Cálculo Científico para Ingeniería

Fundamentos, Métodos y Aplicaciones Computacionales

WILLIAM LA CRUZ, Dr.

Profesor Titular
Departamento de Electrónica, Computación y Control
Escuela de Ingeniería Eléctrica
Facultad de Ingeniería
Universidad Central de Venezuela

Un patrocinio de la Facultad de Ingeniería
de la Universidad Central de Venezuela

Universidad Central de Venezuela

Universidad Central de Venezuela, 2021

Cálculo Científico para Ingeniería: Fundamentos, Métodos y Aplicaciones Computacionales
por William La Cruz, se distribuye bajo una licencia libre.

Publicado por William La Cruz, e-mail: lacruzw@gmail.com, Caracas, Venezuela
Disponible en https://saber.ucv.ve/jspui/

xxix p. 373 p. cm.—(Serie: Ingeniería Eléctrica)
145 figuras.
"Universidad Central de Venezuela-Facultad de Ingeniería."
1. Cálculo Científico. 2. Ingeniería Eléctrica.

Contenido

Contenido ii

Antes de leer y usar los programas de este libro v

1 Herramientas Matemáticas Básicas 1
1.1 Conceptos Básicos del Álgebra Lineal . . . 1
1.1.1 Espacio Vectorial, Producto Escalar y Normas Vectoriales . . . 1
1.1.2 Matrices . . . 5
1.1.3 Normas Matriciales . . . 11
1.1.4 Alcance y Rango de una Matriz . . . 12
1.1.5 Matrices Especiales . . . 13
1.1.6 Autovalores y Autovectores . . . 15
1.2 Conceptos Básicos del Cálculo . . . 15
1.2.1 Funciones de una Variable Real . . . 16
1.2.2 Funciones de Varias Variables . . . 21
1.3 Calculando con Números . . . 23
1.3.1 Representación Numérica . . . 24
1.3.2 Representación de Punto-Flotante . . . 24
1.3.3 Eficiencia, Estabilidad y Condicionamiento . . . 30
1.3.4 Velocidad de Convergencia . . . 37
1.4 Ejercicios . . . 38

2 Sistemas de Ecuaciones Lineales 43
2.1 Conceptos Básicos . . . 44
2.1.1 Solución de un Sistema de Ecuaciones Lineales por Determinantes . . . 45
2.1.2 Sistemas Lineales Especiales . . . 46
2.2 Eliminación Gaussiana . . . 49
2.2.1 Eliminación Gaussiana Simple . . . 50
2.2.2 Eliminación Gaussiana con Pivoteo . . . 54
2.3 Factorización LU . . . 58
2.3.1 Factorización $A = LU$ . . . 58
2.3.2 Factorización $PA = LU$ . . . 63
2.3.3 Solución de un Sistema Lineal usando LU . . . 67
2.3.4 Cálculo de la Inversa y el Determinante usando LU . . . 69
2.4 Factorización de Cholesky . . . 71
2.4.1 Solución de un Sistema Lineal usando Cholesky . . . 74
2.4.2 Cálculo del Determinante y la Inversa usando Cholesky . . . 75
2.5 Factorización *QR* . . . 76
2.5.1 Solución de un Sistema Lineal usando *QR* . . . 82
2.5.2 Cálculo del Determinante y la Inversa usando *QR* . . . 82

2.6 Descomposición en Valores Singulares . . . 82
2.7 Métodos Iterativos . . . 86
2.7.1 Jacobi, Gauss-Seidel y SOR . . . 86
2.7.2 Métodos Basados en Subespacios de Krylov . . . 97
2.8 Precondicionamiento . . . 107
2.9 Sistemas Sobredeterminados . . . 109
2.10 Ejercicios . . . 112

3 Interpolación y Ajuste de Curvas 117
3.1 Interpolación Polinomial . . . 118
3.1.1 Polinomio Interpolante en la Forma de Lagrange . . . 120
3.1.2 Polinomio Interpolante en la Forma de Newton . . . 122
3.1.3 Tabla de Diferencias Divididas . . . 125
3.1.4 Error de Interpolación . . . 130
3.1.5 Interpolación de Hermite . . . 136
3.2 Interpolación Polinomial a Trozos . . . 138
3.2.1 Splines de orden k . . . 141
3.3 Interpolación Trigonométrica y la FFT . . . 145
3.3.1 Transformada Rápida de Fourier . . . 150
3.3.2 Interpolación Trigonométrica en un Intervalo $[a,b)$ . . . 155
3.4 Ajuste de Curvas . . . 157
3.4.1 Ajuste Polinomial de Datos . . . 157
3.4.2 Ajuste con Funciones No Lineales . . . 161
3.5 Aplicaciones . . . 168
3.5.1 Ajuste de Datos de Medición . . . 168
3.6 Ejercicios . . . 173

4 Ecuaciones No Lineales 178
4.1 Raíces o Ceros de una Función Escalar . . . 178
4.1.1 Método de Bisección . . . 179
4.1.2 Método de Newton . . . 183
4.1.3 Método de la Secante . . . 187
4.1.4 Método de Müller . . . 190
4.1.5 Iteración de Punto Fijo . . . 193
4.2 Sistemas de Ecuaciones no Lineales . . . 198
4.2.1 Iteración de Punto Fijo . . . 201
4.2.2 Método de Newton . . . 203
4.2.3 Métodos Quasi-Newton . . . 206
4.3 Aplicaciones . . . 216
4.3.1 Diseño de Líneas de Transmisión . . . 216
4.4 Ejercicios . . . 220

5 Diferenciación e Integración Numérica 225
5.1 Diferenciación Numérica . . . 225
5.1.1 Fórmulas de Diferencias Finitas . . . 227
5.2 Integración Numérica . . . 232
5.2.1 Fórmulas de Newton-Cotes . . . 233
5.2.2 Método de los Coeficientes Indeterminados . . . 240

5.2.3 Cuadraturas de Gauss-Legendre . . . 243
5.2.4 Cuadraturas Adaptativas . . . 247
5.3 Ejercicios . . . 253

6 Cálculo de Autovalores y Autovectores 256
6.1 Propiedades Básicas y Algunos Resultados Útiles . . . 256
6.2 Método de las Potencias y sus Variantes . . . 258
6.2.1 Método de las Potencias Inverso . . . 263
6.2.2 Método de las Potencias Inverso con Desplazamiento . . . 265
6.2.3 Método del Cociente de Rayleigh . . . 266
6.2.4 Extensiones al Caso No Simétrico . . . 267
6.3 Método *QR* . . . 268
6.4 Ejercicios . . . 278

7 Solución Numérica de Ecuaciones Diferenciales Ordinarias 282
7.1 Problema de Valor Inicial . . . 282
7.2 Métodos de un Paso . . . 285
7.2.1 Métodos de Taylor . . . 285
7.2.2 Métodos de Runge-Kutta . . . 290
7.3 Métodos Multipaso . . . 298
7.3.1 Fórmulas de Adams-Bashforth . . . 299
7.3.2 Fórmulas de Adams-Moulton . . . 300
7.3.3 Orden de un Método Multipaso . . . 300
7.4 Métodos Predictor-Corrector . . . 303
7.5 Convergencia, Consistencia y Estabilidad . . . 306
7.6 Sistemas de Ecuaciones Diferenciales . . . 312
7.7 Ecuaciones de Orden Superior . . . 319
7.8 Problemas de Valores en la Frontera . . . 322
7.8.1 Método de Disparo . . . 323
7.8.2 Método de Diferencias Finitas . . . 325
7.9 Ejercicios . . . 329

8 Optimización Numérica 334
8.1 Optimización sin Restricciones . . . 335
8.1.1 Condiciones de Optimalidad . . . 335
8.1.2 Esquema Básico de los Métodos de Optimización . . . 336
8.1.3 Método de Cauchy . . . 341
8.1.4 Método de Newton . . . 349
8.1.5 Métodos de Gradientes Conjugados . . . 353
8.1.6 Métodos Quasi-Newton . . . 360
8.2 Ejercicios . . . 365

Bibliografía 368

Antes de leer y usar los programas de este libro

Todos los programas en este libro se escribieron en el lenguaje Python versión 3.6 (ver página Web para detalles: https://www.python.org/), para una computadora personal. Los programas del libro conforman el módulo `calnum`, que está disponible en el sitio Web:

https://sites.google.com/view/metnumingucv

A lo largo del libro se asume que todos los objetos del módulo `calnum` están cargados previamente, a través de la instrucción:

```
>>> from calnum import *
```

con la cual se cargan todos los programas del módulo.

En el libro también se utilizan las siguientes librerías científicas y gráficas de Python:

- `Numpy`: http://www.numpy.org/
- `Scipy`: https://www.scipy.org/
- `Mapplotlib`: https://matplotlib.org/

Los objetos de la librería `Numpy` se utilizan con el prefijo `np` (siguiendo la sintaxis de una clase), ya que se asume que `Numpy` se ha cargado previamente con la instrucción:

```
>>> import numpy as np
```

1 Herramientas Matemáticas Básicas

En este libro se usan sistemáticamente conceptos matemáticos básicos que el lector debería saber, es posible que no los recuerde de forma inmediata. Emplearemos este capítulo para recordar tales conceptos e introducir nuevos conceptos que pertenecen al Análisis Numérico. Comenzamos condensando nociones que son típicas de los cursos de Álgebra Lineal y Cálculo, reformulándolas de manera adecuada para su uso en el Cálculo Científico. Solo comentaremos resultados fundamentales, por lo cual hemos evitado las demostraciones de los mismos, ya que esto escapa del objetivo del libro. Finalizamos el capítulo calculando con números, esto es, exploramos la aritmética en computadora. En las siguientes referencias se pueden encontrar las demostraciones de cada uno de los teoremas que enunciaremos en este capítulo [1, 28, 19, 31, 34, 14, 6].

1.1 Conceptos Básicos del Álgebra Lineal

En esta sección haremos una revisión de los conceptos básicos del algebra lineal tales como: vectores, espacio vectorial, producto escalar, norma, matrices, inversa y determinante de una matriz, norma de una matriz, autovalores y autovectores.

1.1.1 Espacio Vectorial, Producto Escalar y Normas Vectoriales

El concepto de espacio vectorial es uno de los más importantes en las Matemáticas.

Definición 1.1. Un conjunto no vacío V de elementos $\mathbf{x},\mathbf{y},\mathbf{z},\ldots$ se llama *espacio vectorial*, cuando satisface las siguientes condiciones:

I. Para cualesquiera dos elementos $\mathbf{x},\mathbf{y} \in V$ está definido unícovamente un tercer elemento $\mathbf{w} \in V$, llamado suma de ellos y denotado por $\mathbf{x}+\mathbf{y}$ tal que
   1) $\mathbf{x}+\mathbf{y} = \mathbf{y}+\mathbf{x}$ (conmutatividad),
   2) $\mathbf{x}+(\mathbf{y}+\mathbf{z}) = (\mathbf{x}+\mathbf{y})+\mathbf{z}$ (asociatividad),
   3) existe en V un elemento 0 tal que $\mathbf{x}+0 = \mathbf{x}$ para todo $\mathbf{x} \in V$,
   4) para todo $\mathbf{x} \in V$ existe un elemento $-\mathbf{x}$ tal que $x+(-\mathbf{x}) = 0$.

II. Para cualquier número α y cualquier elemento $\mathbf{x} \in V$ está definido el elemento $\alpha\mathbf{x} \in V$ de manera que:
   1) $\alpha(\beta\mathbf{x}) = (\alpha\beta)\mathbf{x}$, para cualquier número β,
   2) $1\cdot\mathbf{x} = \mathbf{x}$.

III. Las operaciones de adición y multiplicación están relacionadas entre sí mediante las leyes distributivas:
   1) $(\alpha+\beta)\mathbf{x} = \alpha\mathbf{x}+\beta\mathbf{x}$, para cualesquiera números α y β,
   2) $\alpha(\mathbf{x}+\mathbf{y}) = \alpha\mathbf{x}+\alpha\mathbf{y}$, para cualquier número α.

A los elementos de un espacio vectorial se les acostumbra llamar *vectores*. Veamos dos ejemplos de espacios vectoriales, dejando como ejercicio para el lector la comprobación, en cada uno de ellos, de los axiomas enunciados en la Definición 1.1.

Ejemplo 1.1.

1. El conjunto de los números reales, denotado por $\mathbb{R}$, con las operaciones habituales de adición y multiplicación representa un espacio vectorial.

2. El conjunto formado por las n-uplas de números reales, esto es, $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, donde $x_i \in \mathbb{R}$ para todo $i = 1, 2, \ldots, n$, es un espacio vectorial. Éste se denota por $\mathbb{R}^n$. Los elementos de $\mathbb{R}^n$ también se pueden representar como un *vector columna*,

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}.$$

Por comodidad de notación, a lo largo del libro denotaremos un vector de $\mathbb{R}^n$ como una n-upla o un vector columna.

Definición 1.2. Decimos que un conjunto no vacío W de un espacio vectorial V es un subespacio de V si, y sólo si W es un espacio vectorial. En particular, el conjunto W de las combinaciones lineales de un sistema de p vectores, $\{\mathbf{v}_1, \ldots, \mathbf{v}_p\}$, es un subespacio de V, llamado *subespacio generado* por el sistema de vectores y es denotado por

$$\begin{aligned} W &= \operatorname{span}(\mathbf{v}_1, \ldots, \mathbf{v}_p) \\ &= \{\mathbf{v} = \alpha_i \mathbf{v}_1 + \cdots + \alpha_p \mathbf{v}_p : \alpha_i \in \mathbb{R}\}. \end{aligned}$$

Definición 1.3. Un sistema de vectores $\{\mathbf{v}_1, \ldots, \mathbf{v}_m\}$ de un espacio vectorial V es llamado *linealmente independiente* si la relación

$$\alpha_i \mathbf{v}_1 + \cdots + \alpha_m \mathbf{v}_m = 0$$

con $\alpha_1, \ldots, \alpha_m \in \mathbb{R}$ implica que $\alpha_1 = \cdots = \alpha_m = 0$. De lo contrario, el sistema se dice *linealmente dependiente*, esto es, existen escalares $\alpha_1, \ldots, \alpha_m$ no todos nulos tales que $\alpha_i \mathbf{v}_1 + \cdots + \alpha_m \mathbf{v}_m = 0$.

Definición 1.4. Si en un espacio vectorial V se pueden encontrar n vectores linealmente independientes y cualesquiera $n+1$ vectores de este espacio son linealmente dependientes, entonces se dice que el espacio V tiene *dimensión n*. Se llama base de un espacio vectorial V de n dimensiones, a todo conjunto de n vectores linealmente independientes. Se dice que cualquier espacio vectorial que tenga una base con un número finito de elementos es de *dimensión finita*. Otros espacios vectoriales se conocen como de dimensión infinita.

Es fácil comprobar que $\mathbb{R}^n$ es de dimensión n, además, una base de $\mathbb{R}^n$, denominada *base canónica*, es el conjunto de vectores $\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n$ definidos por

$$\mathbf{e}_i = (0, 0, \ldots, 0, \underbrace{1}_{\substack{\uparrow \\ i\text{-ésima componente}}}, 0, \ldots, 0),$$

es decir, todas las componentes de $\mathbf{e}_i$ son nulas salvo la i-ésima que es igual a 1.

Pasemos ahora a definir producto escalar, con el cual podremos introducir el concepto de vectores ortogonales o perpendiculares.

Definición 1.5. El *producto escalar* de dos vectores $\mathbf{x}$ e $\mathbf{y}$ del espacio vectorial $\mathbb{R}^n$, es un número real $\langle \mathbf{x}, \mathbf{y} \rangle$ que verifica las siguientes condiciones:

1) $\langle \mathbf{x}, \mathbf{y} \rangle = \langle \mathbf{y}, \mathbf{x} \rangle$,

2) $\langle \mathbf{x}, \mathbf{y} + \mathbf{z} \rangle = \langle \mathbf{x}, \mathbf{y} \rangle + \langle \mathbf{x}, \mathbf{z} \rangle$, para todo $\mathbf{z} \in V$,

3) $\langle \lambda \mathbf{x}, \mathbf{y} \rangle = \lambda \langle \mathbf{x}, \mathbf{y} \rangle$, para todo escalar λ,

4) $\langle \mathbf{x}, \mathbf{x} \rangle \geq 0$ y $\langle \mathbf{x}, \mathbf{x} \rangle = 0$ sólo si $\mathbf{x} = 0$.

En adelante, consideraremos el espacio vectorial $\mathbb{R}^n$. El producto escalar en $\mathbb{R}^n$ que utilizaremos a lo largo del libro se define como

$$\langle \mathbf{x}, \mathbf{y} \rangle \equiv \mathbf{x}^T \mathbf{y} = \sum_{i=1}^{n} x_i y_i,$$

Se deja como ejercicio verificar que el producto escalar $\langle \mathbf{x}, \mathbf{y} \rangle \equiv \mathbf{x}^T \mathbf{y}$ en $\mathbb{R}^n$ satisface los axiomas de la Definición 1.5

para $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ e $\mathbf{y} = (y_1, y_2, \ldots, y_n)$, donde $\mathbf{x}^T$ se lee traspuesta de $\mathbf{x}$, cuyo concepto se introducirá en la Sección 1.1.2.

Otro concepto importante en álgebra lineal es la norma o longitud de un vector. Seguidamente mostramos la definición formal de norma.

Definición 1.6. Una *norma vectorial*, denotada por el símbolo $\|\cdot\|$, es una función real definida en el espacio vectorial $\mathbb{R}^n$, que satisface las siguientes propiedades:

1) $\|\mathbf{x}\| \geq 0$, para $\mathbf{x} \in \mathbb{R}^n$ y $\|\mathbf{x}\| = 0$ si y sólo si $\mathbf{x} = \mathbf{0}$.

2) $\|\alpha \mathbf{x}\| = |\alpha| \, \|\mathbf{x}\|$, para $\mathbf{x} \in \mathbb{R}^n$ y $\alpha \in \mathbb{R}$.

3) $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$, para $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$.

En palabras, las condiciones de la Definición 1.6 requieren que: 1) la norma de un vector no nulo es positiva, 2) la norma de un vector escalado es igual a la norma del vector escalado por el valor absoluto de la misma cantidad, y 3) la norma de la suma vectorial no excede la suma de las normas de sus partes.

