of size h and round it towards
45 #infinite.
46 pt = int(np.ceil((b-a)/h))
47 #then, we modify the integration step to acomodate at
48 #the rounded number of interval. This is, of course, not
49 #necessary to implement the method, there are other
50 #possibilities to deal with the integration step and
51 #the limit b
52 hft = (b-a)/pt
53 t = np.arange(a,b+hft,hft)
54 x = np.zeros([xa.shape[0],pt+1])
55 x[:,0] = xa
56 for i in range(1,pt+1):
57     x[:,i] = x[:,i-1] + hft*fun(t[i-1],x[:,i-1])
58 pl.plot(t,x.T)
59 return(t,x)
60
61
```

---

Un ejemplo sencillo de problema de condiciones iniciales de primer orden, nos los suministra la ecuación diferencial de la carga y descarga de un condensador eléctrico. La figura 9.7, muestra un circuito eléctrico elemental formado por una resistencia  $R$  en serie con un condensador  $C$ .

La intensidad eléctrica que atraviesa un condensador depende de su capacidad, y de la variación con respecto al tiempo del voltaje entre sus extremos,

The differential equation that models the charge and discharge of an electronic capacitor supplies a simple example of an first order initial condition problem. Figure 9.7 shows a electronic circuit composed of a resistor  $R$  in series with a capacitor  $C$ .

The electric current that passes through a capacitor depends on the capacitor capacity and on the variation on time of the voltage applied to it.

$$I = c \frac{dV_o}{dt}$$

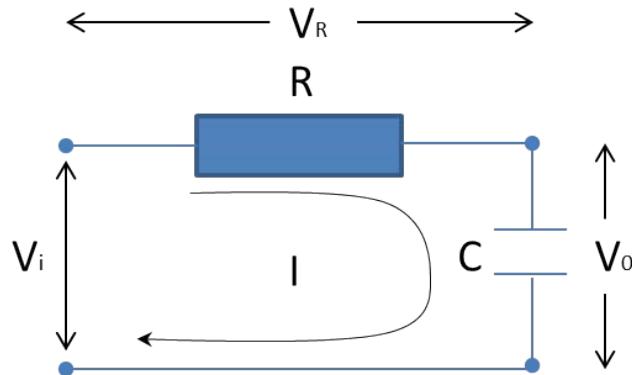


Figura 9.7: Circuito RC

La intensidad que atraviesa la resistencia se puede obtener a partir de la ley de Ohm,

the current through the resistor can be computed using Ohm's law.

$$V_R = I \cdot R$$

Por otro lado, la intensidad que recorre el circuito es común para la resistencia y el condensador. El voltaje suministrado tiene que ser la suma de las caídas de voltaje en la resistencia y en el condensador,  $V_i = V_o + V_R$ . Sustituyendo y despejando,

Additionally, the current flowing through the circuit is the same for both the resistor and the capacitor. The voltage supplied to the circuit should equal the sum of the resistor and capacitor voltage drop, expressed as  $V_i = V_o + V_R$ . By substituting and solving,

$$V_i = V_o + V_R \rightarrow V_i = V_o + I \cdot R \rightarrow V_i = V_o + R \cdot C \frac{dV_o}{dt}$$

Si reordenamos el resultado,

And, after rearranging the result,

$$\frac{dV_o}{dt} = \frac{V_i - V_o}{R \cdot C}$$

Obtenemos una ecuación diferencial para el valor del voltaje en los extremos del condensador que puede tratarse como un problema de valor inicial. Para este problema, la función  $f(t, x)$  toma la forma,

We get a differential equation for the capacitor tension drop that could be solved as an initial value problem. For this problem the function  $f(t, x)$  takes the form,

$$f(t, V_0 \equiv x) = \frac{V_i - V_o}{R \cdot C}$$

Además necesitamos conocer un valor inicial  $V_o(0)$  para el voltaje en el condensador. Si suponemos que el condensador se encuentra inicialmente descargado, entonces  $V_o(0) = 0$ . Para este problema se conoce la solución analítica. El voltaje del condensador en función del tiempo viene dado por la función,

In addition, we need to know an initial value  $V_o(0)$  for the voltage across the capacitor. If we consider the capacitor is initially discharged, then  $V_o(0) = 0$ . For this problem we know the analytic solution. We can express the voltage across the capacitor as a function of time,

$$V_o(t) = V_i \left( 1 - e^{-t/R \cdot C} \right)$$

Podemos resolver el problema de la carga del condensador, empleando la función `Euler` incluida más arriba. Lo único que necesitamos es definir una función en Python para representar la función  $f(x, t)$  de nuestro problema de valor inicial,

```
In [151]: def condensador(t,Vo,C=1.,R=1.,Vi=10.):
....:     dVo = (Vi-Vo)/(R*C)
....:     return(dVo)
```

We can solve the capacitor problem using the function `euler` included above. WE only need to define a Python funtion to represent the funtion  $f(x, t)$  for our initial value problem,

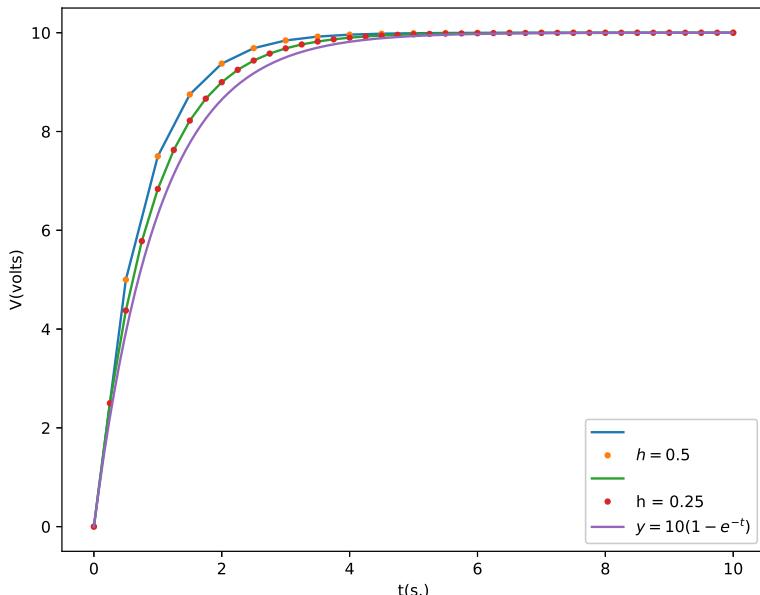


Figura 9.8: Comparacion entre los resultados obtenidos mediante el método de Euler para dos pasos de integración  $h = 0.5$  y  $h = 0.25$  y la solución analítica para el voltaje  $V_o$  de un condensador durante su carga. Hemos tomado  $C = R = 1$  y  $V_i = 10$

Figure 9.8: Comparison among the results obtained using Euler's method with two different integration steps  $h = 0.5$  y  $h = 0.25$ , and the analytic solution for voltage  $V_o$  across a capacitor. We took  $C = R = 1$  and  $V_i = 10$ .

La figura 9.8 compara gráficamente los resultados obtenidos si empleamos la función `euler(condensador, 0, 0, 10, h)`, para obtener el resultado de aplicar al circuito un voltaje de entrada constante  $V_i = 10V$  durante 10 segundos, empleando dos pasos de integración distintos  $h = 0.5s$  y  $h = 0.025s$ . Además, se ha añadido a la gráfica el resultado analítico.