Las normas vectoriales más comúnmente usadas son los casos especiales de las normas de Hölder:

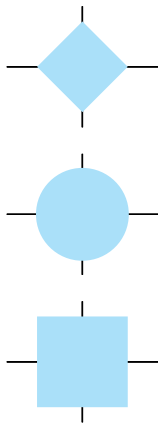
$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}, \qquad 1 \leq p \leq \infty.$$

Los casos particulares de $p = 1$, $p = 2$ y $p = \infty$, son las normas más importantes en la práctica y las mismas se definen a continuación. La bola cerrada $\{\mathbf{x} \in \mathbb{R}^n : \|\mathbf{x}\| \leq 1\}$ correspondiente de cada norma se muestra a la derecha para el caso $n = 2$.

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{n} |x_i| = |x_1| + |x_2| + \cdots + |x_n|,$$

$$\|\mathbf{x}\|_2 = \left(\sum_{i=1}^{n} x_i^2 \right)^{1/2} = \left(x_1^2 + x_2^2 + \cdots + x_n^2 \right)^{1/2},$$

$$\|\mathbf{x}\|_\infty = \max_{i=1,\ldots,n} |x_i|.$$



La norma-2 también se denomina *norma Euclídea*. Las normas que más emplearemos en este libro son la norma-2 y su norma matricial inducida, que más adelante definiremos.

El concepto de ortogonalidad es importante en muchas aplicaciones, particularmente en álgebra lineal.

Definición 1.7. Dos vectores $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, se dicen *ortogonales* si

$$\mathbf{x}^T \mathbf{y} = \sum_{i=1}^{n} x_i y_i = 0.$$

Un vector $\mathbf{x}$ para el cual $\|\mathbf{x}\|_2 = 1$ se dice *normalizado*. El conjunto de vectores $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$ se dice *ortonormal*, si ellos son mutuamente ortogonales y normalizados, esto es,

$$\mathbf{x}_i^T \mathbf{x}_j = \begin{cases} 0, & \text{si } i \neq j, \\ 1, & \text{si } i = j. \end{cases}$$

Ejemplo 1.2. La base canónica de $\mathbb{R}^n$ es un conjunto ortonormal. En efecto, se tiene que

$$\mathbf{e}_i^T \mathbf{e}_j = (0, \ldots, 0, \underset{\substack{\uparrow \\ i\text{-ésima componente}}}{1}, 0, \ldots, 0)^T (0, \ldots, 0, \underset{\substack{\uparrow \\ j\text{-ésima componente}}}{1}, 0, \ldots, 0) = 0,$$

siempre que $i \neq j$; además, es fácil ver que $\mathbf{e}_i^T \mathbf{e}_i = 1$.

El ángulo entre dos vectores de $\mathbb{R}^n$ se define de la siguiente manera.

Definición 1.8. El águlo θ entre dos vectores $\mathbf{x}$ e $\mathbf{y}$ de $\mathbb{R}^n$ está dado por

$$\cos(\theta) = \frac{\mathbf{x}^T \mathbf{y}}{\|\mathbf{x}\|_2 \|\mathbf{y}\|_2}.$$

De esta forma, dos vectores $\mathbf{x}$ e $\mathbf{y}$ son ortogonales si el ángulo entre ellos es $\theta = 90°$.

Una propiedad importante de las normas de Hölder es la *desigualdad de Hölder*,

$$\left|\mathbf{x}^T\mathbf{y}\right| \leq \|\mathbf{x}\|_p \, \|\mathbf{y}\|_q,$$

donde

$$\frac{1}{p} + \frac{1}{q} = 1.$$

Un caso especial de la desigualdad de Hölder es la *desigualdad de Cauchy-Schwarz*,

$$\left|\mathbf{x}^T\mathbf{y}\right| \leq \|\mathbf{x}\|_2 \, \|\mathbf{y}\|_2. \tag{1.1}$$

En muchos métodos numéricos es necesario construir un sistema de vectores ortogonales. El proceso de Gram-Schmidt se utiliza para generar un conjunto de vectores ortogonales (u ortonormales si se normaliza) a partir de un conjunto de vectores linealmente independientes.

Teorema 1.1 (Proceso de Gram-Schmidt)**.** *Sea* $\{\mathbf{v}_1, \ldots, \mathbf{v}_m\}$ *un sistema de vectores linealmente independientes de* $\mathbb{R}^n$*. Para* $i = 1, \ldots, m$ *se definen los vectores* $\mathbf{u}_i$ *de manera recursiva,*

$$\mathbf{u}_i = \frac{\mathbf{v}_i - \sum_{j=1}^{i-1} (\mathbf{v}_i^T \mathbf{u}_j)\mathbf{u}_j}{\left\| \mathbf{v}_i - \sum_{j=1}^{i-1} (\mathbf{v}_i \mathbf{u}_j)\mathbf{u}_j \right\|_2}.$$

Entonces, $\{\mathbf{u}_1, \ldots, \mathbf{u}_m\}$ *es un sistema de vectores ortonormales.*

1.1.2 Matrices

Una matriz es un concepto muy importante en álgebra lineal, que tiene muchas aplicaciones en ingeniería y en las ciencias aplicadas.

Definición 1.9. Una *matriz* real $A = (a_{ij})$ de orden $m \times n$ es un arreglo rectangular de números reales a_{ij}, $i = 1, \ldots, m$, $j = 1, \ldots, n$. El conjunto de todas las matrices reales de orden $m \times n$ se denota por $\mathbb{R}^{m \times n}$. Las matrices de orden $m \times 1$ se llaman *vectores columna* o simplemente *vectores*; en cambio, las matrices de orden $1 \times n$ se llaman *vectores fila*.

Definición 1.10. La *traspuesta* de una matriz $A \in \mathbb{R}^{m \times n}$ es la matriz $A^T \in \mathbb{R}^{n \times m}$ tal que $(A^T)_{ij} = a_{ji}$.

En el siguiente ejemplo mostramos algunas matrices y sus traspuestas.

Ejemplo 1.3. Observe las matrices $A \in \mathbb{R}^{3\times 3}$, $B \in \mathbb{R}^{1\times 4}$ y $C \in \mathbb{R}^{3\times 1}$, con sus respectivas traspuestas.

$$A = \begin{bmatrix} 3 & -1.1 & 0 \\ 0 & 2 & 0.4 \\ -1 & 1 & 2.1 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & -3.4 & \frac{1}{2} & 9 \end{bmatrix}, \quad C = \begin{bmatrix} 4 \\ -0.001 \\ 2 \end{bmatrix},$$

$$A^T = \begin{bmatrix} 3 & 0 & -1 \\ -1.1 & 2 & 1 \\ 0 & 0.4 & 2.1 \end{bmatrix}, \quad B^T = \begin{bmatrix} 1 \\ -3.4 \\ 1/2 \\ 9 \end{bmatrix}, \quad C^T = \begin{bmatrix} 4 & -0.001 & 2 \end{bmatrix}.$$

Aprovechemos este ejemplo para comenzar a trabajar con Python. Las matrices A, B y C pueden ser creadas y manipuladas en Python. Para ello, se utiliza la librería `Numpy`. Seguidamente se muestra la manera de hacerlo. Primero, en el *Shell* (escritorio de trabajo) de Python, cargamos la librería `Numpy`.

```
>>> import numpy as np
```

Con estas instrucciones se carga la librería `Numpy` con el nombre `np`. Todas las funciones disponibles en `Numpy` se utilizan con la siguiente sintaxis: `np.funcion` (siguiendo las reglas de una clase). A lo largo del escrito utlizaremos el prefijo `np` para denotar la librería `Numpy`, la cual, por supuesto, debe cargarse previamente con el nombre `np`. Ahora bien, para crear una matriz se pueden utilizar los comandos `array` o `matrix` de `Numpy`. Por ejemplo, las matrices A, B y C dadas arriba se crean de la siguiente manera:

```
>>> A = np.array([[3,-1.1,0],[0,2,0.4],[-1,1,2.1]])
>>> B = np.array([1,-3.4,0.5,9])
>>> C = np.array([[4],[-0.001],[2]])
>>> print(A)
[[ 3.   -1.1   0. ]
 [ 0.    2.    0.4]
 [-1.    1.    2.1]]
>>> print(B)
[ 1.   -3.4   0.5   9. ]
>>> print(C)
[[  4.00000000e+00]
 [ -1.00000000e-03]
 [  2.00000000e+00]]
```

Para hallar la traspuesta de una matriz se utiliza uno de los comandos `.transpose()` o `.T`, como se indica a continuación:

```
>>> A.T
array([[ 3. ,  0. , -1. ],
       [-1.1,  2. ,  1. ],
       [ 0. ,  0.4,  2.1]])
>>> C.transpose()
array([[  4.00000000e+00,  -1.00000000e-03,   2.00000000e+00]])
```

Definamos las principales operaciones con matrices.

Definición 1.11. Sean $A = (a_{ij})$, $B = (b_{ij})$ y $C = (c_{ij})$ matrices.

- *Suma*: La suma de las matrices $A \in \mathbb{R}^{m\times n}$ y $B \in \mathbb{R}^{m\times n}$, es una matriz $C = A + B$ tal que

$$c_{ij} = a_{ij} + b_{ij}, \quad i = 1,\dots,m, \quad j = 1,\dots,n.$$

- *Multiplicación por un escalar*: La multiplicación de un número real α por una matriz $A \in \mathbb{R}^{m\times n}$, es una matriz $C = \alpha A$ tal que

$$c_{ij} = \alpha a_{ij}, \quad i = 1,\dots,m, \quad j = 1,\dots,n.$$

- *Multiplicación por otra matriz*: La multiplicación de dos matrices $A \in \mathbb{R}^{m\times p}$ y $B \in \mathbb{R}^{p\times n}$, es una matriz $C = AB$ tal que $C \in \mathbb{R}^{m\times n}$ y

$$c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}, \quad i = 1,\dots,m, \quad j = 1,\dots,n.$$

En general, el producto de matrices no es conmutativo.

El conjunto de las matrices reales $\mathbb{R}^{m\times n}$ con las operaciones algebraicas dadas en la definición anterior, conforma un espacio vectorial.

Ejemplo 1.4.

$$\begin{bmatrix} -1 & 3 \\ 5 & 4 \\ 0 & -2 \end{bmatrix} + \begin{bmatrix} 2 & 5 \\ -6 & -2 \\ 8 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 8 \\ -1 & 2 \\ 8 & 4 \end{bmatrix}, \qquad 4\begin{bmatrix} -1 & 3 \\ 5 & 4 \\ 0 & -2 \end{bmatrix} = \begin{bmatrix} -4 & 12 \\ 20 & 16 \\ 0 & -8 \end{bmatrix},$$

$$\begin{bmatrix} 3 & -1 & 0 \\ 0 & 2 & 4 \\ -1 & 1 & 2 \end{bmatrix} \begin{bmatrix} 2 \\ -6 \\ 8 \end{bmatrix} = \begin{bmatrix} 12 \\ 20 \\ 8 \end{bmatrix}.$$

En Python las operaciones: $A+B$, $A-B$, aC y AB, se realizan respectivamente como: `A+B`, `A-B`, `a*C` y `A.dot(B)`. Las operaciones anteriores se realizan de la siguiente manera:

```
>>> np.array([[-1,3],[5,4],[0,-2]])+np.array([[2,5],[-6,-2],[8,6]])
array([[ 1,  8],
       [-1,  2],
       [ 8,  4]])
>>> 4*np.array([[-1,3],[5,4],[0,-2]])
array([[-4, 12],
       [20, 16],
       [ 0, -8]])
>>> np.array([[3,-1,0],[0,2,4],[-1,1,2]]).dot(np.array([[2],[-6],[8]]))
array([[12],
       [20],
       [ 8]])
```

Definición 1.12. Una matriz $A \in \mathbb{R}^{m\times n}$ es *cuadrada* si tiene el mismo número de columnas y filas, es decir, si $m = n$. Cuando una matriz cuadrada A tiene la propiedad que $A^T = A$, se dice que A es *simétrica*.

Una matriz cuadrada importante es la matriz identidad de orden n dada por:

$$I = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix},$$

en otras palabras, $I = (\delta_{ij})$ es una matriz de orden n tal que

$$\delta_{ij} = \begin{cases} 1, & i = j, \\ 0, & i \neq j. \end{cases}$$

La matriz identidad satisface la igualdad $IA = AI = A$, para toda matriz cuadrada A de orden n. Con el comando `eye(n)` de la librería `Numpy` se crea la matriz identidad de orden n. A continuación creamos la matriz identidad de orden 3.

```
>>> np.eye(3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

Definición 1.13. Una matriz cuadrada A de orden n se dice *invertible* o *no singular*, si existe una matriz cuadrada B de orden n tal que $BA = AB = I$. B es llamada *matriz inversa* de A y se denota por A^{-1}. Una matriz cuando no es invertible se denomina *singular*.

Teorema 1.2. *Si A y B son matrices no singulares de orden n, entonces:*

i) $(A^{-1})^{-1} = A$.

ii) $(AB)^{-1} = B^{-1}A^{-1}$.

Ahora definiremos un número asociado con la matriz cuadrada $A = (a_{ij})$ de orden n, llamado *determinante* de la matriz y denotado por $\det(A)$. El determinante de orden n se define en términos de los determinantes de orden $n-1$, junto con la condición que el determinante de una matriz 1×1 de un sólo elemento, $A = (a)$, está dado por

$$\det(A) = a.$$

Esto significa que el determinante de una matriz de orden 2×2 está definido en términos de determinantes de orden 1×1. El determinante de una matriz de orden 3×3 está definido en términos de determinantes de orden 2×2, y así sucesivamente. Para precisar, introducimos las siguientes definiciones.

Definición 1.14. El determinante de una matrix de orden $(n-1)$ obtenida eliminando la i-ésima fila y la j-ésima columna de una matrix $A=(a_{ij})$ de orden n, es llamado el *menor* de a_{ij} y es denotado por M_{ij}. El número

$$A_{ij}=(-1)^{i+j}M_{ij},$$

es llamado el *cofactor* de a_{ij}.

Definición 1.15. El determinante de una matriz cuadrada A de orden n está definido por la expresión

$$\det(A)=\sum_{j=1}^{n}a_{1j}A_{1j},$$

conjuntamente con la definición del $\det(A)$ para $n=1$. En otras palabras, el determinante de A es la suma de los productos de los elementos en la primera fila de la matriz por sus respectivos cofactores.

Teorema 1.3. *Si A y B son matrices cuadradas de orden n, entonces:*

i) $\det(AB)=\det(BA)=\det(A)\det(B)$.
ii) $\det(A^T)=\det(A)$.
iii) $\det(\alpha A)=\alpha^n\det(A)$.
iv) $\det(I)=1$.
v) *A es no singular si, y sólo si* $\det(A)\neq 0$.

Definición 1.16. Sean $A=(a_{ij})$ una matriz cuadrada y A_{ij} el cofactor de a_{ij}. La traspuesta de la matriz de cofactores se conoce como la *adjunta* de la matriz A, denotada por $\operatorname{adj}A$,

$$\operatorname{adj}A=(A_{ji}).$$

Teorema 1.4. *Si A es una matriz no singular, entonces la inversa de A está dada por*

$$A^{-1}=\frac{1}{\det A}\operatorname{adj}A.$$

Definición 1.17. La traza de una matriz cuadrada $A=(a_{ij})$ de orden n es la suma de los elementos de su diagonal, esto es, $\operatorname{tr}(A)=\sum_{i=1}^{n}a_{ii}$.

En el siguiente ejemplo ilustramos los conceptos vistos hasta ahora. Aprovechemos en este ejemplo para introducir los comandos de Python que permiten calcular la inversa, el determinante y la traza de una matriz.

Ejemplo 1.5. Consideremos la matriz

$$A=\begin{bmatrix}1 & 0 & 0\\ 0 & 2 & 1\\ 0 & -1 & 3\end{bmatrix}$$

Se tiene que la inversa, la traza y el determinante de A son:

$$A^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 3/7 & -1/7 \\ 0 & 1/7 & 2/7 \end{bmatrix}, \text{ tr}(A) = 6, \text{ y } \det(A) = 1 \cdot \det(A_{11}) = 2 \cdot \det(3) - 1 \cdot \det(-1) = 7.$$

En Python, el determinante y la inversa de una matriz se calculan, respectivamente, con los comandos `det` e `inv` del módulo `linalg` de la librería `Numpy`, y La traza se calcula con el comando `trace` de la librería `Numpy`. Con la matriz A anterior se tiene:

```
>>> from numpy.linalg import inv, det
>>> from numpy import trace
>>> A = np.array([[1,0,0],[0,2,1],[0,-1,3]])
>>> det(A)
7.0000000000000009
>>> trace(A)
6
>>> inv(A)
array([[ 1.        ,  0.        ,  0.        ],
       [ 0.        ,  0.42857143, -0.14285714],
       [ 0.        ,  0.14285714,  0.28571429]])
```

Observe que las instrucciones