Figure 9.8 graphically compares the results achieved using the function `euler(condensador, 0, 0, 10, h)` to compute the voltage across the capacitor after supplying a constant input voltage  $V_i = 10V$  during ten seconds, using two different integration steps,  $h = 0.5s$  ans  $h = 0.25s$ . We have also added the analytical solution to the problem to the

En primer lugar, es interesante observar como el voltaje  $V_0$  crece hasta alcanzar el valor  $V_i = 10$  V del voltaje suministrado al circuito. El tiempo que tarda el condensador en alcanzar dicho voltaje y quedar completamente cargado depende de su capacidad y de la resistencia presente en el circuito.

Como era de esperar, al hacer menor el paso de integración la solución numérica se aproxima más a la solución analítica. Sin embargo, como pasaba en el caso de los métodos de diferenciación de funciones, hay un valor de  $h$  óptimo. Si disminuimos el paso de integración por debajo de ese valor, los errores de redondeo empiezan a dominar haciendo que la solución empeore.

**Problema de segundo orden.** Vamos a considerar ahora un sistema con una masa colgada de un resorte con un dispositivo mecánico (amortiguador) que ejerce una fuerza opuesta al movimiento proporcional a la velocidad. En este caso la ecuación diferencial sería la siguiente:

$$m \frac{d^2y}{dt^2} = m \cdot g - k \cdot y - \mu \frac{dy}{dt}$$

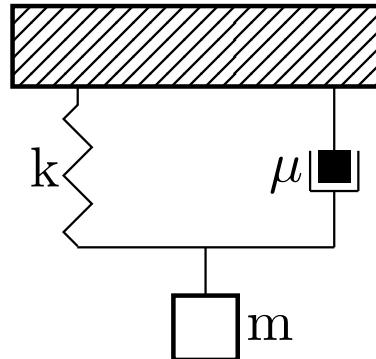


Figura 9.9: Masa suspendida en vertical de un muelle con rozamiento.  
Figure 9.9: A mass vertically hanging of a sprint with a damping resistance

Tenemos un problema de valor inicial de segundo orden, con una ecuación en la que conviven la variable  $y$  su derivada  $\frac{dy}{dt}$  y su derivada segunda  $\frac{d^2y}{dt^2}$ . Para resolver este tipo de problemas lo que hacemos es reescribir

graphic.

It's interesting to observe how the voltage, denoted as  $V_0$ , increases until it reaches the value  $V_i = 10V$ , which corresponds to the voltage supplied to the circuit. The time it takes for the capacitor to reach this voltage and become fully charged depends on its capacitance and the value of the resistor in the circuit.

As expected, when we diminish the integration step, the numerical solution approximates the analytical solution better. However, as we saw for the case of numerical differentiation, there is an optimal  $h$ . If we take the integration step value below this optimal value, the round-off errors worsen the solution.

**a Second-order problem.** Let's consider now a system with a mass hanging of a spring and with a mechanical gadget (damper) that exerts a force opposite to the mass movement and proportional to the mass speed. For this system the differential equation is as follows:

We have a second-order initial values problem, with an equation that links the variable  $y$  its first derivative  $\frac{dy}{dt}$  and its second derivative  $\frac{d^2y}{dt^2}$ . To solve this kinda problem, we rewrite the differential equation using two first-order

la ecuación como dos ecuaciones de primer orden, una para la posición  $y$  y otra para la velocidad  $v_y = \frac{dy}{dt}$  de manera que obtendríamos un sistema de dos ecuaciones de primer orden acopladas:

$$\begin{aligned}\frac{dy}{dt} &= v_y \\ \frac{dv_y}{dt} &= g - \frac{k}{m} \cdot y - \frac{\mu}{m} \cdot v_y\end{aligned}$$

Ese sistema de ecuaciones de primer orden se puede resolver usando el método de Euler. Necesitamos definir una función de Python para representar el sistema de ecuaciones diferenciales,

```
def amortiguador(t,y):
    """
    Defines the set of differential equations for a damping
    oscilator

    Parameters
    -----
    t : TYPE double
    DESCRIPTION. time
    y : TYPE numpy array
    DESCRIPTION. y[0] -> position
    y[1] -> velocity

    Returns
    -----
    dydt : TYPE numpy arra
    DESCRIPTION. dydt[0] -> velocity
    dydt[1] -> acceleration

    """
    g = 9.8 #gravity accel.
    k = 100. #spring constant
    m = 2. #mass
    mu = 0.5 #friction constant
    dydt = np.zeros(2)
    dydt[0]=y[1]
    dydt[1]=g-(k/m)*y[0]-(mu/m)*y[1];
    return(dydt)
```

Empleando la función descrita `euler('amortiguador',[0 0],0,50,0.01)` obtenemos la evolución temporal de la posición y la velocidad de la masa unida al resorte como

differential equations. one for the position  $y$  and the other for the velocity  $v_y = \frac{dy}{dt}$ . IN this way, we obtain two coupled first-order equations:

This system can be solve using the Euler's method, but we need to define a Python function to represent the system of differential equations,

Using the function just described `euler('amortiguador',[0 0],0,50,0.01)` we compute the temporal evoluntion of position and velocity for the mass attached to the spring.

puede verse en la figura 9.10. En esta figura se aprecia como la masa oscila en torno a la posición de equilibrio hasta que finalmente se para.

Figure 9.10 shows the result, the mass oscillates around an equilibrium position until the damping mechanism, eventually, stops it.

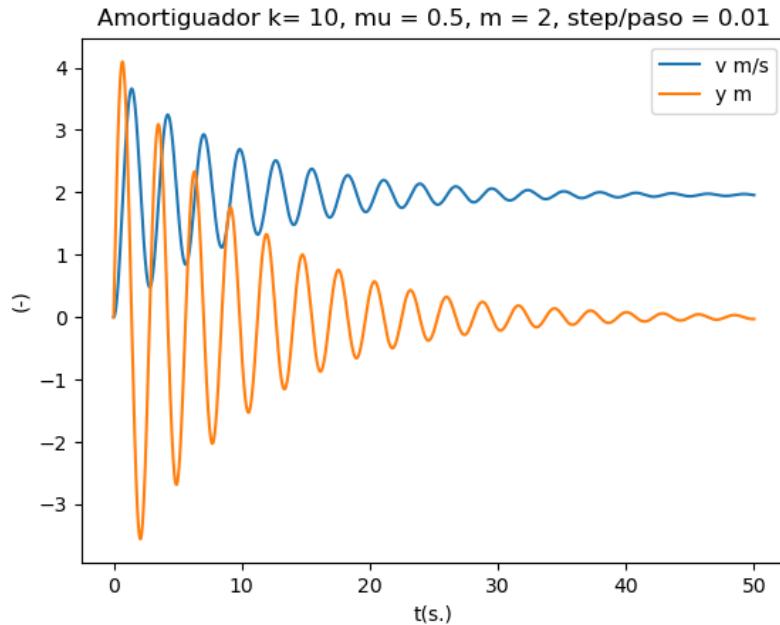


Figura 9.10: Resultado de aplicar el método de Euler a una masa suspendida en vertical de un muelle con rozamiento.

Figure 9.10: Results achieved using Euler's method to solve the damped oscillator problem.

Si cambiamos el valor de la constante del muelle a  $k = 100$  ( $kg/s^2$ ) y volvemos a aplicar el método de Euler para resolver obtenemos el resultado mostrado en la figura 9.11. En la gráfica puede observarse que la masa comienza a oscilar con oscilaciones de amplitud creciente. Para analizar este comportamiento anómalo vamos a repetir el método de Euler reduciendo en un orden de magnitud el paso de integración. Si lo hacemos así obtenemos los resultados de la figura 9.12. Al reducir el paso vemos que la masa oscila en torno a la posición de equilibrio hasta que se estabiliza. El comportamiento observado en la figura 9.11 se debe a errores de integración del algoritmo de Euler.

En la práctica se emplean algoritmos más precisos (y complejos) que el de Euler para re-

If we change the values of the spring constant to  $k = 100$  ( $kg/s^2$ ) and repeat the Euler's method, we arrive to the results showed figure 9.11. We can see in the graphic how the mass oscillates with increasing oscillation amplitude. To analyse this unexpected result we are going to repeat the Euler's method computing, this time using an integration step one order of magnitude less. We obtain for this new calculation the behaviour showed in figure 9.12. Notice how reducing the integrations step reverts the situation and now, the solution of the problem is reasonable, the mass oscillations diminish until, eventually, it stops. The behaviour observed in figure 9.11 is due to Euler's method integration error.

In practice, there are more precise (and complex) algorithms than Euler's to solve initial

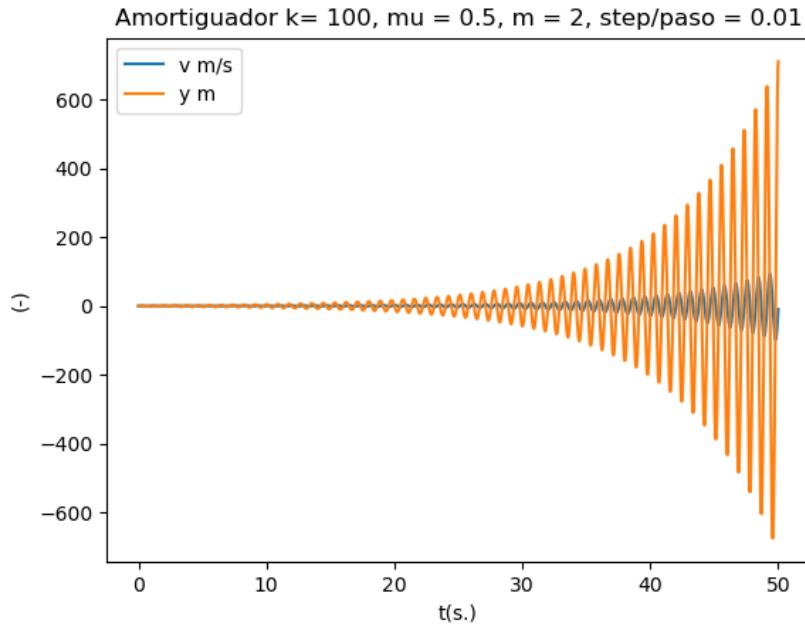


Figura 9.11: Resultado de aplicar el método de Euler a una masa suspendida en vertical de un muelle con rozamiento  $k = 100$ .

Figure 9.11: Result achieved using Euler's method to solve the damped oscillator problem.  $K = 100$

solver problemas de valor inicial, por ejemplo los métodos de Runge-Kutta.



### 9.3.2. Métodos de Runge-Kutta

Si intentamos emplear polinomios de Taylor de grado superior, al empleado en el método de Euler, nos encontramos con la dificultad de obtener las derivadas sucesivas, de la función  $f$ . Así, si quisieramos emplear el polinomio de Taylor de segundo grado, para la función  $x$ , tendríamos,

$$x(t_{i+1}) = x(t_i) + h \cdot f(t_i, x_i) + \frac{h^2}{2} f'(t_i, y_i)$$

Pero,

value problems. For instance, the family of Runge-Kutta methods.

### 9.3.2. The Runge-Kutta methods

If we try to use Taylor's polynomials of higher degree than that used in Euler's method, we have the find the problem of computing the successive derivatives of the function  $f$ . So if we want to use the Taylor's second-degree polynomial, to compute the values of  $x$ , we have,

$$f'(t, x) = \frac{\partial f(t, x)}{\partial t} + \frac{\partial f(t, x)}{\partial x} \cdot f(t, x)$$

but,

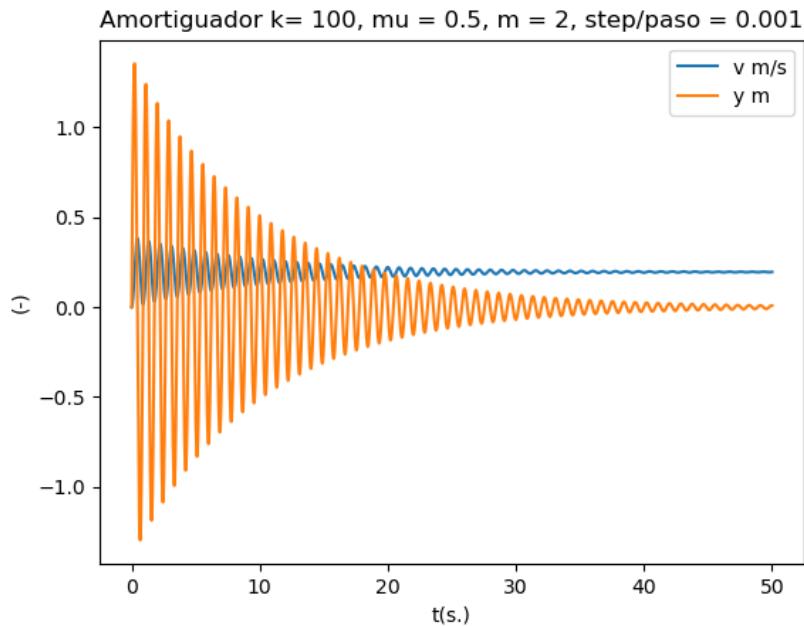


Figura 9.12: Resultado de aplicar el método de Euler a una masa suspendida en vertical de un muelle con rozamiento  $k = 100$  y paso  $h = 0.001$ .

Figure 9.12: Result achieved using Euler's method to solve the damped oscillator problem.  $K = 100$ ,  $h = 0.001$

Sacando factor común  $h$ , obtenemos,

$$x(t_{i+1}) = x(t_i) + h \left( f(t_i, x_i) + \frac{h}{2} \left( \frac{\partial f(t_i, x_i)}{\partial t} + \frac{\partial f(t_i, x_i)}{\partial x} \cdot f(t_i, x_i) \right) \right)$$

A partir de este resultado, es fácil comprender que resulte complicado obtener métodos de resolución basados en el desarrollo de Taylor. De hecho, se vuelven cada vez más complicados según vamos empleando polinomios de Taylor de mayor grado.

Desde un punto de vista práctico, lo que se hace es buscar aproximaciones a los términos sucesivos del desarrollo de Taylor, para evitar tener que calcularlos explícitamente. Estas aproximaciones se basan a su vez, en el desarrollo de Taylor para funciones de dos variables.