```
from numpy.linalg import inv, det
from numpy import trace
```

cargan solamente las funciones `det`, `inv` y `trace`, sin cargar las otras funciones de la librería.

En algunas aplicaciones, es necesario calcular la inversa de una matriz B la cual difiere de una matriz singular A solamente en una perturbación de rango 1. La fórmula de Sherman-Morrison permite calcular la inversa de B.

Teorema 1.5 (Fórmula de Sherman-Morrison)**.** *Si* $\mathbf{u}$ *y* $\mathbf{v}$ *son dos vectores de* $\mathbb{R}^n$ *y* A *es una matriz no singular de orden* n*, entonces*

$$(A - \mathbf{u}\mathbf{v}^T)^{-1} = A^{-1} + \alpha(A^{-1}\mathbf{u}\mathbf{v}^T A^{-1}), \quad \textit{con } \alpha = \frac{1}{(1 - \mathbf{v}^T A^{-1}\mathbf{u})}, \quad \textit{si } \mathbf{v}^T A^{-1}\mathbf{u} \neq 1.$$

Es importante subrayar que son necesarias $n!$ operaciones (multiplicaciones y sumas) para calcular el determinante de una matriz de orden n empleando la definición recursiva. Por ello el cálculo del determinante es prohibitivo cuando la dimensión de la matriz es grande ($n \gg 1000$); por ejemplo, si $n = 100$, entonces son necesarias aproximadamente 9.3326×10^{157} operaciones para calcular el determinante. Como el cálculo de la inversa de una matriz puede involucrar el cálculo de determinantes, entonces obtener la inversa de matrices grandes es computacionalmente costoso. De esta forma, si la inversa o el determinante no están disponibles, mejor sería no calcularlos directamente usando su definición, sino buscar métodos alternativos, más adelante hablaremos de ello.

Otra clase de matrices cuadradas importantes son las matrices *definidas positivas*.

Definición 1.18. Una matriz cuadrada A de orden n es definida positiva, si $\mathbf{x}^T A\mathbf{x} > 0$ para todo vector $\mathbf{x} \in \mathbb{R}^n$ no nulo.

Ejemplo 1.6. La matriz A del Ejemplo 1.5 es definida positiva. En efecto, para todo $0 \neq \mathbf{x} \in \mathbb{R}^3$ se tiene:

$$\mathbf{x}^T A\mathbf{x} = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 1 \\ 0 & -1 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_2 \end{bmatrix} = x_1^2 + 2x_2^2 + 3x_3^2 > 0.$$

Generalmente, verificar que una matriz es definida positiva a partir de la definición no es una tarea sencilla, ya que la condición $\mathbf{x}^T A\mathbf{x} > 0$ se debe satisfacer para un vector *arbitrario* $\mathbf{x} \neq \mathbf{0}$. Más adelante estudiaremos algunos procedimientos para determinar cuando una matriz es definida positiva.

1.1.3 Normas Matriciales

Una matriz de orden $m \times n$ se puede ver como un vector en el espacio nm-dimensional: a partir de la matriz $A = (a_{ij})$ se construye, por ejemplo, el vector

$$(a_{11}, a_{21}, \ldots, a_{m1}, \ldots, a_{1n}, a_{2n}, \ldots, a_{mn}) \in \mathbb{R}^{mn}.$$

Por tanto, cualquier norma vectorial en $\mathbb{R}^{mn}$ puede medir el "tamaño" de tal matriz. Sin embargo, al tratar con el espacio vectorial de las matrices existen normas especiales que son más útiles que las normas vectoriales definidas previamente. Éstas son las *normas inducidas*.

Definición 1.19. Para una matriz $A \in \mathbb{R}^{m\times n}$, se define la norma de A como

$$\|A\|_{pq} = \max_{\mathbf{x}\in\mathbb{R}^n, \mathbf{x}\neq 0} \frac{\|A\mathbf{x}\|_p}{\|\mathbf{x}\|_q}.$$

La norma matricial $\|\cdot\|_{pq}$ es *inducida* por las normas vectoriales $\|\cdot\|_p$ y $\|\cdot\|_q$.

Las normas matriciales satisfacen las propiedades usuales de las normas, esto es,

1. $\|A\| = 0$, si y sólo si $A = 0$, para $A \in \mathbb{R}^{m\times n}$.
2. $\|\alpha A\| = |\alpha|\,\|A\|$, para $A \in \mathbb{R}^{m\times n}$ y $\alpha \in \mathbb{R}$.
3. $\|A + B\| \leq \|A\| + \|B\|$, para $A, B \in \mathbb{R}^{m\times n}$.

El caso $p = q$ es de interés muy particular y la norma asociada $\|\cdot\|_{pq}$ es simplemente denotada por $\|\cdot\|_p$ y se llama norma p. Los casos más importantes son de nuevo los asociados con $p, q = 1, 2, \infty$. Para $p = 1$ y $p = \infty$, se tienen las siguientes definiciones alternativas:

$$\|A\|_1 = \max_{j=1,\ldots,n} \sum_{i=1}^{m} |a_{ij}|, \qquad \|A\|_\infty = \max_{i=1,\ldots,n} \sum_{j=1}^{n} |a_{ij}|.$$

La norma matricial más importante que no es inducida por una norma vectorial es la *norma de Frobenius*, definida por

$$\|A\|_F = \operatorname{tr}(A^T A) = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}^2 \right)^{1/2}, \qquad \text{para } A \in \mathbb{R}^{m \times n}.$$

Ejemplo 1.7. Para la matriz

$$A = \begin{bmatrix} 1 & 4 & 6 \\ 3 & -1 & 5 \\ 0 & 8 & 7 \end{bmatrix}$$

se tiene

$$\|A\|_1 = \text{máx}\{1+3+0, 4+|-1|+8, 6+5+7\} = 18,$$
$$\|A\|_\infty = \text{máx}\{1+4+6, 3+|-1|+5, 0+8+7\} = 15,$$
$$\|A\|_2 = \frac{1713}{130}, \qquad \|A\|_F = \sqrt{201}.$$

El comando `norm` del módulo `linalg` de la librería `Numpy`, se utiliza para calcular las normas p y la norma de Frobenius. Para la matriz A,

```
>>> A = np.array([[1,4,6],[3,-1,5],[0,8,7]])
```

se tiene que las normas calculadas anteriormente están dadas respectivamente por:

```
>>> np.linalg.norm(A,1)
18.0
>>> np.linalg.norm(A,np.inf)
15.0
>>> np.linalg.norm(A,2)
13.176932423207695
>>> np.linalg.norm(A,'fro')
14.177446878757825
```

1.1.4 Alcance y Rango de una Matriz

Un subespacio importante asociado con una matriz $A \in \mathbb{R}^{m \times n}$ es su *alcance* o *espacio de llegada*, definido por

$$\text{alc}(A) = \{\mathbf{y} \in \mathbb{R}^m : \mathbf{y} = A\mathbf{x},\ \mathbf{x} \in \mathbb{R}^n\}.$$

El *rango* de una matriz A, denotado por $\text{rang}(A)$, es la dimensión del alcance de A, es decir, el número de columnas linealmente independientes de A.

Ejemplo 1.8. El rango de la matriz

$$A_1 = \begin{bmatrix} 1 & 4 & 6 \\ 3 & -1 & 5 \\ 0 & 8 & 7 \end{bmatrix}$$

es rang$(A_1) = 3$. Ahora, el rango de la matriz

$$A_2 = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 0 \\ 1 & 1 & 2 \end{bmatrix}$$

es rang$(A_2) = 2$. Con el comando `matrix_rank` del módulo `linalg` de la librería `Numpy`, se calcula el rango de una matriz en Python:

```
>>> A1 = np.array([[1,4,6],[3,-1,5],[0,8,7]])
>>> A2 = np.array([[1,0,2],[0,1,0],[1,1,2]])
>>> np.linalg.matrix_rank(A1)
3
>>> np.linalg.matrix_rank(A2)
2
```

1.1.5 Matrices Especiales

Algunas matrices tienen estructuras particulares que a menudo son convenientes para propósitos computacionales. La siguiente lista, aunque incompleta, da una idea de estas matrices especiales que desempeñan un papel importante en el análisis numérico y en aplicaciones del cálculo científico.

Matriz Diagonal

Las matrices diagonales son matrices cuadradas de orden n de la forma

$$D = \begin{bmatrix} d_{11} & & & \\ & d_{22} & & \\ & & \ddots & \\ & & & d_{nn} \end{bmatrix},$$

donde $d_{ij} = 0$ para $i \neq j$. Utilizamos la notación $D = \text{diag}(d_{11}, d_{22}, \ldots, d_{nn})$ para indicar que D es una matriz diagonal.

El determinante de una matriz diagonal es el producto de los elementos de su diagonal. Una matriz diagonal D tiene inversa si todos los elementos de su diagonal son distintos de cero y

$$D^{-1} = \begin{bmatrix} d_{11}^{-1} & & & \\ & d_{22}^{-1} & & \\ & & \ddots & \\ & & & d_{nn}^{-1} \end{bmatrix}.$$

Matrices Triangulares

Una matriz cuadrada A de orden n se dice *triangular superior* si $a_{ij} = 0$ para $i > j$, mientras que ésta se dice *triangular inferior* si $a_{ij} = 0$ para $i < j$. Seguidamente se muestra la forma de las

matrices triangular inferior y triangular superior, respectivamente.

$$L = \begin{bmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{bmatrix}, \qquad U = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix}.$$

Recordemos algunas propiedades algebraicas de las matrices triangulares que son fáciles de verificar.

- El determinante de una matriz triangular superior (inferior) es el producto de los elementos de su diagonal;
- la inversa de una matriz triangular inferior (respectivamente, superior) también es una matriz triangular inferior (respectivamente, superior);
- el producto de dos matrices triangulares inferiores (respectivamente, superiores) es una matriz triangular inferior (respectivamente, superior).

Matrices Bidiagonales

Una matriz cuadrada B de orden n se dice *bidiagonal superior* si $b_{ij} = 0$ para $j \neq i$ o $j \neq i+1$, mientras ésta se dice *bidiagonal inferior* si $b_{ij} = 0$ para $j \neq i$ o $j \neq i-1$. Seguidamente se muestra la forma de las matrices bidiagonales inferior y superior, respectivamente.

$$A = \begin{bmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ & \ddots & \ddots & \\ & & a_{n,n-1} & a_{nn} \end{bmatrix}, \qquad B = \begin{bmatrix} b_{11} & b_{12} & & \\ & b_{22} & \ddots & \\ & & \ddots & b_{n-1,n} \\ & & & b_{nn} \end{bmatrix}$$

Matrices Tridiagonales

Una matriz cuadrada T de orden n se dice *tridiagonal* si $t_{ij} = 0$ para i, j tal que $|j-i| > 1$. Seguidamente se muestra la forma de una matriz tridiagonal.

$$T = \begin{bmatrix} t_{11} & t_{12} & & \\ t_{21} & t_{22} & \ddots & \\ & \ddots & \ddots & t_{n-1,n} \\ & & t_{n,n-1} & t_{nn} \end{bmatrix}$$

Matriz de Hessenberg superior

Una matrix $A = (a_{ij})$ se dice *matriz de Hessenberg superior* si ella tiene ceros debajo de la primera subdiagonal principal, es decir, $a_{ij} = 0$ para $i > j+1$. A continuación mostramos ejemplos de matrices de Hessenberg.

$$A = \begin{bmatrix} 1 & 8 & 0 & 1 & -2 \\ 7 & 3 & 8 & -6 & 0 \\ 0 & 2 & -4 & 5 & 1 \\ 0 & 0 & 7 & 3 & 4 \\ 0 & 0 & 0 & -8 & 5 \end{bmatrix}_{5\times 5}, \qquad B = \begin{bmatrix} 1 & -1 & 3 \\ 7 & 3 & 2 \\ 0 & 2 & 5 \\ 0 & 0 & 6 \\ 0 & 0 & 0 \end{bmatrix}_{5\times 3}$$

Matriz Permutación

Una matriz cuadrada se dice *matriz permutación* si sus columnas son una permutación de las columnas de la matriz identidad. Seguidamente se muestran ejemplos.

$$P_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}_{4\times 4}, \quad P_2 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}_{4\times 4}, \quad P_3 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}_{4\times 4}.$$

1.1.6 Autovalores y Autovectores

Sea A una matriz cuadrada de orden n. El número $\lambda \in \mathbb{C}$ se llama *autovalor* de A, si existe un vector no nulo $\mathbf{x} \in \mathbb{R}^n$ tal que $A\mathbf{x} = \lambda\mathbf{x}$. El vector $\mathbf{x}$ es el *autovector* asociado con el autovalor λ y el conjunto de los autovalores de A se denomina *espectro* de A, denotado por $\sigma(A)$. Diremos que $\mathbf{x}$ e $\mathbf{y}$ son respectivamente un *autovector derecho* y un *autovector izquierdo*, asociados al autovalor λ, si se cumple

$$A\mathbf{x} = \lambda\mathbf{x}, \qquad \mathbf{y}^T A = \lambda\mathbf{y}^T.$$

El número λ es la solución de la *ecuación característica*

$$p_A(\lambda) = \det(A - \lambda I) = 0,$$

donde $p_A(\lambda)$ es el *polinomio característico*. Como éste último es un polinomio de grado n con respecto a λ, existen n autovalores de A no necesariamente distintos. Las siguientes identidades se cumplen:

$$\det(A) = \prod_{i=1}^{n} \lambda_i, \qquad \mathrm{tr}(A) = \sum_{i=1}^{n} \lambda_i,$$

y como $\det(A^T - \lambda I) = \det((A - \lambda I)^T) = \det(A - \lambda I)$, se concluye que $\sigma(A^T) = \sigma(A)$. De todo lo anterior se obtiene que una matriz es singular si y sólo si ésta tiene al menos un autovalor nulo, dado que $p_A(0) = \det(A) = \prod_{i=1}^{n} \lambda_i = 0$. Ahora, como A es una matriz real, $p_A(\lambda)$ es un polinomio con coeficientes reales; por lo tanto, los autovalores complejos de A deberán aparecer en pares complejos conjugados.

El módulo máximo de los autovalores de A se denomina *radio espectral* de A y es denotado por: $\rho(A) = \max_{\lambda\in\sigma(A)} |\lambda|$. Seguidamente damos el Teorema de Cayley-Hamilton.

Teorema 1.6 (Cayley-Hamilton)**.** *Si p es el polinomio característico de A, entonces $p(A) = 0$.*

1.2 Conceptos Básicos del Cálculo

Seguidamente realizamos una revisión de los conceptos básicos del Cálculo tales como función, límite, continuidad, derivada, integral, Teorema del Valor Medio, desarrollo de Taylor, entre otros. A lo largo del libro utilizamos la notación habitual de la Teoría de Conjuntos, además, empleamos los siguientes conjuntos: $\mathbb{N}$: conjunto de los números naturales, $\mathbb{Z}$: conjunto de los números enteros, $\mathbb{Q}$: conjunto de los números racionales, $\mathbb{R}$: conjunto de los números reales y $\mathbb{C}$: conjunto de los números complejos.

1.2.1 Funciones de una Variable Real

Para introducir el concepto de función es necesario considerar el *plano cartesiano* o *plano xy*, que consiste en formar el producto cartesiano de $\mathbb{R}\times\mathbb{R}=\mathbb{R}^2$ y es representado geométricamente a través de dos rectas perpendiculares de referencia (llamadas *ejes coordenados*), uno horizontal (llamado eje de las x o de las *abscisas*), y el otro vertical (el eje de las y o de las *ordenadas*); su punto de intersección, se indica por 0 y se denomina *origen*. El par de números (x,y) con $x,y\in\mathbb{R}$, se denomina *par ordenado*.

Definición 1.20. Una *función* f es un conjunto de pares ordenados (x,y) ninguno de los cuales tiene el mismo primer elemento. Escribiremos $y=f(x)$ para indicar que y es *imagen* x bajo la función f; el símbolo $f(x)$ se lee "f de x", "f en x" o "f evaluada en x". El conjunto de x donde f está bien definida (que no se produzca división por cero) se denomina *dominio* y el conjunto de imágenes de f se denomina *rango*.

En el Cálculo elemental se tiene interés en considerar funciones en las que el dominio y el rango son conjuntos de números reales. Estas funciones se llaman *funciones de variable real* o más brevemente *funciones reales* y se pueden representar geométricamente mediante una gráfica en el plano xy. Ahora bien, es conveniente distinguir entre funciones, es decir, es importante saber cuando dos funciones son iguales. Diremos que dos funciones f y g son iguales, si f y g tienen el mismo dominio y $f(x)=g(x)$ para todo x del dominio de f. Seguidamente damos uno ejemplos de funciones de uso común en el cálculo.

Ejemplo 1.9. 1. *Funciones lineales.* Una función f definida para $x\in\mathbb{R}$ mediante la fórmula $f(x)=ax+b$, donde $a,b\in\mathbb{R}$.

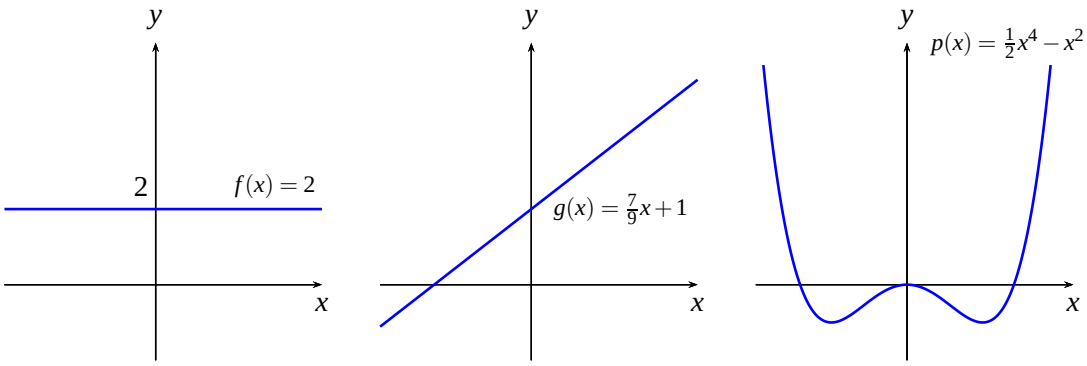
2. *Funciones potenciales*. Una función f definida para $x\in\mathbb{R}$ mediante la fórmula $f(x)=x^n$, donde n es un entero positivo.

3. *Polinomios*. Un polinomio p es una función definida para $x\in\mathbb{R}$ por una ecuación de la forma

$$p(x)=a_0+a_1x+\cdots+a_nx^n=\sum_{k=1}^{n}a_kx^k,$$

donde los números $a_0,a_1,\ldots,a_n$ son los *coeficientes* del polinomio y el entero no negativo n es su *grado* (si $a_n\neq 0$).

En la Figura 1.1 se muestran una parte de las gráficas de las funciones $f(x)=2$, $g(x)=\frac{7}{9}x+1$ y $p(x)=\frac{1}{2}x^4-x^2$.



(a) $f(x) = 2$, función constante (b) $g(x) = \frac{7}{9}x + 1$, función lineal (c) $p(x) = \frac{1}{2}x^4 - x^2$, polinomio de grado 4

Figura 1.1. Tipos de funciones

Para funciones f y g se pueden construir la funciones: suma $f+g$, producto $f \cdot g$ (o fg) y cociente f/g. El dominio de cada una de esta nuevas funciones viene dado por la intersección de los dominios de las funciones involucradas, además, la función f/g, generalmente, no está definida en los números x tales que $g(x) = 0$.

Definición 1.21. Una función f se dice que es *creciente* en un conjunto $S \subset \mathbb{R}$, si $f(x) \geq f(y)$ para $x, y \in S$ con $x < y$. Si se verifica la desigualdad estricta $f(x) < f(y)$ para todo $x < y$ en S se dice que la función es *estrictamente creciente*. Análogamente, una función se dice *decreciente* en S si $f(x) \leq f(y)$ para $x, y \in S$ con $x < y$. Si $f(x) > f(y)$ para todo $x < y$ en S se dice que la función es *estrictamente decreciente*. Una función se denomina *monótona* en S si es creciente en S o decreciente en S. *Estrictamente monótona* significa que f, o es estrictamente creciente en S o es estrictamente decreciente en S.

Describamos ahora el concepto de continuidad, uno de los más importantes en el Cálculo. Para ello es necesario introducir el concepto de límite de una función.

Definición 1.22. Sean f una función definida de un intervalo (a,b) a $\mathbb{R}$, y $x_0 \in (a,b)$. Se dice que $w \in \mathbb{R}$ es un *límite* de f cuando x tiende a x_0, si para todo $\varepsilon > 0$, existe $\delta = \delta(\varepsilon)$ tal que $|f(x) - w| < \varepsilon$ siempre que $0 < |x - x_0| < \delta$. Para indicar que w es un límite de f cuando x tiende a x_0, se escribe

$$\lim_{x \to x_0} f(x) = w.$$

Definición 1.23. Se dice que una función f es *continua* en $x_0 \in \mathbb{R}$ si:

i) f está definida en x_0, y
ii) $\lim_{x \to x_0} f(x) = f(x_0)$.

A continuación damos los teoremas fundamentales sobre límites y algunas propiedades de las funciones continuas.

Teorema 1.7. *Sean f y g dos funciones reales tales que $\lim_{x\to x_0} f(x) = A$ y $\lim_{x\to x_0} g(x) = B$. Entonces:*

i) $\lim_{x\to x_0}[f(x)\pm g(x)] = A \pm B$,

ii) $\lim_{x\to x_0}[f(x)\cdot g(x)] = A\cdot B$,

iii) $\lim_{x\to x_0}[f(x)/g(x)] = A/B$, *si* $B \neq 0$.

Teorema 1.8. *Sean f y g dos funciones reales continuas en el punto x_0. La suma $f+g$, la diferencia $f-g$, y el producto $f\cdot g$ son también continuas en x_0. Si $g(x_0)\neq 0$, entonces f/g también es continua en x_0. Además, si f es continua en x_0 y g es continua en $y_0 = f(x_0)$, entonces la función compuesta $h(x) = g(f(x))$ es continua en x_0.*

Teorema 1.9 (Teorema de Bolzano)**.** *Sea f continua en cada punto del intervalo cerrado $[a,b]$ y supongamos que $f(a)f(b) < 0$. Entonces, existe por lo menos un punto $c \in (a,b)$ tal que $f(c) = 0$.*

Teorema 1.10 (Teorema del valor intermedio)**.** *Sea f continua en cada punto de un intervalo $[a,b]$. Si $x_1 < x_2$ son dos puntos cualesquiera de $[a,b]$ tales que $f(x_1) \neq f(x_2)$, entonces la función f toma todos los valores comprendidos entre $f(x_1)$ y $f(x_2)$ por lo menos una vez en el intervalo (x_1,x_2).*

A continuación describimos los conceptos básicos del cálculo diferencial.

Definición 1.24. Sea f una función definida por lo menos en un intervalo abierto (a,b). La *derivada* de f en $x \in (a,b)$, denotada por $f'(x)$, se define como

$$f'(x) = \lim_{h\to 0} \frac{f(x+h)-f(x)}{h},$$

con tal que el límite exista.

Teorema 1.11. *Sean f y g dos funciones definidas en un conjunto común de números reales S. Supongamos que f y g tienen derivadas en cada punto de S. Entonces, la suma $f+g$, la diferencia $f-g$, el producto fg y el cociente f/g, tienen derivadas que se calculan por las fórmulas:*

i) $(f\pm g)' = f' \pm g'$,

ii) $(f\cdot g)' = f'\cdot g + g'\cdot f$,

iii) $\left(\frac{f}{g}\right)' = \frac{f'\cdot g - g'\cdot f}{g^2}$, *en puntos x tales que* $g(x)\neq 0$.

Además, si $f = u\circ v$ tal que las derivadas $v'(x)$ y $u'(y)$ existen con $y = v(x)$, entonces la derivada $f'(x)$ también existe y está dada por la fórmula

$$f'(x) = u'(v(x))\cdot v'(x),$$

denominada ***regla de la cadena***.

Como el Teorema 1.7, que nos enseña a calcular el límite de la suma, diferencia, producto y cociente de dos funciones, el teorema anterior nos proporciona un conjunto de reglas para el cálculo de derivadas. La derivada tiene muchos usos en el Cálculo, especialmente se puede utilizar para la localización de los máximos y mínimos de las funciones.

Definición 1.25. Se dice que una función real f tiene un *máximo absoluto* en un conjunto $S \subset \mathbb{R}$, si existe por lo menos un punto $c \in S$ tal que $f(x) \leq f(c)$, para todo $x \in S$. Ahora, se dice que f tiene un *máximo relativo* en un punto $c \in S$, si existe un cierto intervalo abierto $I \subset \mathbb{R}$ que contiene a c tal que $f(x) \leq f(c)$, para todo $x \in I \cap S$. Los conceptos de *mínimo absoluto* y *mínimo relativo* se definen del mismo modo con la desigualdad invertida.

Teorema 1.12. *Sea f una función definida en un intervalo abierto I, y supongamos que f tiene un máximo relativo o un mínimo relativo en un punto $c \in I$. Si la derivada $f'(c)$ existe, entonces $f'(c) = 0$.*

A continuación presentamos otros teoremas importantes del cálculo diferencial.

Teorema 1.13 (Teorema del Valor Medio para Derivadas)**.** *Si f es una función continua en todo punto del intervalo $[a,b]$ que tiene derivada en cada punto del intervalo (a,b), entonces existe por lo menos un punto $c \in (a,b)$ para el que*

$$f(a) - f(b) = f(c)(b-a).$$

Teorema 1.14. *Supongamos f es continua en un intervalo $[a,b]$ y que existe la derivada f' en todo punto del intervalo (a,b), excepto en un punto $c \in (a,b)$.*

i) Si $f'(x) > 0$ para todo $x < c$ y $f'(x) < 0$ para todo $x > c$, entonces f tiene un máximo relativo en c.

ii) Si $f'(x) < 0$ para todo $x < c$ y $f'(x) > 0$ para todo $x > c$, entonces f tiene un mínimo relativo en c.

Teorema 1.15. *Sea c un punto crítico de f en un intervalo (a,b); esto es, supongamos que $a < c < b$ y $f'(c) = 0$. Supongamos también que la segunda derivada f'' existe en (a,b). Entonces:*

i) Si $f''(x) < 0$ para todo $x \in (a,b)$, entonces f tiene un máximo relativo en c.

ii) Si $f''(x) > 0$ para todo $x \in (a,b)$, entonces f tiene un mínimo relativo en c.

Seguidamente describimos algunos conceptos básicos del cálculo integral.

Definición 1.26. Una *partición* P de un intervalo $[a,b]$ es un conjunto finito de puntos $x_0, x_1, \ldots, x_n$ tales que

$$a = x_0 \leq x_1 \leq \cdots \leq x_n = b.$$

Escribimos

$$\Delta x_i = x_i - x_{i-1}, \qquad \text{para } i = 1, \ldots, n.$$

Definición 1.27. Sea f una función acotada en el intervalo $[a,b]$, esto es, existen números m y M tales $m \leq f(x) \leq M$ para $x \in [a,b]$. Para cada partición P de $[a,b]$ hacemos

$$M_i = \sup_{x_{i-1} \leq x \leq x_i} f(x),\ m_i = \inf_{x_{i-1} \leq x \leq x_i} f(x),\ U(P,f) = \sum_i^n M_i \Delta x_i,\ L(P,f) = \sum_i^n m_i \Delta x_i.$$

Las *integrales superior e inferior de Riemann* se definen respectivamente por

$$\overline{\int_a^b} f(x)\,dx = \inf U(P,f) \qquad \text{y} \qquad \underline{\int_a^b} f(x)\,dx = \sup L(P,f).$$

Si las integrales superior e inferior son iguales, se dice que f es *integrable* en $[a,b]$ y el valor común de estas integrales se representa con el símbolo

$$\int_a^b f(x)\,dx,$$

que se denomina *integral de Riemann* de f o simplemente *integral* de f en $[a,b]$.

Los siguientes teoremas relacionan la diferenciación con la integración.

Teorema 1.16. *Sea f integrable en el $[a,b]$. Para $x \in [a,b]$ hagamos*

$$F(x) = \int_a^x f(t)\,dt.$$

Entonces, F es continua en $[a,b]$; además, si f es continua en el algún punto $z \in [a,b]$, la derivada $F'(z)$ existe y

$$F'(z) = f(z).$$

La función F del teorema anterior usualmente se denomina *primitiva* de f. Seguidamente damos el Teorema Fundamental del Cálculo.

Teorema 1.17 (Teorema Fundamental del Cálculo)**.** *Si f es integrable en el $[a,b]$ y si existe una función F derivable en $[a,b]$ tal que $F' = f$, entonces*

$$\int_a^b f(x)\,dx = F(b) - F(a).$$

El Teorema de Taylor juega un papel importante en la definición de métodos para la aproximación de funciones, la resolución de ecuaciones diferenciales, entre otros.

Teorema 1.18 (Teorema de Taylor)**.** *Sean f una función que tiene derivada continua de orden $n+1$ en el intervalo $[a,b]$ y $x_0 \in [a,b]$. Entonces, para todo $x \in [a,b]$ existe $\beta = \beta(x)$ tal que*

$$f(x) = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x-x_0)^k + E_n(x),$$

donde $E_n(x) = \frac{f^{(n+1)}(\beta)}{(n+1)!}(x-x_0)^{n+1}$.

1.2.2 Funciones de Varias Variables

Los conceptos de campo escalar, campo vectorial y transformación lineal, son muy importantes para el estudio de métodos para la aproximación de funciones, sistemas de ecuaciones lineales y no lineales y optimización.

Definición 1.28. Sea D un subconjunto de $\mathbb{R}^n$. Una función f de D en $\mathbb{R}$ se llama un *campo escalar* o *una función real de n variables*. La función f asigna a cada vector $\mathbf{x} \in D \subset \mathbb{R}^n$ un valor real $f(\mathbf{x})$.

Definición 1.29. Una función $\mathbf{f} : D \subset \mathbb{R}^n \to \mathbb{R}^m$ se llama *función vectorial de varias variables*. Si $n = m > 1$, la función se denomina *campo vectorial*. La función $\mathbf{f}$ se puede expresar como

$$\mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}), \ldots, f_m(\mathbf{x})) \qquad \text{para } \mathbf{x} \in D,$$

donde $f_i : D \subset \mathbb{R}^n \to \mathbb{R}$, para $i = 1, \ldots, m$, se llaman *funciones componentes* de $\mathbf{f}$.

Definición 1.30. Se dice que una función vectorial $T : \mathbb{R}^n \to \mathbb{R}^m$ es una *transformación lineal* si

$$T(\mathbf{x}_1 + \mathbf{x}_2) = T(\mathbf{x}_1) + T(\mathbf{x}_2), \qquad T(\alpha\mathbf{x}) = \alpha T(\mathbf{x}),$$

para todo $\mathbf{x}, \mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^n$ y todo escalar α.

El concepto de continuidad de una función de varias variables se define como una generalización de la continuidad de una función de una variable.

Definición 1.31. Sean $\mathbf{f} : D \subset \mathbb{R}^n \to \mathbb{R}$ y $\mathbf{x}_0 \in D$. Se dice que f es continua en $\mathbf{x}_0$ si para $\varepsilon > 0$ existe $\delta = \delta(\varepsilon) > 0$ tal que $|f(\mathbf{x}) - f(\mathbf{x}_0)| < \varepsilon$, siempre que $0 < \|\mathbf{x} - \mathbf{x}_0\| < \delta$ con $\mathbf{x} \in D$.

Definición 1.32. Una función vectorial $\mathbf{f} : D \subset \mathbb{R}^n \to \mathbb{R}^m$ es continua en $\mathbf{x}_0 \in D$, si cada una de sus funciones componentes es continua en $\mathbf{x}_0$.

Definición 1.33. Sean $\mathbf{f} : D \subset \mathbb{R}^n \to \mathbb{R}^m$ y $\mathbf{x}_0 \in D$. Se dice que $\mathbf{f}$ es diferenciable en $\mathbf{x}_0$ si existe una transformación lineal $T : \mathbb{R}^n \to \mathbb{R}^m$ tal que

$$\lim_{\mathbf{h}\to\mathbf{0}} \frac{\|\mathbf{f}(\mathbf{x}_0+\mathbf{h}) - \mathbf{f}(\mathbf{x}_0) - T(\mathbf{h})\|}{\|\mathbf{h}\|} = \mathbf{0}.$$

La generalización del concepto de derivaba de una función vectorial dada arriba, no es muy operativa. Seguidamente damos algunos conceptos que permiten calcular la derivada de funciones vectoriales.

Definición 1.34. Sean $f : D \subset \mathbb{R}^n \to \mathbb{R}$ y $\mathbf{x} \in D$. La *derivada parcial* del campo escalar f con respecto a la j-ésima variable x_j de $\mathbf{x}$ se define por

$$\frac{\partial f}{\partial x_j}(\mathbf{x}) = \lim_{t\to 0} \frac{f(\mathbf{x}+\mathbf{e}_j) - f(\mathbf{x})}{t},$$

en caso de que el límite exista.

Las derivadas parciales de una función vectorial $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$, se definen de manera análoga. La derivada parcial de $\mathbf{f}$ con respecto a la j-ésima variable x_j de $\mathbf{x}$ existe si y sólo si existen las derivadas parciales de sus funciones componentes $f_1, \ldots, f_m$ y, en este caso,

$$\frac{\partial \mathbf{f}}{\partial x_j}(\mathbf{x}) = \left(\frac{\partial f_1}{\partial x_j}(\mathbf{x}), \ldots, \frac{\partial f_m}{\partial x_j}(\mathbf{x}) \right).$$

Esto último nos permite definir la *matiz jacobiana* o *jacobiano* de un campo vectorial.

Definición 1.35. Sean $\mathbf{f} : D \subset \mathbb{R}^n \to \mathbb{R}^m$ y $\mathbf{x} \in D$. Sean $f_1, \ldots, f_m$ las funciones componentes de $\mathbf{f}$. El jacobiano del campo vectorial $\mathbf{f}$ en $\mathbf{x}$ está dado por

$$\mathbf{f}'(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(\mathbf{x}) & \frac{\partial f_1}{\partial x_2}(\mathbf{x}) & \cdots & \frac{\partial f_1}{\partial x_n}(\mathbf{x}) \\ \frac{\partial f_2}{\partial x_1}(\mathbf{x}) & \frac{\partial f_2}{\partial x_2}(\mathbf{x}) & \cdots & \frac{\partial f_2}{\partial x_n}(\mathbf{x}) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1}(\mathbf{x}) & \frac{\partial f_m}{\partial x_2}(\mathbf{x}) & \cdots & \frac{\partial f_m}{\partial x_n}(\mathbf{x}) \end{bmatrix}$$

en caso de que las derivadas parciales existan.

En el caso de un campo escalar el jacobiano tiene una sola fila y se le conoce con el nombre de gradiente.

Definición 1.36. Sean $f : D \subset \mathbb{R}^n \to \mathbb{R}$ un campo escalar y $\mathbf{x} \in D$. El *gradiente* de f en $\mathbf{x}$ se define por

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}(\mathbf{x}), \frac{\partial f}{\partial x_2}(\mathbf{x}), \ldots, \frac{\partial f}{\partial x_n}(\mathbf{x}) \right)$$

en caso de que las derivadas parciales existan.

Como el gradiente es un campo vectorial, entonces su jacobiano es una matriz cuadrada denominada Hessiana.

Definición 1.37. Sean $f : D \subset \mathbb{R}^n \to \mathbb{R}$ un campo escalar y $\mathbf{x} \in D$. La *Hessiana* de f en $\mathbf{x}$ se define por

$$\nabla^2 f(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2}(\mathbf{x}) & \frac{\partial^2 f}{\partial x_1 \partial x_2}(\mathbf{x}) & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_1}(\mathbf{x}) \\ \frac{\partial^2 f}{\partial x_2 \partial x_1}(\mathbf{x}) & \frac{\partial^2 f}{\partial x_2^2}(\mathbf{x}) & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n}(\mathbf{x}) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1}(\mathbf{x}) & \frac{\partial^2 f}{\partial x_n \partial x_2}(\mathbf{x}) & \cdots & \frac{\partial^2 f}{\partial x_n^2}(\mathbf{x}) \end{bmatrix}$$

en caso de que las derivadas parciales existan.

A continuación damos el Teorema de Taylor para funciones de dos variables.

Teorema 1.19 (Teorema de Taylor para funciones de dos variables)**.** *Sean f(t, y) y sus derivadas parciales hasta orden* $(n+1)$ *continuas en el rectángulo abierto*

$$R = \{(t,y) \in \mathbb{R}^2 : a < t < b, c < y < d\},$$

y sea $(t_0, y_0) \in R$*. Entonces,*

$$f(t,y) = f(t_0,y_0) + \left[(t-t_0)\frac{\partial f}{\partial t}(t_0,y_0) + (y-y_0)\frac{\partial f}{\partial y}(t_0,y_0) \right] + \cdots$$
$$+ \left[\frac{1}{n!} \sum_{i=0}^{n} \binom{n}{i} (t-t_0)^{n-i} (y-y_0)^i \frac{\partial^n f}{\partial t^{n-i} \partial y^i}(t_0,y_0) \right] + E_n(t,y),$$

donde $E_n(t,y)$ *es el término de error que envuelve las derivadas parciales de* $f(t,y)$*.*

1.3 Calculando con Números

Todo algoritmo numérico, no importa cuán complicado sea, siempre se reduce a realizar una secuencia de operaciones aritméticas simples. Prácticamente todos los cálculos numéricos se realizan en algún tipo de computadora digital; por ello es necesario describir los detalles de la aritmética en computadora. Primero, revisemos cómo se representan los números en una computadora.

1.3.1 Representación Numérica

La representación o sistema decimal que usamos a diario se nos hace muy familiar y trabajamos con éste fácilmente. La representación decimal es un sistema posicional que usa la base 10. Esto es, el valor de un número depende de sus dígitos y de la posición de los mismos con respecto a un punto fijo, el punto decimal. El valor de cada número se multiplica por una potencia de 10, creciendo las potencias positivas a la izquierda del punto decimal y decreciendo las potencias negativas a la derecha. Por ejemplo, el número 512.31 es representado como un número decimal por:

$$(512.31)_{10} = 5\times 10^2 + 1\times 10^1 + 2\times 10^0 + 3\times 10^{-1} + 1\times 10^{-2}$$

El sistema decimal no es eficiente para ser usado internamente por una computadora. En lugar de ello se utiliza, generalmente, el sistema de base 2 o binario. Un sistema binario es semejante al sistema decimal, excepto que la base es 2, los dígitos binarios (bits) permitidos son 0 y 1, y cada bit es multiplicado por una potencia de 2, dependiendo de su posición relativa al punto binario. La conversión de binario a decimal y viceversa es fácil:

$$\begin{aligned}(1001.001)_2 &= 1\times 2^3 + 0\times 2^2 + 0\times 2^1 + 0\times 2^{-1} + 0\times 2^{-2} + 1\times 2^{-3}\\ &= 8 + 1 + \frac{1}{8} = (9.125)_{10}\end{aligned}$$

Una dificultad con esta representación, a menudo llamada de *punto-fijo,* es su ineficiencia para representar números muy grandes o muy pequeños; por ejemplo, el número 0.00000015 tiene nueve dígitos, pero solamente dos de ellos ofrecen información, el resto son todos ceros que son necesarios para escalar el número. Tal inconveniente puede remediarse con la *notación científica*, cuyo objetivo principal es representar un número en dos partes. La primera es un número decimal con representación de punto-fijo y, la segunda parte, es un factor de escala que permite representar un rango de números. Específicamente, un número real x se representa en notación científica como

$$x = \pm 0.d_1 d_2 \cdots d_n d_{n+1} \cdots \times \beta^e$$

donde $d_i \in \{0,1,\ldots,\beta-1\}$, para $i = 1,2,\ldots$, e es un entero positivo y β es la base (comúnmente se usa $\beta = 2$ o $\beta = 10$). Los dígitos d_i empezando con el primer dígito distinto de cero se denominan *dígitos significativos*. Si d_1 es distinto de cero, se dice x está normalizado. Por ejemplo, el número 0.00000015 puede ser representado como 0.15×10^{-6} o 0.015×10^{-5} (existen infinitas representaciones, se deja al lector encontrar otra). La primera representación es la notación científica normalizada de 0.00000015; en cambio la segunda no lo es, además, se aprecia que 0.00000015 tiene dos dígitos significativos. A partir de ahora supondremos, a menos que se indique lo contrario, que los números estarán representados en notación científica normalizada con lo cual garantizamos la unicidad en la representación. Una versión binaria de esta notación es usada comúnmente en computadoras y se describirá en detalle en la siguiente sección.

1.3.2 Representación de Punto-Flotante

La representación de *punto-flotante* de un número tiene dos partes: una mantisa m, que representa los dígitos significativos y un exponente e, que da un factor de escala. Ambos m y e están codificados en binario con un número fijo de bits, y el factor de escala es alguna potencia de 2. Dentro de estas especificaciones existen muchas posibilidades de escogencia. En 1985 el Instituto de Ingeniería Eléctrica y Electrónica (conocido por sus siglas IEEE, en inglés Institute of Electrical and Electronics Engineers) produjo un conjunto de especificaciones y requerimientos para la

aritmética de punto-flotante que son conocidos como *estándar IEEE*. Actualmente, el esquema más usado para punto-flotante es el *formato doble IEEE* o de *doble precisión*, que se muestra en la Figura 1.2.

Figura 1.2. Formato doble IEEE

El formato doble IEEE utiliza 64 bits (u 8 bytes); un bit es usado para el signo del número s (1 cuando el número es negativo y 0 cuando es positivo), y 52 bits son dedicados a la mantisa m. Los restantes 11 bits representan el exponente e. Un número x representado por este arreglo se escribe como:

$$x = (-1)^s \times 1.(m)_{10} \times 2^E$$

El campo del exponente e es una representación sesgada de su valor actual E (entero en base decimal); además, e y E están relacionados por

$$(e)_{10} = E + 1023.$$

La mantisa m es una fracción con el punto binario inmediatamente a su izquierda, por ejemplo,

$$m = 1101000000000000000000000000000000000000000000000000$$

tiene un valor decimal

$$(m)_{10} = 2^{-1} + 2^{-2} + 2^{-4} = 0.8125$$

Ahora, e y s son considerados enteros. Pasemos a ver un ejemplo.

Ejemplo 1.10. Hallemos la representación del número -11.6 en el formato doble IEEE. Se comienza expresando a -11.6 punto-flotante

$$-11.6 = (-1)^1 \times (1.4375) \times 2^3$$

De allí se obtiene que $s = 1$. Ahora, para encontrar la representación binaria de la parte fraccionaria, .4375, se procede de la siguiente manera. Debemos hallar los bits $a_{-1}, a_{-2}, a_{-3}, \ldots$ tales que

$$.4375 = a_{-1}2^{-1} + a_{-2}2^{-2} + a_{-3}2^{-3} + \cdots$$

Para hallar a_{-1}, multiplicamos por 2 la ecuación anterior en ambos lados. Así obtenemos:

$$0.8750 = a_{-1} + a_{-2}2^{-1} + a_{-2}2^{-2} + \cdots$$

luego, $a_{-1} = 0$. Procediendo de la misma forma hallamos los siguientes bits. En su totalidad, el procedimiento es el siguiente:

$$\begin{array}{rr} & .4375 \\ & \underline{2} \\ \boxed{0} & .8750 \\ & \underline{2} \\ \boxed{1} & .7500 \\ & \underline{2} \\ \boxed{1} & .5000 \\ & \underline{2} \\ \boxed{1} & .0000 \end{array}$$

Por lo tanto, la mantisa m es la cadena de bits

$$m = 01110000000000000000000000000000000000000000000000000$$

El valor de E es 3; por lo tanto,

$$(e)_{10} = 1026 = 2^{10} + 2.$$

Así,

$$e = 10000000010$$

Colocando juntos s, m y e, obtenemos que el número decimal -11.6 está representado en el formato doble IEEE por la cadena de bits:

$$1100000000100111000000000000000000000000000000000000000000000000$$

Note que el formato doble IEEE tiene una longitud finita de bits (o cualquier otro formato de punto-flotante). Por ello, este formato puede representar un conjunto finito de números decimales. El rango de números que pueden ser representados varía de $2^{-1023} \approx 10^{-308}$ a $2^{1024} \approx 10^{309}$.

Existen números reales que no pueden ser representados de forma exacta en punto-flotante, es decir, se incurre en un error al representar el número real como un número punto-flotante. Dos medidas muy utilizadas del error cometido al aproximar un número son el *error absoluto* y el *error relativo*.

Definición 1.38. Sea x un número distinto de cero y sea $\tilde{x}$ una aproximación de x. Entonces, el error absoluto y el error relativo se definen como $|x - \tilde{x}|$ y $\dfrac{|x-\tilde{x}|}{|x|}$, respectivamente.

En el siguiente ejemplo mostramos el error cometido en la representación en punto-flotante del número 0.1.

Ejemplo 1.11. El número

$$x = 0.1 = (-1)^0 \times (1.6) \times 2^{-4}$$

puede ser representado en el formato doble IEEE mediante las siguientes cadenas de bits

$$c_1 = 0111111101110011001100110011001100110011001100110011001100110011$$

o

$$c_2 = 0111111101110011001100110011001100110011001100110011001100110010$$

Para la cadena de bits c_1, se observa que

$$\begin{aligned} s &= 0, \\ e &= 1111111011, \\ m_1 &= 10011001100110011001100110011001100110011001100110011001; \end{aligned}$$

por lo tanto, al realizar la conversión a decimal se obtiene:

$$(e)_{10} = 1019 \quad \Rightarrow \quad E = -4,$$
$$(m_1)_{10} = \sum_{k=1}^{13} (2^{-4k+3} + 2^{-4k}),$$

así,

$$\begin{aligned} 0.1 \approx x_1 &= (-1)^0 \times (1.(m_1)_{10}) \times 2^{-4} \\ &= 2^{-4} + \sum_{k=1}^{13} (2^{-4k-1} + 2^{-4k-4}) \\ &= 2^{-4} + (2^{-1} + 2^{-4}) \sum_{k=1}^{13} 2^{-4k} \\ &= 2^{-4} + \left(\frac{2^3+1}{2^4}\right)\left[\frac{2^{-52}-1}{1-2^4}\right] \\ &\approx 0.09999999999999999167332731531133 \end{aligned}$$

Para la cadena de bits c_2, se observa que

$$\begin{aligned} s &= 0, \\ e &= 1111111011, \\ m_2 &= 10011001100110011001100110011001100110011001100110011010; \end{aligned}$$

por lo tanto, al realizar la conversión a decimal se obtiene:

$$(e)_{10} = 1019 \quad \Rightarrow \quad E = -4,$$
$$(m_2)_{10} = (m_1)_{10} - 2^{-52} + 2^{-51},$$

así,

$$\begin{aligned} 0.1 \approx x_2 &= (-1)^0 \times (1.(m_2)_{10}) \times 2^{-4} = 2^{-4} + \left(\frac{2^3+1}{2^4}\right)\left[\frac{2^{-52}-1}{1-2^4}\right] - 2^{-56} + 2^{-55} \\ &\approx 0.10000000000000000555111512312579 \end{aligned}$$

De esta forma, si se toma a x_1 como la representación de punto-flotante del número 0.1, se comete un error absoluto

$$|0.1 - x_1| \approx 8.326673 \times 10^{-18},$$

y un error relativo

$$\frac{|0.1 - x_1|}{0.1} \approx 8.326673 \times 10^{-17}.$$

Ahora, si se toma a x_2 como la representación de punto-flotante del número 0.1, se comete un error absoluto

$$|0.1 - x_2| \approx 5.551115 \times 10^{-18},$$

y un error relativo

$$\frac{|0.1 - x_2|}{0.1} \approx 5.551115 \times 10^{-17}.$$

Los números en punto-flotante x_1 y x_2 del Ejemplo 1.11 se obtuvieron mediante el proceso de **truncamiento**, que consiste en descartar los bits excedentes de la representación binaria de la mantisa m. Observe que x_1 se encuentra a la izquierda de 0.1 en la recta real; en cambio, x_2 queda a la derecha. El número x_2 también se obtuvo por truncamiento, pero a diferencia de x_1, se obtuvo por el proceso de redondeo por **exceso**. Esto significa que se elimina el exceso de bits de la mantisa m, pero en esta ocasión se aumenta en una unidad el último bit. Ahora bien, para que no exista ambigüedad en la representación en punto-flotante de un número x, se escoge el número punto-flotante más cercano a x, el cual denotaremos como $fl(x)$. En el caso del Ejemplo 1.11, la representación en punto flotante de 0.1 es $fl(0.1) = x_2$.

Construyamos una cota del error relativo de un número $x \in [2^{-1023}, 2^{1024}]$ cuya representación en el formato doble IEEE es $fl(x)$. Se tiene que todo número real $x \neq 0$ se puede representar como

$$x = \pm 0.d_1 d_2 \cdots d_{52} d_{53} \cdots \times 2^{\alpha},$$

donde $2^{-1023} \leq \alpha \leq 2^{1024}$. Es claro que se satisface:

$$2^{\alpha-1} \leq |x| \leq 2^{\alpha}, \tag{1.2}$$

además, $|\lfloor x \rfloor - x| \leq 1$, donde $\lfloor x \rfloor$ es el entero más cercano a x tal que $\lfloor x \rfloor \leq x$. Luego,

$$\lfloor x 2^{53-\alpha} \rfloor = \pm d_1 d_2 \cdots d_{53},$$

y de esto se obtiene

$$fl(x) = \lfloor x 2^{53-\alpha} \rfloor 2^{\alpha-53}.$$

De esta forma, podemos escribir:

$$\begin{aligned} |fl(x) - x| &= \left| \lfloor x 2^{52-\alpha} \rfloor 2^{\alpha-53} - x 2^{\alpha-53} 2^{53-\alpha} \right| \\ &= \left| \lfloor x 2^{53-\alpha} \rfloor - x 2^{53-\alpha} \right| 2^{\alpha-53} \\ &\leq 2^{\alpha-53} \end{aligned} \tag{1.3}$$

que es una cota del error absoluto. Por (1.2) se tiene $1/|x| \leq 2^{1-\alpha}$. Entonces, por (1.3) obtenemos

$$\left| \frac{x - fl(x)}{x} \right| \leq 2^{\alpha-53} 2^{1-\alpha} = 2^{-52},$$

que es una cota del error relativo. Si tomamos $\mu = (fl(x) - x)/x$, la desigualdad anterior adquiere la siguiente forma:

$$fl(x) = x(1+\mu), \qquad |\mu| \leq \varepsilon = 2^{-52}, \tag{1.4}$$

para todo número real x.

Definición 1.39. El número ε que aparece en la ecuación (1.4) se denomina *error de redondeo unitario* o, equivalentemente, *epsilon* o *precisión de la máquina*, y se denota como `eps`. El epsilón de la máquina está directamente relacionado con el número de bits asignados a la mantisa. En el formato doble IEEE el valor de `eps` es $2^{-52} \approx 2.2 \times 10^{-16}$.

El épsilon de la máquina `eps` también puede definirse como el número positivo más pequeño tal que $(1 + \texttt{eps}) > 1$. Con esta definición diseñaremos un pequeño programa que nos permitirá calcular el `eps` de la computadora que estemos utilizando.

Ejemplo 1.12. Para encontrar la medida del `eps`, le sumaremos el valor 1 a una sucesión decreciente de números positivos, hasta que los resultados no muestren cambios. De esta forma, encontraremos el número positivo `eps` tal que $(1+\texttt{eps}) > 1$. Seguidamente se muestra un simple programa que calcula el epsilon de la máquina.