Los métodos de integración resultantes, se conocen con el nombre genérico de métodos de Runge-Kutta. Veamos cómo se haría para el caso que acabamos de mostrar del polinomio de Taylor de segundo grado.

Taking out common factor  $h$ , we get,

This result clearly shows that developing methods for solving initial value problems based on Taylor's expansion is challenging. In fact, it becomes increasingly complex as we employ Tailor's polynomials of a higher degree.

From a practical point of view, the common practice is to search for approximations to successive Taylor's expansion terms rather than calculate them explicitly. These approximation are, in turn, based on Taylor's expansion for two variable functions.

The resulting methods are known in general as the Runge-Kutta methods. Let us see how to do it for the case we have just shown of a second-degree Taylor's polynomial.

First, we will find the first-degree Taylor expansion polynomial for the function  $f(t, x)$

En primer lugar obtenemos el desarrollo del polinomio de Taylor de primer grado, en dos variables de la función  $f(t, x)$ , en un entorno del punto  $t_i, x_i$ ,

$$f(t, x) = f(t_i, x_i) + \left( (t - t_i) \frac{\partial f(t_i, x_i)}{\partial t} + (x - x_i) \frac{\partial f(t_i, x_i)}{\partial x} \right)$$

Si ahora comparamos este resultado con la ecuación anterior, podríamos tratar de identificar entre sí los términos que acompañan las derivadas parciales,

around the point  $t_i, x_i$  with respect to two variables,

If we compare this result with the previous equation, we can establish a relationship between the coefficients accompanying the partial derivatives.

$$\begin{aligned} t - t_i &= \frac{h}{2} \rightarrow t = t_i + \frac{h}{2} \\ x - x_i &= \frac{h}{2} \cdot f(t_i, x_i) \rightarrow x = x_i + \frac{h}{2} \cdot f(t_i, x_i) \end{aligned}$$

Es decir,

| That is,

$$f(t_i, x_i) + \frac{h}{2} \left( \frac{\partial f(t_i, x_i)}{\partial t} + \frac{\partial f(t_i, x_i)}{\partial x} \cdot f(t_i, x_i) \right) = f(t_i + \frac{h}{2}, x_i + \frac{h}{2} \cdot f(t_i, x_i))$$

Si ahora sustituimos este resultado en nuestra expresión del polinomio de Taylor de segundo grado de la función  $x(t)$ ,

And after substituting this result in our expression for function  $x(t)$  second-degree Taylor's polynomial expansion,

$$x(t_{i+1}) = x(t_i) + h \cdot f(t_i + \frac{h}{2}, x_i + \frac{h}{2} \cdot f(t_i, x_i))$$

Donde  $x_i \equiv x(t_i)$ . Esta aproximación da lugar al primero y más sencillo de los métodos de Runge-Kutta, conocido como método del punto medio. El nombre es debido a que la función  $f$  se evalúa en un punto a mitad de camino entre  $t_i$  y  $t_{i+1} = t_i + h$ . El cálculo de la solución de un problema de valor inicial mediante este método, se puede expresar de un modo análogo al del método de Euler,

Where  $x_i \equiv x(t_i)$ . This approximation paves the way for the first and simplest of Runge-Kutta methods, the midpoint method. This name comes from evaluating the function  $f$  at a point midway between  $t_i$  and  $t_{i+1} = t_i + h$ . We can write the midpoint method algorithm to compute an initial value problem solution in a similar way to Euler's method.

$$\begin{aligned} x_0 &= x(a) \\ x_1 &= x_0 + h \cdot f(a + \frac{h}{2}, x_0 + \frac{h}{2} f(a, x_0)) \\ &\vdots \\ x_{i+1} &= x_i + h \cdot f(t_i + \frac{h}{2}, x_i + \frac{h}{2} f(t_i, x_i)) \end{aligned}$$

La siguiente función en Python implementa el método del punto medio,

The following Python code implement the midpoint method,

---

midpoint\_method.py

```

1  # -*- coding: utf-8 -*-
2  """
3  Created on Fri Aug 23 16:10:09 2024
4  The Runge-Kutta midpoint method
5  @author: abierto
6  """
7  import numpy as np
8  import matplotlib.pyplot as pl
9  def pmedio(fun,xa,a,b,h):
10     """
11         This funtion implements the second-order Runge-Kutta method
12         better known as the midpoint method
13
14     Parameters
15     -----
16     fin : TYPE function
17         DESCRIPTION. a funtion describing the initial value problem
18         derivative to be integrate
19     xa : TYPE numpy array
20         DESCRIPTION. initial condition
21     a : TYPE double
22         DESCRIPTION. initial integratio point
23     b : TYPE double
24         DESCRIPTION. final integration point
25     h : TYPE double
26         DESCRIPTION. integration step
27
28     Returns
29     -----
30     x: TYPE numpy array
31         DESCRIPTION. solutions at points a, a+h, a+2h etc
32     t. TYPE numpy array
33         DESCRIPTION. times x has been computed at
34     """
35     if a >= b:
36         raise ValueError('I need an increasing interval')
37
38     #to arrive to the final condition, we modify slightly the
39     #integration step. First we calculate the aproximate number
40     #of intervals between a and b of size h and round it towards
41     #infinite.
42     pt = int(np.ceil((b-a)/h))
43     #then, we modify the integration step to acomodate at
44     #the rounded number of interval. This is, of course, not
45     #necessary to implement the method, there are other
46     #possibilities to deal with the integration step and
47     #the limit b
48     hft = (b-a)/pt
49     t = np.arange(a,b+hft,hft)
50     x = np.zeros([xa.shape[0],pt+1])
51     x[:,0] = xa

```

```

52     for i in range(1,pt+1):
53         x[:,i] = x[:,i-1] + hft*fun(t[i-1]+hft/2,x[:,i-1]\
54                                     +hft*fun(t[i-1],x[:,i-1])/2)
55     pl.plot(t,x.T)
56     return(t,x)

```

El resto de los métodos de Runge-Kutta, los dejaremos para cuando seáis más mayores... Pero que conste que es de lo más interesante de los métodos numéricos.

We leave the remaining Runge-Kutta method until you are grown up. But notice that it is one of the most interesting topics in numerical methods.



## 9.4. Ejercicios

1. Crea una función en Python que tome como variables de entrada otra función  $f$ , un valor  $x_0$ , un intervalo  $h$  y una última variable `metodo` que contenga el nombre de un método y devuelva el valor de la derivada de  $f$  calculada en el punto  $x_0$ , empleando el método indicado en la variable `metodo`. Los métodos pueden ser:

- `metodo = '2ad'`: diferencia de dos puntos adelantada
- `metodo = '2ce'`: diferencia de dos puntos centrada
- `metodo = '3ad'`: diferencia de tres puntos adelantada

Emplea la función que acabas de crear para obtener la derivada de la función  $f(x) = 1/x$  en el punto  $x_0 = 1$ . Prueba para valores de  $h$  0.1, 0.5, 1.5. Explica los resultados.

2. Analiza los programas mostrados en la sección 9.2 para el cálculo de integrales empleando los métodos de Trapecio, Simpson y Simpson 3/8 y la función para calcular los métodos extendidos. Comprueba la precisión de los métodos calculando la integral,

$$\int_0^\pi \sin(x) = 2$$

## 9.4. Exercices

1. Build a Python function that takes another function  $f$ , a value  $x_0$ , an interval  $h$  and a variable `method` as input variables, and return the derivative of  $f$  calculated at point  $x_0$ . The function should use the method indicated in the input variable `method` to calculate the derivative. Possible method to chose should be:

- `metodo = '2ad'`: two points forward difference
- `metodo = '2ce'`: two points central difference
- `metodo = '3ad'`: three points forward difference

Employ the function just built to compute the function  $f(x) = 1/x$  derivative at point  $x_0 = 1$ . Try for  $h$  values, 0.1, 0.5, 1.5. Explain the result you get.

2. Analyse the programs for computing integrals using the trapezium, Simpson's and Simpson's 3/8 described in section 9.2, and also the function for the extended methods. Check the precision of the different methods by computing the following integral,

**3.** El programa del apartado anterior, solo es útil cuando se conoce la función que se desea integrar. Sin embargo es posible aplicar los métodos de integración numérica descritos cuando solo se dispone de una tabla de datos.

a) Escribe una función admite como entrada un vector de datos equiespaciados [ $y = y_0, y_1, \dots, y_n$ ], un intervalo de integración  $h = x_i - x_{i-1}$ , y una variable con el nombre de un método (de modo análogo a como se ha hecho en el ejercicio anterior). La función deberá devolver la integral de la función que representan los datos de la tabla, aplicando el correspondiente método extendido, sobre los datos de  $y$ :

**3.** The previous question program is only useful when you know the function you intend to integrate. However, it is possible to applying the method also when you only have a data table.

a) Write a function that takes as inputs a vector of equispaced data [ $y = y_0, y_1, \dots, y_n$ ], an integration interval  $h = x_i - x_{i-1}$ , and a variable with a method name (as in the previous exercise). The function should return the integral of the function represented by the data table, applying the indicated (extended) method onto the  $y$  data:

Trapecio/Trapezium:

$$\begin{aligned} I(y) &= \int_{x_0}^{x_n} y dx \approx \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} y(x) dx = \sum_{i=0}^{n-1} \frac{h}{2} (y_i + y_{i+1}) \\ &= \frac{h}{2} (y_0 + 2y_1 + 2y_2 + \dots + 2y_{n-1} + y_n) \end{aligned}$$

Simpson's:

$$\begin{aligned} I(y) &\approx \sum_{i=0}^{\frac{n}{2}-1} \int_{x_{2i}}^{x_{2i+2}} P_2(x) dx = \sum_{i=0}^{\frac{n}{2}-1} \frac{h}{3} (y_{2i} + 4y_{2i+1} + y_{2i+2}) \\ &= \frac{h}{3} (y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \dots + 2y_{n-2} + 4y_{n-1} + y_n) \end{aligned}$$

Simpson's 3/8

$$\begin{aligned} I(y) &\approx \sum_{i=0}^{\frac{n}{3}-1} \int_{x_{3i}}^{x_{3i+3}} P_3(x) dx = \sum_{i=0}^{\frac{n}{3}-1} \frac{3h}{8} (y_{3i} + 3y_{3i+1} + 3y_{3i+2} + y_{3i+3}) \\ &= \frac{3h}{8} (y_0 + 3y_1 + 3y_2 + 2y_3 + 3y_4 + 3y_5 + 2y_6 + \dots + 2y_{n-3} + 3y_{n-2} + 3y_{n-1} + y_n) \end{aligned}$$

b) Es importante tener en cuenta que, para el caso de Simpson,  $n$  –El número de datos del vector  $y$  menos 1– debe ser divisible entre dos. Añade el código necesario al programa realizado para que integre el último intervalo empleando el método del trapecio, caso de no cumplirse la condición anterior. Del mismo modo, para Simpson 3/8,  $n$  debe ser divisible entre tres. Añada el código necesario para que, si no se cumple esta condición, integre por el método del trapecio, si sobra un intervalo o por el método de Simpson si sobran dos.

c) Aplica la función de los apartados ante-

b) It is important to note that in the case of Simpson's method,  $n$  –the number of elements in the array  $y$  minus one– should be divisible by two. Add the required code to the existing program to integrate the last interval using the trapezium method if the previous condition is not met. Similarly, for Simpson's 3/8 the number of elements minus one should be divisible by 3. Add the necessary code so that, if this condition is not met, the function uses the trapezoid method if there is one interval leftover or the Simpson method if there are two left.

riores a la siguiente tabla de datos,

x	0.	0.314	0.628	0.942	1.256	1.570	1.884	2.199	2.513	2.827	3.141
y	0.	0.309	0.587	0.809	0.951	1.000	0.951	0.809	0.587	0.309	0.000

calcula el resultado empleando los tres métodos, y compáralos con el que dan las funciones de Scipy, `scipy.integrate.trapezoid` y `scipy.integrate.simpson`.

4. La ecuación diferencial de un péndulo físico toma la forma,

$$\frac{d^2\theta}{dt^2} = -\frac{g}{l} \sin(\theta),$$

donde  $\theta$  es el ángulo que el péndulo forma con la vertical,  $g \approx 9.8m/s^2$  es la aceleración de la gravedad y  $l$  es la longitud del péndulo.

a) Reescribe la ecuación del péndulo en dos ecuaciones de primer orden, una para el ángulo en función de la velocidad angular  $\omega = \frac{d\theta}{dt}$  y otra para la velocidad angular  $\frac{d\omega}{dt} = \frac{d^2\theta}{dt^2}$

b) Crea un programa que permita obtener los valores de  $\omega$  y  $\theta$  en función de tiempo, a partir de las ecuaciones obtenidas en el apartado anterior, empleando el método de Euler. El programa deberá permitir dar valores iniciales  $\theta_0$  y  $\omega_0$  a la posición y velocidad angulares, definir un intervalo de tiempo de integración  $[t_0, t_f]$  y un paso de integración  $\Delta t$

c) Para desplazamientos pequeños de  $\theta$ , tomando  $\sin(\theta) \approx \theta$ , es posible aproximar la ecuación exacta del péndulo por la siguiente ecuación,

$$\frac{d^2\theta}{dt^2} = -\frac{g}{l}\theta,$$

Añade al programa realizado en el apartado 4b) el código necesario para que integre simultáneamente las ecuaciones del péndulo exacta y la aproximada.

d) Compara gráficamente los resultados para la posición y velocidad obtenidas a partir de las ecuaciones exacta y aproximada para un péndulo de longitud  $l = 0.1m$ , si parte del reposo, tomando ángulos iniciales  $\theta_0: \pi/10, pi/6, pi/4, pi/3$ . Emplea para la integración un intervalo de tiempo de  $t_f - t_0 = 1s$  y un paso de integración  $\Delta t = 1E - 3s$

c) Apply the function built in the previous sections to the following data table,

compute the results using the three methods and compare them with the results yielded by Scipy functions, `scipy.integrate.trapezoid` y `scipy.integrate.simpson`.

4. The simple pendulum differential equation takes the following form,

where  $\theta$  is the angle of the pendulum with respect to the vertical,  $g \approx 9.8m/s^2$  is the gravity acceleration and  $l$  is the pendulum length.