```
eps = 1.
while 1. + eps > 1.:
    eps /= 2.
eps *= 2.
print(eps)
```

Cuando este programa corre en nuestra computadora (en Python), el valor que imprime es

$$\texttt{eps} = 2.2204 \times 10^{-16},$$

exactamente lo que uno espera del formato doble IEEE.

■

En algunas situaciones, cuando realizamos cálculos en una computadora, se puede generar un número de la forma $\pm p \times 2^E$, donde E queda fuera del rango permitido por la computadora. Si E es demasiado grande, se dice que tuvo lugar un **overflow** o **desbordamiento por exceso**. Si E es demasiado pequeño, se dice que ocurrió un **underflow** o **desbordamiento por defecto**. En el formato doble IEEE esto significa que $E > 1024$ o que $E < -1023$, respectivamente. En general, que ocurra un overflow se considera un error fatal, que en muchas computadoras ocasionaría la interrupción de manera automática de la ejecución del programa. Cuando ocurre un underflow, muchas computadoras simplemente asignan a la variable el valor 0 y el cálculo continúa.

Bajo las especificaciones del formato estándar IEEE, se establecen las siguientes leyes de la aritmética punto-flotante [22, 32].

Teorema 1.20 (Leyes de la aritmética punto flotante)**.** *Sean x e y dos números punto-flotante y sean $fl(x+y)$, $fl(x-y)$, $fl(xy)$, y $fl(x/y)$ la suma, resta, producto y división, respectivamente. Entonces:*

i) $fl(x \pm y) = (x \pm y)(1+\mu)$, *donde* $|\mu| < \texttt{eps}$.

ii) $fl(xy) = (xy)(1+\mu)$, *donde* $|\mu| < \texttt{eps}$.

iii) Si $y \neq 0$, *entonces* $fl(x/y) = (x/y)(1+\mu)$, *donde* $|\mu| < \texttt{eps}$.

iv) Si x e y son números reales que no poseen representación exacta punto-flotante, entonces se cumple la siguiente ley:

$$fl(x+y) = (x(1+\mu_1)+y(1+\mu_2))(1+\mu_3), \quad \text{donde } |\mu_1| < \texttt{eps},\ |\mu_2| < \texttt{eps},\ |\mu_3| < \texttt{eps}.$$

Esta última propiedad se cumple de forma similar para la resta, el producto y la división.

El Ejemplo 1.11 es una evidencia que los errores de redondeo son inevitables y difíciles de controlar. Existe otro tipo de errores que sí se pueden de alguna forma controlar. Un error frecuente en análisis numérico es la pérdida de dígitos significativos que aparece como consecuencia de descuidos durante el proceso de programación. El caso más emblemático es la sustracción de cantidades casi iguales. Como ejemplo veamos la magnitud del error relativo que puede producirse al restar dos números muy cercanos.

Ejemplo 1.13. Sean $x = 0.37214786932175$ e $y = 0.37214720021582$. Se tiene que

$$x - y = 0.00000066910593$$

Si los cálculos se realizaran en una computadora decimal con mantisa de 8 dígitos, veríamos que

$$fl(x) = 0.37214786, \qquad fl(y) = 0.37214720, \qquad \Rightarrow \qquad fl(x) - fl(y) = 0.00000066$$

Por lo que, el error relativo es:

$$\left| \frac{x - y - (fl(x) - fl(y))}{x - y} \right| = \left| \frac{0.00000066910593 - 0.00000066}{0.00000066910593} \right| \approx 0,013609 \Rightarrow 1.4\%$$

En otras palabras, el error relativo es grande. Ahora, el número $fl(x) - fl(y)$ se representa en la computadora como $0.66000000 \times 10^{-6}$, aunque los ceros de la mantisa sólo sirven para denotar el lugar decimal. Note que se perdieron 6 dígitos significativos.

■

La situación descrita en el ejemplo anterior también puede aparecer cuando se evalúan expresiones algebraicas. Veamos el siguiente ejemplo.

Ejemplo 1.14. Supongamos que deseamos calcular

$$y = \sqrt{1 + x^2} - 1$$

cuando x es un número pequeño. Observamos que si $x \approx 0$, entonces $y \approx 0$. Este hecho implica la cancelación por sustracción y pérdida de dígitos significativos para valores de x muy cercanos a 0. ¿Cómo podemos evitar este problema?

Si consideramos una forma equivalente de calcular y:

$$y = \left(\sqrt{1 + x^2} - 1 \right) \left(\frac{\sqrt{1 + x^2} + 1}{\sqrt{1 + x^2} + 1} \right) = \frac{x^2}{\sqrt{1 + x^2} + 1},$$

entonces la dificultad se evita.

■

El siguiente teorema establece cotas del número de bits significativos que se pierden al efectuar la sustracción $x - y$ cuando x es cercano a y, en términos de la cantidad $|1 - y/x|$.

Teorema 1.21. *Si x e y son números punto-flotantes, positivos y tales que $x > y$ con*

$$2^{-q} \leq 1 - \frac{y}{x} \leq 2^{-p},$$

entonces en la resta $x - y$ se pierden a lo sumo q y al menos p bits significativos.

1.3.3 Eficiencia, Estabilidad y Condicionamiento

En el cálculo numérico, además de medir el error en aproximaciones, también es necesario medir la *eficiencia* y la *estabilidad* de un algoritmo.

Definición 1.40. Un *algoritmo* es un conjunto finito y ordenado de operaciones, lógicas y aritméticas, las cuales al aplicarse a un problema computacional asociado a un conjunto de datos, llamado *entrada*, produce una solución al problema. Esta solución comprende un conjunto de datos que se conocen como *salida*.

El trabajo computacional de los algoritmos o, equivalentemente, su eficiencia se mide por el número de operaciones punto-flotante que son necesarias en su ejecución.

Definición 1.41. Una operación punto-flotante es toda operación de suma, resta, multiplicación o división.

Definición 1.42. Se dice que un algoritmo es de orden $\mathbf{O}(n^k)$, si el término dominante en el número de operaciones punto-flotante requeridas en su ejecución es un múltiplo de n^k.

Ejemplo 1.15. Supongamos que deseamos calcular la siguiente suma

$$S = \sum_{i=1}^{N} (-1)^{i+1} \frac{1}{1+i^2}.$$

Desarrollando la suma se tiene

$$\begin{aligned} S &= (-1)^2 \frac{1}{1+1^2} + (-1)^3 \frac{1}{1+2^2} + \cdots + (-1)^{N+1} \frac{1}{1+N^2} \\ &= (-1)\cdot(-1)\frac{1}{1+1\cdot 1} + (-1)^2\cdot(-1)\frac{1}{1+2\cdot 2} + \cdots + (-1)^N\cdot(-1)\frac{1}{1+N\cdot N}. \end{aligned}$$

Luego, el número de operaciones punto-flotante necesarias para calcular suma S es: $2N-1$ sumas, $3N$ multiplicaciones y N divisiones; por lo tanto, para calcular la suma S hacen falta $6N-1$ operaciones punto-flotante. Así, un algoritmo para calcular la suma S es de orden $\mathbf{O}(N)$.

Por otro lado, la precisión de la solución de un problema obtenida por un algoritmo depende de dos características importantes ([6]):

- La *estabilidad* del algoritmo.
- La *condición* del problema considerado, es decir, que tan sensible es el problema a pequeñas perturbaciones en el planteamiento del mismo.

Informalmente, diremos que un algoritmo es inestable si pequeños errores cometidos en algún paso del algoritmo, se magnifican en los siguientes pasos y destruyen la precisión de la solución obtenida. Una definición formal de estabilidad es la siguiente.

Definición 1.43. Un algoritmo es estable si produce la solución exacta de un problema cuyos datos de entrada son cercanos a los datos originales.

Ejemplo 1.16. Un ejemplo simple de estabilidad son las operaciones aritméticas de punto-flotante. Veamos que la suma de dos números punto-flotante es estable. Sean x e y dos números punto-flotante. Por el Teorema 1.20 se tiene:

$$fl(x+y) = (x+y)(1+\mu),$$

donde $|\mu| < \texttt{eps}$. De esta forma, podemos escribir

$$fl(x+y)-(x+y) = (x+y)(1+\mu)-(x+y) = \mu(x+y).$$

Por tanto, como $|\mu| < \texttt{eps}$, la suma de x e y obtenida por $fl(x+y)$ es cercana a la suma $x+y$ de los datos originales x e y. En consecuencia, la operación de sumar dos números punto-flotante es estable. Dde una manera similar se muestra la estabilidad de la multiplicación y la división.

Ejemplo 1.17. Un ejemplo de un algoritmo numérico inestable es el proceso de cálculo de la factorización LU de una matriz cuadrada, el cual lo estudiaremos con mayor detalle en el Capítulo 2. Supongamos que se desea encontrar la factorización LU de la matriz

$$A = \begin{bmatrix} \delta & -1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ l_{21} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} \\ 0 & u_{22} \end{bmatrix}, \qquad 0 < \delta \ll 1,$$

es decir, se debe hallar l_{12}, u_{11}, u_{12} y u_{22}. Es claro que $u_{11} = \delta$, $u_{12} = -1$, $l_{21} = \delta^{-1}$ y $u_{22} = 1 - l_{21}u_{12} = 1 + \delta^{-1}$. En aritmética punto-flotante, si δ es suficientemente pequeño ($\delta < \texttt{eps}$), entonces $fl(u_{22}) = \delta^{-1}$. Aun suponiendo que el resto de las cuentas se obtienen de forma exacta, al multiplicar las dos matrices triangulares punto flotante, se obtiene la matriz

$$\begin{bmatrix} \delta & -1 \\ 1 & 0 \end{bmatrix}$$

cuyo elemento $(2,2)$ difiere notablemente del elemento $(2,2)$ de la matriz A. Es decir, el proceso de factorización realizado de esa manera reproduce una matriz que dista mucho de la original, y por ende es un proceso inestable. Comprobemos esto numéricamente. Inicialmente tomemos $\delta = 10^{-15}$ y calculemos el producto LU.