a) Rewrite the pendulum equation as two first order differential equations, one for the angle, depending on the angular velocity  $\omega = \frac{d\theta}{dt}$  and another for the angular velocity  $\frac{d\omega}{dt} = \frac{d^2\theta}{dt^2}$ .

b) Write a program that obtains  $\omega$  and  $\theta$  values as time functions using the equations obtained in the previous section and Euler's method. The program should get as input variables: Initial values  $\theta_0$  y  $\omega_0$  for angular position and velocity, an integration time interval  $[t_0, t_f]$  and an integration step  $\Delta t$

c) For small  $\theta$  displacements, we can approximate  $\sin(\theta) \approx \theta$ , then the pendulum equation can be approximate as,

add to the program made in section 4b) the code necessary to integrate simultaneously the exact and the approximate pendulum equations.

d) Compare graphically the position and velocity obtained using the exact and the approximate pendulum equations for a pendulum of length  $l = 0.1m$ . Consider the pendulum initial velocity equal to zero and initial angular positions  $\theta_0: \pi/10, pi/6, pi/4, pi/3$ . Employ an integration time interval  $t_f - t_0 = 1s$  and an integration step  $\delta t = 1E - 3s$

5. La dinámica de un planeta que gira en torno a una estrella viene dada por las siguientes ecuaciones del movimiento,

$$\begin{aligned}\frac{d^2x}{dt^2} &= -G \frac{M}{(x^2 + y^2)^{3/2}} x \\ \frac{d^2y}{dt^2} &= -G \frac{M}{(x^2 + y^2)^{3/2}} y\end{aligned}$$

Donde  $G$  es la constante universal de la gravedad,  $M$  la masa de la estrella, que se encuentra situada en el origen del sistema de coordenadas, y  $(x, y)$  las coordenadas del vector posición del planeta.

a). transforma las ecuaciones del movimiento en un sistema de cuatro ecuaciones de primer orden, empleando la velocidad del planeta,  $v_x = \frac{dx}{dt}$ ,  $v_y = \frac{dy}{dt}$ .

b). Escribe un programa que permita obtener la trayectoria (órbita) de un planeta resolviendo mediante el método de Euler las ecuaciones obtenidas en el apartado anterior.

c). Obtén la trayectoria (órbita) de un planeta que gira en torno a una estrella, empleando los siguientes datos, medidos en unidades arbitrarias: Masa de la estrella:  $M = 1$ . Constante de la gravedad:  $G = 1$ . Posición inicial del planeta:  $x_0 = 1$ ,  $y_0 = 0$ . Velocidad inicial  $v_x = 0$ ,  $v_y = 0.7$ . Emplea un paso de integración  $\Delta t = 0.01$ . Calcula la solución durante un tiempo total  $t = 10$ .

**Nota:** El método de Euler no es suficientemente preciso para resolver este tipo de problemas. Es fácil obtener resultados muy alejados de la realidad en función del valor que tomen las condiciones iniciales.

## 9.5. Test del curso 2020/21

Sobre un plano inclinado se coloca un bloque sujeto por un muelle, tal y como muestra en la Figura 9.13. La superficie de contacto entre el bloque y el plano está muy pulida; por lo que se puede considerar que el rozamiento entre ambos es proporcional a la velocidad  $v_x(t(t)) \in \mathbb{R}$  a lo largo del eje paralelo al plano.

El movimiento del bloque en la dirección del plano puede determinarse mediante la ecua-

5. The dynamic of planet that rotates around a star is defined by the following dynamical equations,

Where  $G$  is the universal gravity constant,  $M$  is the star mass, which is located in the origin of the coordinate system, and  $(x, y)$  are the coordinates of the planet position vector.

a). Transform the dynamical equations in a set of four first-order equations, using the planet velocity,  $v_x = \frac{dx}{dt}$ ,  $v_y = \frac{dy}{dt}$ .

b). Write a program to compute the trajectory (orbit) of the planet, solving by Euler's method the equations obtained in the previous section.

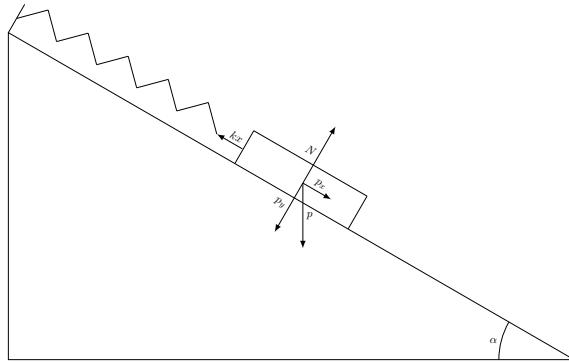
c). Compute the trajectory (orbit) of a plane that rotates around a star, using the following data, defined in arbitrary units: Star mass  $M = 1$ . Gravity constant:  $G = 1$ . Planet initial position :  $x_0 = 1$ ,  $y_0 = 0$ . Planet initial velocity  $v_x = 0$ ,  $v_y = 0.7$ . Step size  $\Delta t = 0.01$ . Compute the solution for a total time  $t = 10$ .

**Note:** The Euler's method is not accurate enough to solve this kind of problem. For this reason, it is likely to obtain results far away from the actual solution, depending on the initial condition values.

## 9.5. Course 2020/21 test.

A block attached to a spring is placed on an inclined plane, as shown in figure 9.13. The contact surface between the plane and the block is very smooth, and the friction between both then can be considered proportional to the speed  $v_x(t(t)) \in \mathbb{R}$  along the plane surface.

The displacement of the block along the plane can be described using the following equation,



$g = 10 \text{ m/s}^2$  gravedad/gravity

$k = 60 \text{ N/m}$  constante del muelle/ spring constant

$m = 2 \text{ kg}$  masa del bloque/block mass

$\alpha = \frac{\pi}{6}$  ángulo del plano inclinado/inclined plane angle

$N$  = Reacción normal del plano/ plane normal force

$p = mg$

$\mu = 0.1 \text{ kg/s}$  coeficiente de rozamiento/friction coefficient

Figura 9.13: Sistema masa-muelle-plano inclinado.

Figure 9.13: Mass-spring-inclined plane system.

ción

$$ma_x(t) = -kx(t) - \mu|N|v_x(t) + p_x, \quad (9.1)$$

donde  $a(t) \in \mathbb{R}$  representa la aceleración del bloque,  $v_x(t) \in \mathbb{R}$  la velocidad,  $x(t) \in \mathbb{R}$  su posición y el resto constantes se describen en la Figura 9.13. El origen,  $x = 0$ , se toma en el punto en donde la fuerza recuperadora del muelle es nula. La posición crece en el sentido de bajada a lo largo del plano.

**1.** Emplea el método de Euler para estimar las posición  $x(t)$  y la velocidad  $v_x(t)$  del bloque a partir de la ecuación (9.1). Utiliza un paso de integración  $h = 10^{-3}\text{s}$ . Considera una posición inicial  $x(0) = 0\text{m}$ , una velocidad inicial  $v(0) = 0\text{m/s}$  y un tiempo final de integración  $t_f = 11\text{s}$ .

**2.** Representa gráficamente los resultados obtenidos: Posición frente a tiempo y velocidad frente a tiempo. Emplea una figura distinta para cada representación. No olvides añadir rótulos a los ejes indicando las variables representadas con sus unidades.

**3.** La solución analítica del problema, para velocidad inicial  $v(0) = 0$  y cualquier posición inicial  $x(0)$  toma la siguiente forma,

Where  $a(t) \in \mathbb{R}$  represents the block acceleration,  $v_x(t) \in \mathbb{R}$  the block velocity,  $x(t) \in \mathbb{R}$  its position and the remaining constant are described in figure 9.13. The coordinate origin  $x = 0$  is taken where the spring recovery strength is null. The block position increases in the downward direction along the plane.

**1.** Employ the Euler's method to stimate the block position  $x(t)$  and velocity  $v_x(t)$  from equation (9.1). Use a step size  $h = 10^{-3}\text{s}$ . Take  $x(0) = 0\text{m}$  as the initial position,  $v(0) = 0\text{m/s}$  as the initial velocity and a final integration time  $t_f = 11\text{s}$ .

**2.** Represent graphically the results achieved: Position versus time and velocity versus time, using a different figure for each representation. Do not forget to add axis labels indicating the variables and their units.

**3.** The problem analytical solution, for  $v(0) = 0$  initial speed and an arbitrary initial position  $x(0)$  is,

$$x(t) = \frac{p_x}{k} - \left( \frac{p_x}{k} - x(0) \right) \frac{\omega_0}{\omega} e^{-\eta t} \cos(\omega t - \phi) \quad (9.2)$$

donde,

$$\omega_0 = \sqrt{\frac{k}{m}} \quad \eta = \frac{\mu N}{2m}$$

$$\omega = \sqrt{\omega_0^2 - \eta^2} \quad \phi = \arcsin\left(\frac{\eta}{\omega_0}\right)$$

a) Calcula la posición del bloque mediante la ecuación (9.2). Emplea para ello el mismo intervalo y los mismos instantes de tiempo empleados en el apartado 9. Representa los resultados sobre el gráfico para la posición calculada mediante el método de Euler que has obtenido en el apartado 9.

b) Representa en un gráfico de barras los residuos resultantes de comparar la solución analítica con la obtenida por el método de Euler.

c) Calcula el error cuadrático medio cometido al emplear el método de Euler. Considera como exacta la solución analítica (9.2).

**4.** Los instantes de tiempo para los que la posición del bloque alcanza un máximo o un mínimo local pueden obtenerse a partir de la frecuencia de oscilación  $\omega$ ,

$$t_{max/min} = \frac{n\pi}{\omega}, n = 1, 2, 3, \dots, \infty \quad (9.3)$$

a) Emplea las ecuaciones (9.3) y (9.2) para obtener los primeros 20 puntos singulares (máximos o mínimos) del movimiento del bloque. Represéntalos sobre el gráfico de la posición, obtenido en el apartado 3.b).

b) Utilizando los datos obtenidos en el apartado 4.a), emplea el método de diferencia de dos puntos centrada para obtener las derivadas de las posiciones en los máximos y mínimos locales con respecto al tiempo. Si hay puntos para los que no es posible aplicar este método, calcula su derivada empleando otra aproximación razonable.

c) Representa los resultados obtenidos en el apartado anterior sobre el gráfico de la velocidad obtenido en el ejercicio 9. A la vista de los resultados, ¿Cómo valorarías la precisión del método empleado para obtener las derivadas?

where,

$$\eta = \frac{\mu N}{2m}$$

$$\phi = \arcsin\left(\frac{\eta}{\omega_0}\right)$$

a) Compute the block position using equation (9.2). Employ the same interval and the same time instants, used in section refp1, to do it. Draw the result on top the position graphic computed in section 9, using the Euler's method.

b) Compare the analytical solution with Euler's method solution and represent, using a bar graphic, the residual obtained from this comparison.

c) Compute the quadratic error made using the Euler's method. Take the analytical solution as the exact solution.

**4.** We can obtain the time instant at which the block position reaches a local maximum or minimum using the the oscillation frequency  $\omega$ ,

a) Use equations (9.3) and (9.2) to obtain the first twenty singular points (maxima or minima) for the block displacement. Draw them on top the position graph obtained in section 3.b).

b) Using the data obtained in section 4.a), employ the two point central difference method to obtain the time derivatives of the block position at the local maxima and minima. If there are points for which it is not possible to apply this method, compute the derivative using another reasonable approach.

c) Represent the result obtained on top of the graphic for the velocity drawn in section 9. According to the resulting graphic, how do you consider the precision of the method used for computing the derivatives?

**5.** From the results obtained for the velocity in section 9, compute the the position of the block at time  $t = 1s$  using the following inte-

**5.** A partir de los resultados obtenidos para la velocidad en el apartado 9, calcula la posición del bloque en el instante de tiempo  $t = 1s$  mediante la siguiente integral

$$x(1) = \int_0^1 v_x(t) dt, \quad (9.4)$$

a) Emplea para ello el método del trapecio.  
 b) Compara el resultado con los valores obtenidos tanto mediante el método de Euler como a partir de la solución analítica.

gral,

a) Employ to do it the trapezium formula  
 b) Compare the results with those obtained from the Euler's method and from the analytical solution.