```
>>> delta = 1.0e-15
>>> L = np.array([[1,0],[1/delta,1]])
>>> U = np.array([[delta,-1],[0,1+(1/delta)]])
>>> L.dot(U)
array([[  1.e-15,  -1.e+00],
       [  1.e+00,   1.e+00]])
```

En este caso, el producto de las matrices L y U arroja la matriz A original. Ahora, tomemos $\delta = 10^{-16}$ y calculemos nuevamente el producto LU.

```
>>> delta = 1.0e-16
>>> L = np.array([[1,0],[1/delta,1]])
>>> U = np.array([[delta,-1],[0,1+(1/delta)]])
>>> L.dot(U)
array([[  1.e-16,  -1.e+00],
       [  1.e+00,   0.e+00]])
```

La matriz obtenida difiere en el elemento $(2,2)$ de la matriz A original. Esta inestabilidad se resolverá en el Capítulo 2 introduciendo diferentes estrategias de pivoteo para factorizar matrices.

Hemos visto que la precisión de la solución de un problema depende de la estabilidad de los algoritmos, pero exigir que el algoritmo sea estable no es suficiente para garantizar la buena calidad de las soluciones obtenidas. Como indicamos arriba, el condicionamiento también contribuye en la precisión de las soluciones esperadas. La condición de un problema es una propiedad intrínseca del problema y se refiere a como cambia la solución cuando los datos contienen algún tipo de ruido o perturbación, independientemente del algoritmo usado. El estudio teórico efectuado por analistas numéricos para investigar el efecto que estas perturbaciones producen se conoce como *análisis de perturbación*. Este tipo de análisis permite discernir si un problema dado es "bueno" o "malo", es decir, si pequeñas perturbaciones en los datos producen pequeños o grandes cambios en la solución del problema ([6]).

Definición 1.44. Un problema (con respecto a un conjunto de datos) se llama *mal-condicionado* si un error relativo pequeño en los datos produce un error relativo grande en la solución, independientemente del algoritmo usado. En caso contrario se llama *bien-condicionado*.

Una interpretación matemática de esta definición es la siguiente. Supongamos que un problema P se resuelve para un conjunto de datos d y denotamos con $P(d)$ el valor de la solución obtenida. Sea δ_d una perturbación de los datos d. Diremos que P está mal-condicionado si el error relativo de la solución es mucho mayor que el error relativo de los datos, en otras palabras, si se cumple la desigualdad

$$\kappa_P = \frac{\dfrac{|P(d+\delta_d)-P(d)|}{|P(d)|}}{\dfrac{|\delta_d|}{|d|}} \gg 1.$$

El escalar $\kappa_P > 0$ se denomina *número de condición del problema*. Si $\kappa_P \ll 1$, se dice que el problema está bien-condicionado. Vale destacar que, en general, un problema puede estar mal-condicionado para un conjunto de datos y bien-condicionado para otro conjunto de datos. Veamos un ejemplo.

Ejemplo 1.18. Supongamos que se desea calcular un valor de la función

$$f(x) = \sqrt{x^2+1} + e^x,$$

para $x \in \mathbb{R}$. Calculemos el número de condición de este problema. Sea $\delta \in (0,1)$ lo suficientemente pequeño. Se tiene que el número de condición del problema está dado por:

$$\kappa_P = \frac{\dfrac{|f(x+\delta)-f(x)|}{|f(x)|}}{\dfrac{|\delta|}{|x|}} = \left|\frac{f(x+\delta)-f(x)}{\delta}\right| \cdot \frac{|x|}{|f(x)|} \simeq \left|x\,\frac{f'(x)}{f(x)}\right|,$$

para todo $x \in \mathbb{R}$. Calculemos el número de condición del problema para ciertos valores de x. Se tiene que

$$f'(x) = \frac{x}{\sqrt{x^2+1}} + e^x.$$

Para $x = 100$,

$$\kappa_P = \left|100 \cdot \frac{f'(100)}{f(100)}\right| = \frac{100\,e^{100} + \frac{10000\sqrt{10001}}{10001}}{e^{100}+\sqrt{10001}} \approx 100.$$

Para $x = 10$,

$$\kappa_P = \left| 10 \cdot \frac{f'(10)}{f(10)} \right| = \frac{10e^{10} + \frac{100\sqrt{101}}{101}}{e^{10} + \sqrt{101}} \approx 10.$$

Para $x = 1$,

$$\kappa_P = \left| 1 \cdot \frac{f'(1)}{f(1)} \right| = \frac{e + \frac{\sqrt{2}}{2}}{e + \sqrt{2}} \approx 0.82889109017393.$$

Para $x = 10^{-4}$,

$$\kappa_P = \left| 10^{-4} \cdot \frac{f'(10^{-4})}{f(10^{-4})} \right| = \frac{100000001\, e^{\frac{1}{10000}} + \sqrt{100000001}}{100000001 \left(10000\, e^{\frac{1}{10000}} + \sqrt{100000001} \right)} \approx 5.0 \times 10^{-5}.$$

De esta forma, calcular un valor de la función $f(x)$ dada es un problema que está mal o bien-condicionado dependiendo del valor de x considerado. Por ejemplo, si $x = 100$ el problema está mal-condicionado; por el contrario, si $x = 10^{-5}$ el problema está bien-condicionado.

■

Condicionamiento de Sistemas Lineales

En esta sección estudiaremos el efecto de pequeñas perturbaciones en sistemas lineales de la forma $A\mathbf{x} = \mathbf{b}$, donde $A \in \mathbb{R}^{n\times n}$ y $\mathbf{b} \in \mathbb{R}^n$. Estos problemas aparecen en muchas aplicaciones. En el Capítulo 2 los estudiaremos en detalle. En un sistema lineal, los datos de entrada son una matriz cuadrada A de orden n y el vector $\mathbf{b} \in \mathbb{R}^n$, que pueden tener perturbaciones. Veremos que un número especial, conocido como el *número de condición de la matriz A*, jugará un papel fundamental en el condicionamiento de un sistema lineal.

Definición 1.45. El *número de condición de la matriz* $A \in \mathbb{R}^{n\times n}$ no singular, denotado por $\kappa_p(A)$, se define por

$$\kappa_p(A) = \|A\|_p \|A^{-1}\|_p,$$

donde $\|\cdot\|_p$ es un norma matricial inducida.

Denotemos las perturbaciones de A, $\mathbf{b}$ y de la solución $\mathbf{x}$ de la siguiente manera:

$$\begin{aligned} A &\to A + \triangle A \quad (\triangle A\text{: perturbación de la matriz } A), \\ \mathbf{b} &\to \mathbf{b} + \delta\mathbf{b} \quad (\delta\mathbf{b}\text{: perturbación del vector } \mathbf{b}), \\ \mathbf{x} &\to \mathbf{x} + \delta\mathbf{x} \quad (\delta\mathbf{x}\text{: cambio en la solución}). \end{aligned}$$

El siguiente teorema establece que el error relativo en la solución $\mathbf{x}$ está dominado por el número de condición de la matriz A, siempre que las perturbaciones relativas en A y $\mathbf{b}$ sean pequeñas. Para la demostración de este resultado general ver, por ejemplo, el libro [13, p. 123].

Teorema 1.22. *Sean $\triangle A$ y $\delta\mathbf{b}$ las perturbaciones de A y $\mathbf{b} \neq 0$, respectivamente, y sea $\delta\mathbf{x}$ el error absoluto en $\mathbf{x}$ al resolver $A\mathbf{x} = \mathbf{b}$. Sea $\|\cdot\|_p$ una norma matricial inducida. Si A es no singular y $\|\triangle A\|_p < \dfrac{1}{\|A^{-1}\|_p}$, entonces*

$$\frac{\|\delta\mathbf{x}\|_p}{\|\mathbf{x}\|_p} \leq \frac{\kappa_p(A)}{(1 - \|\triangle A\|_p \|A^{-1}\|_p)} \left(\frac{\|\triangle A\|_p}{\|A\|_p} + \frac{\|\delta\mathbf{b}\|_p}{\|\mathbf{b}\|_p} \right).$$

Del Teorema 1.22 se infiere:

- Si $\kappa_p(A)$ es grande, entonces el sistema $A\mathbf{x} = \mathbf{b}$ está mal-condicionado.
- Si $\kappa_p(A)$ es pequeño, entonces el sistema $A\mathbf{x} = \mathbf{b}$ está bien-condicionado.

El número de condición de una matriz depende de la norma utilizada, sin embargo, recordando que en espacios de dimensión finita (como $\mathbb{R}^n$ o $\mathbb{R}^{n\times n}$) todas las normas son equivalentes (ver Ejercicio 4), podemos afirmar que si una matriz es mal-condicionada en una norma, también lo es en cualquier otra norma. Algunas de las propiedades de $\kappa_p(A)$ que podemos destacar son las siguientes:

(i) $\kappa_p(A) = \kappa_p(A^{-1})$.

(ii) $\kappa_p(A) \geq 1$ para cualquier norma inducida.

(iii) $\kappa_p(cA) = \kappa_p(A)$, para cualquier escalar $c \neq 0$.

(iv) $\kappa_p(A^T) = \kappa_p(A)$.

(v) $\kappa_p(AB) \leq \kappa_p(A)\kappa_p(B)$, para cualquier norma inducida.

(vi) $\kappa_p(A^T A) = (\kappa_p(A))^2$.

El comando `cond` del módulo `linalg` de la librería `Numpy`, se utiliza para calcular el número de condición $\kappa_p(A)$, para valores de $p = 1, 2, \text{inf}, \text{'fro'}$, entre otros. Generalmente, se emplea el número de condición $\kappa_2(A)$ calculado con la norma-2 o norma Euclídea. Veamos un ejemplo.

Ejemplo 1.19. Para la matriz A,

```
>>> A = np.array([[1,4,6],[3,-1,5],[0,8,7]])
>>> A
array([[ 1,  4,  6],
       [ 3, -1,  5],
       [ 0,  8,  7]])
```

se tiene que el número de condición $\kappa_p(A)$ se calcula como:

```
>>> from numpy import linalg as LA
>>> LA.cond(A)          # Número de condición con p=2
69.82769061243755
>>> LA.cond(A, 1)       # Número de condición con p=1
127.38461538461519
>>> LA.cond(A, 'fro')   # Número de condición con la norma de Frobenius
75.18627754690965
>>> LA.cond(A, np.inf)  # Número de condición con la norma infinito
107.30769230769216
```

Es claro que el número de condición $\kappa_p(A)$ está definido para matrices no singulares. Se puede establecer que $\kappa_p(A)$ es el inverso de la distancia a la singularidad, es decir, si $\kappa_p(A)$ es grande entonces A es muy cercana a una matriz singular ([6]).

En el siguiente teorema se establece la versión simplificada del Teorema 1.22 para el caso en que la matriz no sufre perturbaciones, esto es, cuando $\triangle A = 0$. Este es un caso de interés, ya que en muchas aplicaciones la matriz A representa el modelo matemático fijo y el ruido o perturbación aparece en el vector b de mediciones experimentales.

Teorema 1.23. *Sean $\delta\mathbf{b}$ y $\delta\mathbf{x}$, respectivamente, la perturbación de $\mathbf{b}$ y $\mathbf{x}$ al resolver $A\mathbf{x} = \mathbf{b}$. Sea $\|\cdot\|_p$ una norma matricial inducida. Si A es no singular y $\mathbf{b} \neq 0$, entonces*

$$\frac{\|\delta\mathbf{x}\|_p}{\|\mathbf{x}\|_p} \leq \kappa_p(A)\, \frac{\|\delta\mathbf{b}\|_p}{\|\mathbf{b}\|_p}.$$

El Teorema 1.23 dice que el error relativo en la solución puede llegar a ser tan grande como la $\kappa_p(A)$ multiplicada por el error relativo en el vector $\mathbf{b}$. Por tanto, si $\kappa_p(A)$ no es muy grande, entonces una pequeña perturbación en $\mathbf{b}$ tendrá poco efecto en la solución obtenida. Mientras que si $\kappa_p(A)$ es grande, entonces aún una pequeña perturbación en $\mathbf{b}$ puede ocasionar un cambio muy grande en la solución.

Ejemplo 1.20. Consideremos el sistema lineal $A\mathbf{x} = \mathbf{b}$, donde

$$A = \begin{bmatrix} 2 & 1 \\ 2 & 1.0001 \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

La solución del sistema es $\mathbf{x} = (5000.5, -10000)$. El efecto del mal condicionamiento puede verificarse cambiando el elemento a_{22} de la matriz A; por ejemplo, si $a_{22} = 1.0002$, entonces la solución del sistema de ecuaciones es $\mathbf{x} = (2500.5, -5000)$. Observe que un cambio del 0.01% en el coeficiente a_{22} produce un cambio del 100% en la solución. Por otro lado, si variamos un poco el vector del lado derecho, digamos $\mathbf{b} = (0.999, 0)$, entonces la solución del sistema es $\mathbf{x} = (4995.5, -9990)$. Estos hechos nos permiten asegurar que la matriz A está mal condicionada, cuyo número de condición es $\kappa_2(A) = 50001$.

Ejemplo 1.21. Consideremos un sistema lineal $A\mathbf{x} = \mathbf{b}$, donde

$$A = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 0.0003 & 0.0002 \\ 1 & 0.0002 & 0.0003 \end{bmatrix} \quad \text{y} \quad \mathbf{b} = \begin{bmatrix} 4 \\ 1.0005 \\ 1.0005 \end{bmatrix}.$$

La solución exacta del sistema lineal es $\mathbf{x} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$. Si cambiamos el vector $\mathbf{b}$ por

$$\widehat{\mathbf{b}} = \begin{bmatrix} 4 \\ 1.0001 \\ 1.0007 \end{bmatrix},$$

entonces la perturbación relativa de b es

$$\frac{\|\mathbf{b}-\widehat{\mathbf{b}}\|_2}{\|\mathbf{b}\|_2} = \frac{\|\delta\mathbf{b}\|_2}{\|\mathbf{b}\|_2} = 1.054\times 10^{-4} \quad \text{(pequeña)}.$$

Si ahora resolvemos el sistema $A\widehat{\mathbf{x}} = \widehat{\mathbf{b}}$, obtenemos la siguiente solución

$$\widehat{\mathbf{x}} = \mathbf{x} + \delta\mathbf{x} = \begin{bmatrix} 0.9999 \\ -1.9999 \\ 4.0001 \end{bmatrix},$$

la cual es completamente diferente al vector $\mathbf{x}$. Es de notar que en este caso el número de condición de la matriz A es $\kappa_2(A) = 2.7322\times 10^4$ (grande).

1.3.4 Velocidad de Convergencia

En este libro trataremos con frecuencia la sucesión de vectores y su convergencia. Para este propósito, recordemos que una sucesión de vectores $\{\mathbf{x}_k\}$ en $\mathbb{R}^n$, converge a un vector $\mathbf{x}\in\mathbb{R}^n$, y escribimos $\lim_{k\to\infty}\mathbf{x}_k = \mathbf{x}$, si

$$\lim_{k\to\infty} x_i^k = x_i,$$

donde x_i^k y x_i son las componentes de los vectores correspondientes. Por otra parte, decir que la sucesión $\{\mathbf{x}_k\}$ converge $\mathbf{x}$ es equivalente a exigir que la sucesión $\{\mathbf{x}_k-\mathbf{x}\}$ converja a 0. Este hecho unido a que en $\mathbb{R}^n$ las normas son equivalentes, se establece el siguiente resultado que permite definir convergencia de una sucesión de vectores utilizando cualquier norma vectorial en $\mathbb{R}^n$.

Teorema 1.24. *Sea $\|\cdot\|$ una norma vectorial en $\mathbb{R}^n$. Entonces,*

$$\lim_{k\to\infty}\mathbf{x}_k = \mathbf{x} \qquad \Leftrightarrow \qquad \lim_{k\to\infty}\|\mathbf{x}_k-\mathbf{x}\| = 0,$$

donde $\mathbf{x}\in\mathbb{R}^n$ y $\{\mathbf{x}_k\}$ es una sucesión de vectores en $\mathbb{R}^n$.

Un aspecto importante de un método iterativo es su velocidad de convergencia. A continuación presentamos una definición de velocidad de convergencia que emplearemos en distintos capítulos. Supongamos que la sucesión de vectores $\{\mathbf{x}_k\}$ de $\mathbb{R}^n$ converge a un vector $\mathbf{x}^*\in\mathbb{R}^n$.

Definición 1.46. Si existe un número real $p\geq 1$, un entero $k_0>0$ y una constante positiva C independiente de $k_0>0$, tal que

$$\|\mathbf{x}_{k+1}-\mathbf{x}^*\| \leq C\|\mathbf{x}_k-\mathbf{x}^*\|^p \quad \text{para } k\geq K, \tag{1.5}$$

se dice que la sucesión $\{\mathbf{x}_k\}$ converge a $\mathbf{x}^*$ con una Q-velocidad de convergencia de orden p.

Se distinguen tres tipos de convergencias:

1. Cuando $p = 1$ y $C \in (0,1)$, se dice que la sucesión $\{\mathbf{x}_k\}$ converge a $\mathbf{x}^*$ Q-linealmente.
2. Cuando $1 < p < 2$ y $C > 0$, se dice que la sucesión $\{\mathbf{x}_k\}$ converge a $\mathbf{x}^*$ Q-superlinealmente; equivalentemente, una sucesión $\{\mathbf{x}_k\}$ se dice que converge a $\mathbf{x}^*$ Q-superlinealmente, si para alguna sucesión de números positivos $\{c_k\}$ que converge a cero, se cumple
$$\|\mathbf{x}_{k+1} - \mathbf{x}^*\| \leq c_k \|\mathbf{x}_k - \mathbf{x}^*\|.$$
3. Cuando $p = 2$, se dice que la sucesión $\{\mathbf{x}_k\}$ tiene una convergencia Q-cuadrática.

Ejemplo 1.22. Demostrar que la sucesión $\{\frac{1}{2^n}\}_{n=0}^{\infty}$ converge Q-linealmente a 0, la sucesión $\{2^{-(3/2)^n}\}_{n=0}^{\infty}$ converge Q-superlinealmente a 0 y que la sucesión $\{3^{-2^n}\}_{n=0}^{\infty}$ converge Q-cuadráticamente a 0.

Solución. Es claro que las sucesiones $\{\frac{1}{2^n}\}_{n=0}^{\infty}$, $\{2^{-(3/2)^n}\}_{n=0}^{\infty}$ y $\{3^{-2^n}\}_{n=0}^{\infty}$ convergen a 0. Se tiene que
$$\frac{1}{2^{n+1}} = \frac{1}{2} \cdot \frac{1}{2^n}, \quad \text{para todo } n \geq 0,$$
por tanto, la sucesión $\{\frac{1}{2^n}\}_{n=0}^{\infty}$ converge Q-linealmente a 0. Ahora,
$$2^{-(3/2)^{n+1}} = \left(2^{-(3/2)^n}\right)^{3/2}, \quad \text{para todo } n \geq 0,$$
de lo cual se infiere que la sucesión $\{3^{-2^n}\}_{n=0}^{\infty}$ converge Q-superlinealmente a 0. Por último, se tiene que
$$3^{-2^{n+1}} = \left(3^{-2^n}\right)^2, \quad \text{para todo } n \geq 0,$$
de lo cual se infiere que la sucesión $\{3^{-2^n}\}_{n=0}^{\infty}$ converge Q-cuadráticamente a 0. Numérica-mente se puede mostrar la diferencia entre los órdenes de convergencia. En la siguiente tabla se aprecian los siete primeros elementos de las sucesiones $\{\frac{1}{2^n}\}_{n=0}^{\infty}$, $\{2^{-(3/2)^n}\}_{n=0}^{\infty}$ y $\{3^{-2^n}\}_{n=0}^{\infty}$.

¿Qué sucesión se acerca más rápido a 0 y en qué proporción?

n	$\{\frac{1}{2^n}\}_{n=0}^{\infty}$	$\{2^{-(3/2)^n}\}_{n=0}^{\infty}$	$\{3^{-2^n}\}_{n=0}^{\infty}$
0	1.000000000000000000	0.500000000000000000	0.333333333333333315
1	0.500000000000000000	0.353553390593273786	0.111111111111111105
2	0.250000000000000000	0.210224103813428626	0.012345679012345678
3	0.125000000000000000	0.096388176587996297	0.000152415790275873
4	0.062500000000000000	0.029925102521830428	0.000000023230573125
5	0.031250000000000000	0.005176705637342740	0.000000000000000540
6	0.015625000000000000	0.000372460486021616	0.000000000000000000

1.4 Ejercicios

1. Demuestre que $\mathbb{R}^n$ es un espacio vectorial de dimensión n.

2. Si I es la matriz identidad de orden n, demuestre que:

a) $\|I\|_2 = 1$.

b) $\|I\|_F = \sqrt{n}$.

3. Si $D = (d_{ij}) \in \mathbb{R}^{n\times n}$ es una matriz diagonal, demuestre que:

a) $\|D\|_1 = \|D\|_2 = \|D\|_\infty = \max\limits_{i=1,\ldots,n} \{|d_{ii}|\}$.

b) $\|D\|_F = \left(\sum_{i=1}^n d_{ii}^2\right)^{1/2}$.

4. Las p-normas vectoriales y matriciales están relacionadas por varias desigualdades que envuelven las dimensiones m y n. Estas desigualdades hacen que las normas sean equivalentes. Verifique cada una de las siguientes desigualdades y de un ejemplo de un vector o una matriz distintos de cero (para m, n generales) para el cual la desigualdad se cumple. En este problema $\mathbf{x} \in \mathbb{R}^m$ y $A \in \mathbb{R}^{m\times n}$.

a) $\|\mathbf{x}\|_\infty \leq \|\mathbf{x}\|_2$.

b) $\|\mathbf{x}\|_2 \leq \sqrt{m}\|\mathbf{x}\|_\infty$.

c) $\|A\|_\infty \leq \sqrt{n}\|A\|_2$.

d) $\|A\|_2 \leq \sqrt{m}\|A\|_\infty$.

e) $\|A\|_2 \leq \|A\|_F$.

f) $\|A\|_F \leq \sqrt{n}\|A\|_2$.

5. Encuentre en cada uno de los siguientes casos, una fórmula equivalente a la dada que evite la pérdida de cifras significativas.

a) $\ln(x+1) - \ln(x)$, para x grande.

b) $\sqrt{x^2+1} - x$, para x grande.

c) $\cos^2(x) - \text{sen}^2(x)$, para $x \approx \pi/4$.

d) $e^x - x - 1$, para valores negativos de x.

e) $\dfrac{1}{x} - \dfrac{1}{x+1}$, para x grande.

f) $x - \text{sen}(x)$, para valores de x cercanos a cero.

g) $\dfrac{e^x - 1}{x}$, para valores de x cercanos a cero.

6. Determine los errores absoluto y relativo, y el número de dígitos correctos, al aproximar:

a) 0.00001 con 0

b) $\dfrac{1}{3}$ con 0.333

c) 0.3000001 con 0.2999999

7. Suponga que se desea calcular $y = \sqrt{\sum_{i=1}^n a_i^2}$, para $a_i \in \mathbb{R}$. Determine una expresión equivalente para calcular y de manera que se pueda evitar el overflow al calcular $\sum_{i=1}^n a_i^2$, para valores grandes de los números a_i.

8. El formato de *simple precisión* es la representación en punto-flotante que utiliza 32 bits: un bit es usado para el signo del número, 23 bits son dedicados a la mantisa y los restantes 8 bits representan el exponente. ¿Qué respuesta conseguirá usted si calcula las siguientes expresiones en un computador con simple precisión?

a) $\sqrt{10^9 - 1}$

b) $\sqrt{10^{-21}-1}$

c) $10^{16}-5$

9. En doble precisión, ¿cuál de los siguientes cálculos podría resultar en una respuesta distinta de cero?

a) $(100+1/3)-1/3-100$.

b) $(1/3+100)-100-1/3$.

c) $(1/3+100)-(100+1/3)$.

Revise sus predicciones usando la computadora.

10. *La Fórmula Mejorada para la Resolución de la Ecuación de Segundo Grado*. Supongamos que $a \neq 0$ y que $b^2-4ac>0$ y consideremos la ecuación $ax^2+bx+c=0$. Sus raíces pueden hallarse mediante la conocida fórmula

$$x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a} \qquad \text{y} \qquad x_2 = \frac{-b-\sqrt{b^2-4ac}}{2a}. \tag{1.6}$$

Demuestre que estas raíces pueden calcularse mediante las fórmulas equivalentes:

$$x_1 = \frac{-2c}{b+\sqrt{b^2-4ac}} \qquad \text{y} \qquad x_2 = \frac{-2c}{b-\sqrt{b^2-4ac}}. \tag{1.7}$$

Observación: Cuando $|b| \approx \sqrt{b^2-4ac}$, se debe proceder con cuidado para evitar la pérdida de precisión por cancelación. Si $b>0$, entonces x_1 debería calcularse con (1.7) y x_2 debería calcularse con (1.6); mientras que, si $b<0$, entonces x_1 debería calcularse con (1.6) y x_2 debería calcularse con (1.7).

11. Use la fórmula adecuada para calcular x_1 y x_2, tal como se explica en el Ejercicio 9, para hallar las raíces de las siguientes ecuaciones de segundo grado.

a) $x^2-106x+1=0$

b) $10^{-10}x2-10^{10}x+10^{10}=0$

c) $x^2-1000000000001x+1=0$

12. Determine el número de operaciones punto-flotante necesarias para realizar las siguientes operaciones matriciales:

a) El producto AB, donde $A \in \mathbb{R}^{m\times n}$ y $B \in \mathbb{R}^{n\times p}$.

b) El producto $A\mathbf{v}$, donde $A \in \mathbb{R}^{m\times n}$ y $\mathbf{v} \in \mathbb{R}^n$.

c) El producto escalar $\mathbf{u}^T\mathbf{v}$, donde $\mathbf{u},\mathbf{v} \in \mathbb{R}^n$.

d) El producto $\mathbf{u}\mathbf{v}^T$, donde $\mathbf{u},\mathbf{v} \in \mathbb{R}^n$ con $\mathbf{u}$ un vector columna y $\mathbf{v}^T$ un vector fila.

e) El cálculo de la matriz $A = \frac{\mathbf{u}\mathbf{v}^T}{\mathbf{u}^T\mathbf{u}}$, donde $\mathbf{u},\mathbf{v} \in \mathbb{R}^n$.

f) El cálculo de la matriz $B = A - \mathbf{u}\mathbf{v}^T$, donde $A \in \mathbb{R}^{n\times n}$ y $\mathbf{u},\mathbf{v} \in \mathbb{R}^n$.

13. Para cada una de las siguientes funciones $f(x)$, calcule el número de condición κ_P del problema: calcular un valor de la función $f(x)$, para los números $x \in \mathbb{R}$ indicados.

a) $\ln(x+1)-\ln(x)$, para $x \in [1,2]$.

b) $\sqrt{x^2+1}-x$, para $x \in [0,1]$.

c) $\cos^2(x) - \text{sen}^2(x)$, para $x \approx \pi/4$.

d) $e^x - x - 1$, para $x \in [0, 2\pi]$.

e) $\frac{1}{x} - \frac{1}{x+1}$, para $x \in [1000, 5000]$.

f) $x - \text{sen}(x)$, para $x \in [0,1]$.

g) $\dfrac{e^x - 1}{x}$, para $x \in [0,1]$.

14. Sea $U = (u_{ij})$ una matriz triangular superior no singular. Demuestre que el número de condición $\kappa_\infty(U)$ satisface:

$$\kappa_\infty(U) \geq \frac{\text{máx}_{i=1,\ldots,n}\{|u_{ii}|\}}{\text{mín}_{i=1,\ldots,n}\{|u_{ii}|\}}$$

15. Dada la matriz $A = \begin{bmatrix} 1 & a \\ a & 1 \end{bmatrix}$, determine:

a) Los valores de a para que A sea mal-condicionada.

b) El número de condición de condición de A, $\kappa_2(A)$.

16. Use los resultados de los Ejercicios 9 y 10 para construir un algoritmo y un programa en Python, que calcule las raíces de una ecuación cuadrática en todas las situaciones posibles, incluyendo los casos problemáticos cuando $|b| \approx \sqrt{b^2 - 4ac}$.

17. a) Escriba un programa para calcular la media $\overline{x}$ y la desviación estándar σ de una sucesión finita x_i. Su programa debe aceptar un vector $\mathbf{x} = (x_1, \ldots, x_n)$ de dimensión n como entrada y producir la media y la desviación estándar de la sucesión como salida. Para la desviación estándar, utilice la fórmula de dos pasos

$$\sigma = \left[\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \overline{x})^2 \right]^{1/2}$$

y la fórmula de un paso

$$\sigma = \left[\frac{1}{n-1} \left(\sum_{i=1}^{n} x_i^2 - n\overline{x}^2 \right) \right]^{1/2}$$

y compare los resultados para una sucesión de entrada de su elección.

b) ¿Puede construir una sucesión de datos de entrada que ilustre dramáticamente la diferencia numérica entre estas dos fórmulas matemáticamente equivalentes?
(*Precaución*: tenga cuidado de tomar la raíz cuadrada de un número negativo).

18. a) Escriba un programa en Python para construir la matriz $A = (a_{ij}) \in \mathbb{R}^{n \times n}$ triangular inferior ([6]):

$$a_{ij} = \begin{cases} 1, & \text{si } i = j, \\ -1, & \text{si } i > j, \\ 0, & \text{si } i < j. \end{cases}$$

b) Realice un experimento para mostrar que la solución de $A\mathbf{x} = \mathbf{b}$ con $A \in \mathbb{R}^{n\times n}$ como en a) y $\mathbf{b} \in \mathbb{R}^n$ tal que $\mathbf{b} = A\mathbf{x}$, donde $\mathbf{x} = (1,1,\ldots,1)$, es cada vez menos preciso cuando n aumenta, a causa del mal-condicionamiento de A. Sea $\mathbf{x}^*$ la solución obtenida como:

$$\mathbf{x}^* = \texttt{np.linalg.solve(A,b)}.$$

Presente sus resultados de la siguiente forma:

n	$\kappa_2(A)$	$\mathbf{x}^*$	$\frac{\|\mathbf{x}-\mathbf{x}^*\|_2}{\|\mathbf{x}\|_2}$	$\frac{\|\mathbf{b}-A\mathbf{x}^*\|_2}{\|\mathbf{b}\|_2}$
10				
20				
30				
40				
50				

2 | Sistemas de Ecuaciones Lineales

Muchas relaciones en la naturaleza son lineales, esto es, que los efectos son proporcionales a sus causas. En electricidad, por ejemplo, la ley de Ohm, $V = iR$, dice que la tensión (diferencia de potencial) entre los extremos de un conductor es proporcional a la corriente que fluye a través de él, donde la constante de proporcionalidad es la resistencia. En dinámica, la Segunda ley de Newton, $F = ma$, dice que la fuerza es proporcional a la aceleración, donde la constante de proporcionalidad es la masa del objeto. O, en elasticidad, la Ley de Hooke dice que el alargamiento unitario que experimenta un material es proporcional a la tensión, donde la constante de proporcionalidad es el módulo de Young.

La necesidad de resolver simultáneamente un conjunto de ecuaciones lineales aparece en muchas aplicaciones. Los sistemas lineales pueden aparecer directamente de situaciones físicas o pueden aparecer en forma indirecta para encontrar la solución de problemas más complejos, tales como ecuaciones diferenciales en derivadas parciales. El propósito de este capítulo es la resolución numérica de un sistema de ecuaciones lineales de la forma

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ a_{31}x_1 + a_{32}x_2 + \cdots + a_{3n}x_n &= b_3 \\ \qquad\qquad\qquad\qquad \vdots & \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n. \end{cases} \tag{2.1}$$

Éste es un sistema de n ecuaciones y n incógnitas $x_1, x_2, \ldots, x_n$, donde los elementos a_{ij} y b_i son números reales fijos. El sistema (2.1) se puede representar como:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{bmatrix}$$

Esta manera de escribir un sistema de ecuaciones lineales se denomina representación matricial. Ahora, denotando respectivamente las matrices por A, $\mathbf{x}$ y $\mathbf{b}$, entonces la representación matricial del sistema de ecuaciones lineales se reduce simplemente a:

$$A\mathbf{x} = \mathbf{b}.$$

En este capítulo estudiaremos dos tipos de métodos para resolver el sistema (2.1): *directos* e *iterativos*. Los métodos directos requieren un número finito de pasos para hallar la solución del sistema lineal. Particularmente, estudiaremos el método directo denominado eliminación Gaussiana con y sin pivoteo; además, construiremos las siguientes factorizaciones: LU, Cholesky, QR y SVD, que las utilizaremos para hallar la solución del sistema (2.1) y calcular el determinante y la inversa de una matriz. Es de notar que estas factorizaciones son muy utilizadas en distintas aplicaciones. Por otra parte, los métodos iterativos hallan una aproximación de la solución del sistema

(2.1), a través de un proceso iterativo que se detiene bajo cierto criterio de parada. Los métodos iterativos a su vez se clasifican en métodos estacionarios y no estacionarios. Describiremos los métodos estacionarios clásicos: Jacobi, Gauss-Seidel y SOR. Los métodos no estacionarios que estudiaremos en este capítulo son los métodos basados en subespacios de Krylov. Para el caso especial en que A es una matriz simétrica definida positiva, existen otros métodos no estacionarios (método de Cauchy y el método de gradientes conjugados), que estudiaremos en el Capítulo 8.

2.1 Conceptos Básicos

Consideremos el sistema de ecuaciones lineales

$$A\mathbf{x} = \mathbf{b}, \tag{2.2}$$

donde A es una matriz cuadrada de orden n, $\mathbf{x} \in \mathbb{R}^n$ es el *vector de incógnitas* y $\mathbf{b} \in \mathbb{R}^n$ es un vector fijo denominado *vector del lado derecho*. Al resolver el sistema lineal (2.2) se distinguen tres casos:

1. La matriz A es no singular. Existe una única solución dada por $\mathbf{x} = A^{-1}\mathbf{b}$.
2. La matriz A es singular y $\mathbf{b} \in \text{alc}(A)$. El sistema tiene infinitas soluciones.
3. La matriz A es singular y $\mathbf{b} \notin \text{alc}(A)$. El sistema no tiene solución.

Ejemplo 2.1. La ilustración más simple de los casos anteriores se aprecia con matrices diagonales. Sean

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 4 \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} 1 \\ 0 \\ 8 \end{bmatrix}.$$

Así, A es no singular y

$$\mathbf{x} = \begin{bmatrix} 1 \\ 0 \\ 2 \end{bmatrix}$$

es la única solución del sistema lineal $A\mathbf{x} = \mathbf{b}$. Ahora, sean

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix}.$$

Luego, A es singular y $\mathbf{b} \in \text{alc}(A)$, ya que $\mathbf{b}$ se puede escribir como una combinación lineal de las dos primeras columnas de A. Así, el sistema lineal $A\mathbf{x} = \mathbf{b}$ tiene infinitas soluciones de la forma

$$\mathbf{x} = \begin{bmatrix} 1 \\ 1 \\ \alpha \end{bmatrix}, \qquad \text{para } \alpha \in \mathbb{R}.$$

Finalmente, sea A la misma matriz del caso anterior, pero el vector del lado derecho es:

$$\mathbf{b} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

En este caso, A es singular y $\mathbf{b} \notin \text{alc}(A)$; por lo tanto, el sistema lineal $A\mathbf{x} = \mathbf{b}$ no tiene solución.

En Python se puede utilizar el comando `solve` del módulo `linalg` de la librería `Numpy`, para resolver el sistema de ecuaciones $A\mathbf{x} = \mathbf{b}$. Veamos como se hace.