# Índice alfabético

- = Símbolo de asignación, 45
- Acumulación de errores, 234
- Adición en binario, 214
- Ajuste polinómico, 374
- Alineamiento, 234
- ALU, 28
- anaconda, 38
- Anidación
  - If anidado, 91
  - While anidado, 102
  - For anidado, 98
- Anulación catastrófica., 239
- Análisis de convergencia, 335
- Attracting fixed point, 265
- Base 10, 30
- Base 2, 29
- Binario, 209
- bit, 29
- Bit de signo, 212
- Bucles, 93
  - Bucle for, 93
  - Bucle while, 100
- Byte, 29
- Bézier, 385
- Circuito RC, 422
- Compilador, 25
- Complemento a dos, 212
- Computador
  - hardware*, 24
  - arquitectura, 24
- Control de flujo, 85
- Convergence analysis, 335
- Conversion
  - binary to decimal, 31
- Conversión
- binario a decimal, 31
- decimal a binario, números no entero, 32
- decimal a binario. números enteros, 31
- CPU, 28
- Curvas de Bézier, 385
- Curvas de Bézier equivalentes, 392
- Cálculo numérico, 22
- Data, analysis, 21
- Datos, análisis, 21
- Depurador, 80
- Desbordamiento, 232
- Diferenciación
  - Diferencia de dos puntos centrada, 406
  - diferencias finitas, 404
  - polinomio interpolador, 403
  - Diferencia adelantada de dos puntos, 405
- Diferencias Divididas, 355
- Differentiation
  - Two point central difference, 406
- Divided differences, 355
- Ecuación diferencial, 418
- Editor de textos, 65
- epsilon del computador, 230
- Error
  - de sintáxis, 81
- Error de redondeo, 228, 405
- Error de Truncamiento, 405
- Errores
  - de codificación, 82
- Errores aritméticos, 233
- Estándar IEEE 754, 219
  - Doble precisión, 224
  - Simple precisión, 219
- Exceso, 228

- Exponente
  - Representación en exceso, 220
- Factorizacion SVD, 181
- Fijex point
  - Iteration, 268
- Fixed point, 264
  - theorem, 266
- Flow
  - Loops, 93
- Flujo, 85
  - Bucle for, 93
  - Bucle while, 100
  - Diagrama de Flujo, 105
  - Bucles, 93
  - Condicional, 85
- Funciones
  - Recursivas, 103
- funciones, 69
- Funciones
  - Funciones incluidas en Numpy, 155
- Functions
  - Functions included in Numpy, 155
- Gauss
  - Método de eliminación gaussiana, 310
- Gauss-Jordan
  - eliminación, 318
  - elimination, 318
- Gauss-Seidel's method
  - Matrix form, 330
- Gauss-Seidel's method., 328
- Gaussian elimination method, 310
- hodógrafa, 393
- IEEE 754
  - Número desnormalizado más próximo a cero en doble precisión, 224
  - Número desnormalizado más próximo a cero en simple precisión, 223
  - Número más grande representable en doble precisión, 224
  - número más grande representable en simple precisión, 222
  - Números desnormalizados, 222
- import, 79
- indexación, 49
- Integración
  - Formulas de Newton-Cotes, 409
  - Fórmula compuesta del trapecio, 411
  - Fórmula del trapecio, 410
  - Fórmulas de Simpson, 412
- interp1, 371
- Interpolación
  - Diferencias Divididas, 355
  - Polinomio de Lagrange, 353
  - Polinómica, 350
  - Teorema de unicidad, 350
  - Polinomio de Newton-Gregory, 359
  - Splines, 364
- Interpolación de orden cero, 363
- Interpolación lineal, 364
- Interpolation
  - Divided differences, 355
  - Polynomial, 350
- Ipython, 43
- Jacobi's method
  - Matrix expression, 324
- Jacobi's method., 322
- List comprehesion, 99
- Loops, 93
- Mantisa, 217
  - normalizada, 220
- Matrices
  - Operaciones Matriciales, 134
  - en Numpy, 126
- Matriz de Vandermonde, 352
- Método de Euler, 420
- Método de Gauss-Seidel, 328
  - Forma matricial, 330
- Método de Jacobi, 322
  - Expresión matricial, 324
- Método de Jacobi amortiguado, 333
- Método de Runge-Kutta, 428
- Método SOR, 334
- Métodos amortiguados, 332
- Mínimos cuadrados, 374
  - Residuos, 383
- NameSpace, 74
- NaN, 221
- Nesting
  - Nested for, 98
- Not a Number, 221
- Notación científica, 210, 217
- Numpy, 123

- indexación, 128
- Número máquina, 227
- Operaciones, 54
  - Aritméticas, 54
- Operadores
  - Relaciones y lógicos, 58
  - Precedencia, 56
- Operations
  - Arithmétics, 54
- Polinomio de Lagrange, 353
- Polinomio de Newton-Gregory, 359
- Polinomio de Taylor, 346
  - Error de la aproximación, 347
  - Serie de la función exponencial, 347
  - Serie del logaritmo natural, 347
  - Series de las funciones seno y coseno, 349
- Polinomios, 274
  - Raíces de un polinomio, 275
- Polinomios de Bernstein, 385
- Problemas de valor inicial, 418
- Programación
  - aplicaciones, 26
  - lenguajes, 26
- Punto fijo
  - atractivo, 265
  - de una función, 264
  - Método, 268
  - Teorema, 266
- Python, 37
  - entorno de programación, 38
- Representación numérica, 211, 212
- en punto fijo, 215
- en punto flotante, 217
- Residuos, 383
- Scripts, 64
- Sistema operativo, 25
- sistemas, 291
- SOR method, 334
- Speed of convergence, 338
- Splines, 364
  - Cubicos, 366
- Spyder, 42
- SVD Factorisation, 181
- Símbolo de asignación, 45
- Taylor polynomial for the logarithm function, 347
- Truncamiento, 210
- truncamiento, 228
- Variable
  - nombre, 45
  - tipo, 47
  - Ambito, 74
- Variables, 45
- Vectores
  - en Numpy, 126
- Velocidad de convergencia, 338
- Von Neumann, 27
- Weighted Jacobi method, 333
- Weighted methods, 332
- Ámbito de una variable o función, 79



# Alphabetic Index

- = assignment symbol, 45
- [, 74
- 2's complement, 212
  - subtraction, 213
- Anaconda, 38
- Arithmetic errors, 233
- Asignment Symbol, 45
- Base 2, 29
- Binary, 209
  - binary addition, 214
- Bézier, 385
  - Bézier's curves, 385
- Catastrophic cancellation., 239
- Compiler, 25
- Complemento a dos
  - Sustracción, 213
- Computer
  - hardware, 24
  - architecture, 24
- Conversion
  - decimal to binary. Integer numbers, 31
  - denary to binary, non-integer numbers, 32
- Debugger, 80
- Diferentiation
  - Two point forward difference, 405
- Differential equation, 418
- Differentiation
  - finite differences, 404
- differentiation
  - interpolation polynomial, 403
- Equivalent Bezier's curves, 392
- Error
  - syntax error., 81
- errors accumulation, 234
- Exponent
  - Biased, 220
- Flow
  - while loop, 100
- flow, 85
  - for loop, 93
  - Conditional, 85
- flow control, 85
- functions, 69
- IEEE 237 standard
  - Doble precision, 224
- IEEE 754
  - Closest Number to zero in simple precision, 223
- IEEE 754 Standard, 219
  - Simple precision, 219
- import, 79
- indexation, 49
- Initial value problems., 418
- Integration
  - Newton-Cotes formulae, 409
  - Trapezium rule, 410
- Integration
  - Composed trapezium formula, 411
- Interpolation
  - Interpolation theorem, 350
  - Lagrange Polynomial, 353
  - Splines, 364
  - The Newton-Gregory polynomial, 359
- Intrpolation
  - Vandermonde's matrix, 352
- Lagrange Polynomial, 353
- Least Squared Error, 374
- Linear interpolation, 364

- List comprehension, 99
- Loops
  - while loop., 100
- machine epsilon, 230
- Mantissa
  - normalised, 219
- Matrices
  - in Numpy, 126
  - Matrix Operations, 134
- Nesting
  - Nested if, 91
  - Nested While, 102
- Number representation
  - floating-point, 217
- Numeric Representation, 212
- Numpy, 123
  - Indexing, 128
- operating system, 25
- Operations, 54
  - Relational and Logic , 58
- Operators
  - Precedence, 56
- Overflow, 232
- Polynomial fitting, 374
- Programming
  - languages, 26
  - Programmes, 26
- Python, 37
  - IDE, 38
- RC circuit, 422
- residuals, 383
- Roots, 245
- RUnge-Kutta methods, 428
- Scientific computing, 22
- Scientific Notation, 217
- Scripts, 64
- Splines, 364
- Spyder, 42
- systems, 291
- Taylor's polynomial
  - Sine and cosine function expansion., 349
- Taylor Polynomial
  - Series expansion for the exponential function, 347
- Taylor's polynomial, 346
- Text editor, 65
- The Newton-Gregory polynomial, 359
- Truncation, 210
- Underflow, 232
- Varaible and function scope, 79
- Variable
  - name, 45
  - Scope, 74
  - Type, 47
- Variables, 45
- Vectors
  - in Numpy, 126
- Von Neumann, 27
- zero-order interpolation, 363