```
>>> A = np.array([[1,0,0],[0,2,0],[0,0,4]])
>>> b = np.array([1,0,8])
>>> x = np.linalg.solve(A,b)
>>> print(x)
[ 1.  0.  2.]
>>> A = np.array([[1,0,0],[0,2,0],[0,0,0]])
>>> b = np.array([1,2,0])
>>> x = np.linalg.solve(A,b)
...
numpy.linalg.linalg.LinAlgError: Singular matrix
```

2.1.1 Solución de un Sistema de Ecuaciones Lineales por Determinantes

A continuación describiremos un procedimiento para hallar la solución de un sistema de ecuaciones lineales $A\mathbf{x} = \mathbf{b}$, expresada en términos de determinantes. Tal procedimiento se conoce como *regla de Cramer*, que se establece en el siguiente teorema.

Teorema 2.1 (Regla de Cramer)**.** *Si* $\det(A) \neq 0$*, entonces la solución* $\mathbf{x} = (x_i, \ldots, x_n)$ *del sistema de ecuaciones lineales* $A\mathbf{x} = \mathbf{b}$ *está dada por*

$$x_j = \frac{\Delta_j}{\Delta}, \qquad \text{para } j = 1, 2, \ldots, n,$$

donde $\Delta = \det(A)$ *y* $\Delta_j = \det(A^{(j)})$*, con* $A^{(i)}$ *la matriz obtenida reemplazando la j-ésima columna de A por* $\mathbf{b}$.

Veamos un ejemplo del uso de la regla de Cramer.

Ejemplo 2.2. Mediante la regla de Cramer resolvamos el sistema de ecuaciones lineales $A\mathbf{x} = \mathbf{b}$, donde

$$A = \begin{bmatrix} 3 & 2 & 4 \\ 2 & -1 & 1 \\ 1 & 2 & 3 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}.$$

Se tiene que

$$\Delta = \begin{vmatrix} 3 & 2 & 4 \\ 2 & -1 & 1 \\ 1 & 2 & 3 \end{vmatrix} = -5, \qquad \Delta_1 = \begin{vmatrix} 1 & 2 & 4 \\ 0 & -1 & 1 \\ 1 & 2 & 3 \end{vmatrix} = 1,$$

$$\Delta_2 = \begin{vmatrix} 3 & 1 & 4 \\ 2 & 0 & 1 \\ 1 & 1 & 3 \end{vmatrix} = 0, \qquad \Delta_3 = \begin{vmatrix} 3 & 2 & 1 \\ 2 & -1 & 0 \\ 1 & 2 & 1 \end{vmatrix} = -2.$$

Por el Teorema 2.1, la solución $\mathbf{x} = (x_i, x_2, x_2)$ del sistema $A\mathbf{x} = \mathbf{b}$ está dada por:

$$x_1 = \frac{\Delta_1}{\Delta} = -\frac{1}{5}, \quad x_2 = \frac{\Delta_2}{\Delta} = 0, \quad x_3 = \frac{\Delta_3}{\Delta} = \frac{2}{5}.$$

Es evidente que el uso de la regla de Cramer es netamente teórico, ya que el costo computacional de la misma es sumamente alta. Si quisiéramos utilizar la regla de Cramer para hallar la solución de un sistema de ecuaciones lineales de orden n, necesitaríamos calcular $n+1$ determinantes de orden n cada uno; por lo que requeriríamos más de $(n+1)!$ operaciones punto-flotante para llevar a cabo el cálculo, lo cual es excesivamente grande; por ejemplo, si $n = 20$, entonces son necesarias aproximadamente $21 \cdot 20! \approx 5.10909421717094 \times 10^{19}$ operaciones punto-flotante para resolver un sistema lineal de orden 20.

Si una PC normal (Core i5 2.4 GHz) realiza una operación punto-flotante en aproximadamente 2.4×10^{-9} segundos, ¿cuánto tardaría en resolver un sistema lineal de orden 20 usando la regla de Cramer? ¿38 siglos?

2.1.2 Sistemas Lineales Especiales

La estructura de la matriz A juega un rol importante en la resolución de un sistema de ecuaciones lineales $A\mathbf{x} = \mathbf{b}$. Para ciertas aplicaciones los sistema lineales asociados poseen una estructura específica (por ejemplo, la matriz A tiene muchos elementos nulos). En esos casos, generalmente las características estructurales permiten hallar una solución del sistema en forma directa. Describiremos tres sistemas lineales especiales producidos por estructuras muy particulares de A: *diagonales*, *triangulares* y *bidiagonales*.

Sistemas Lineales Diagonales

Un sistema lineal $A\mathbf{x} = \mathbf{b}$ se dice diagonal si la matriz A es diagonal. Si A es una matriz diagonal, esto es, $A = \text{diag}(a_{11}, a_{22}, \ldots, a_{nn})$, entonces la solución del sistema de ecuaciones lineales $A\mathbf{x} = \mathbf{b}$, es el vector $\mathbf{x} = (x_1, \ldots, x_n)$ tal que:

$$x_i = \frac{b_i}{a_{ii}}, \quad \text{para } a_{ii} \neq 0,\ i = 1, 2, \ldots, n.$$

Note que si $a_{kk} = 0$ y $b_k = 0$, para algún k, entonces el sistema tiene infinitas soluciones; ahora, si $a_{kk} = 0$ y $b_k \neq 0$, entonces el sistema no tiene solución. El número total de operaciones de punto-flotante para obtener una solución de un sistema diagonal es n, es decir, es de orden $\mathbf{O}(n)$.

Sistemas Lineales Bidiagonales

Distinguiremos dos tipos de sistemas lineales bidiagonales: *bidiagonal superior* y *bidiagonal inferior*. Un sistema lineal $A\mathbf{x} = \mathbf{b}$ se dice bidiagonal superior si la matriz A es una matriz bidiagonal superior. De forma análoga, el sistema lineal $A\mathbf{x} = \mathbf{b}$ se dice bidiagonal inferior si la matriz A es bidiagonal inferior. En un sistema lineal bidiagonal superior la matriz A tiene la forma:

$$A = \begin{bmatrix} a_{11} & a_{12} & & \\ & a_{22} & \ddots & \\ & & \ddots & a_{n-1,n} \\ & & & a_{nn} \end{bmatrix}$$

En este caso, el vector solución $\mathbf{x} = (x_1, \ldots, x_n)^T$ del sistema bidiagonal superior $A\mathbf{x} = \mathbf{b}$, se obtiene como:

$$x_i = \begin{cases} \dfrac{b_n}{a_{nn}}, & i = n, \\ \dfrac{b_i - a_{i,i+1}x_{i+1}}{a_{ii}}, & i = n-1, \ldots, 1, \end{cases}$$

siempre que $a_{ii} \neq 0$ para todo $i = 1,2,\ldots,n$. Ahora, cuando el sistema lineal $A\mathbf{x} = \mathbf{b}$ es bidiagonal inferior la matriz A tiene la forma:

$$A = \begin{bmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ & \ddots & \ddots & \\ & & a_{n,n-1} & a_{nn} \end{bmatrix}$$

y el vector solución $\mathbf{x} = (x_1,\ldots,x_n)^T$ del sistema bidiagonal inferior $A\mathbf{x} = \mathbf{b}$, se obtiene como:

$$x_i = \begin{cases} \dfrac{b_1}{a_{11}}, & i = n, \\ \dfrac{b_i - a_{i-1,i}x_{i-1}}{a_{ii}}, & i = 2,\ldots,n, \end{cases}$$

siempre que $a_{ii} \neq 0$ para todo $i = 1,2,\ldots,n$.

Para obtener la solución de un sistema bidiagonal son necesarias n divisiones, $n-1$ multiplicaciones y $n-1$ sumas; por tanto, el número total de operaciones punto-flotante para obtener una solución de un sistema bidiagonal es $3n-2$, es decir, es de orden $\mathbf{O}(n)$.

Sistemas Lineales Triangulares

Distinguiremos dos tipos de sistemas lineales triangulares: *triangular superior* y *triangular inferior*. Un sistema lineal $A\mathbf{x} = \mathbf{b}$ se dice triangular superior si la matriz A es una matriz triangular superior. De forma análoga, el sistema lineal $A\mathbf{x} = \mathbf{b}$ se dice triangular inferior si la matriz A es triangular inferior.

Describamos un método para hallar la solución de un sistema lineal triangular superior. La forma estándar de un sistema lineal triangular superior es:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1,n-1}x_{n-1} + a_{1n}x_n &= b_1 \\ a_{22}x_2 + a_{23}x_3 + \cdots + a_{2,n-1}x_{n-1} + a_{2n}x_n &= b_2 \\ a_{33}x_3 + \cdots + a_{3,n-1}x_{n-1} + a_{3n}x_n &= b_3 \\ \vdots & \\ a_{n-1,n-1}x_{n-1} + a_{n-1,n}x_n &= b_{n-1} \\ a_{nn}x_n &= b_n. \end{cases} \tag{2.3}$$

Asumiendo que $a_{nn} \neq 0$, entonces la incógnita x_n del vector solución $\mathbf{x} = (x_1,\ldots,x_n)$ del sistema de ecuaciones lineales (2.3) se puede despejar de la última ecuación,

$$x_n = b_n/a_{nn}.$$

Ahora, asumiendo que $a_{ii} \neq 0$ para $i = 2,\ldots,n$, entonces el resto de las incógnitas del sistema (2.3) se pueden despejar como:

$$x_i = \left[b_i - \sum_{j=i+1}^{n} a_{ij}x_j \right] / a_{ii}, \quad \text{para } i = n-1, n-2, \ldots, 1.$$

Por supuesto, el proceso anterior, denominado *sustitución regresiva*, funciona sólo en el caso de que los elementos de la diagonal de A son todos distintos de cero. Si para cierto k se tiene que

$a_{kk} = 0$, entonces el proceso de sustitución regresiva se detiene. En este caso, el sistema quizás no tenga solución o pudiera tener infinitas soluciones.

El número de multiplicaciones y divisiones que se realizan en la sustitución regresiva es igual a $n(n+1)/2$, mientras que el número de sumas y restas es $n(n-1)/2$. El número total de operaciones punto-flotante de la sustitución regresiva es n^2, es decir, es de orden $\mathbf{O}(n^2)$. Este número nos puede orientar acerca del tiempo que requiere una PC normal para resolver un sistema lineal triangular superior. Una PC normal (Core i5 2.4 GHz) realiza una operación punto-flotante en aproximadamente 2.4×10^{-9} segundos. Ahora, si el sistema es de dimensión $n = 10^4$, a una PC normal le llevaría aproximadamente 0.24 segundos hallar la solución, claro, utilizando la sustitución regresiva. Si la dimensión fuese $n = 10^5$, la PC tardaría aproximadamente 24 segundos para hallar la solución. Pero, si la dimensión fuese $n = 10^6$, la PC tardaría aproximadamente 40 minutos para hallar la solución. Se deja al lector calcular el tiempo requerido por una PC normal para hallar la solución de un sistema triangular superior de dimensión $n = 10^9$ (¿76 años?).

Cuando el sistema $A\mathbf{x} = \mathbf{b}$ es triangular inferior, existe un proceso semejante a la sustitución regresiva para hallar la solución de este sistema. Tal proceso se denomina *sustitución progresiva*. En este caso, el vector solución $\mathbf{x} = (x_1, \ldots, x_n)$ del sistema triangular inferior $A\mathbf{x} = \mathbf{b}$, se obtiene como:

$$x_i = \begin{cases} \dfrac{b_1}{a_{11}}, & i = 1, \\ \dfrac{b_i - \displaystyle\sum_{j=1}^{i} a_{ij} x_j}{a_{ii}}, & i = 2, \ldots, n. \end{cases}$$

Por supuesto, el proceso anterior funciona sólo en el caso de que los elementos de la diagonal de A sean todos distintos de cero. Si para cierto k se tiene que $a_{kk} = 0$, entonces el proceso de sustitución progresiva se detiene.

En los Programas 2.1 y 2.2 implementamos los métodos sustitución regresiva y sustitución progresiva, respectivamente. Tales funciones las denominados `Sust_Regresiva` y `Sust_Progresiva`.

Programa 2.1. Sustitución Regresiva

```
def Sust_Regresiva(A, b):
    A,b = np.asarray(A, dtype=float),np.asarray(b, dtype=float)
    n = A.shape[-1]
    x = np.zeros_like(b)
    if A[n-1,n-1] == 0:
        print('Elemento nulo en la diagonal de A')
        return x
    x[n-1] = b[n-1]/A[n-1,n-1]
    for i in reversed(range(0,n-1)):
        if A[i,i]==0:
            print('Elemento nulo en la diagonal de A')
            return
        x[i] = (b[i] - np.sum(A[i,i+1:]*x[i+1:]))/A[i,i]
    return x
```

Programa 2.2. Sustitución Progresiva

```
def Sust_Progresiva(A,b):
    A,b = np.asarray(A,dtype=float),np.asarray(b,dtype=float)
    n = A.shape[-1]
    x = np.zeros_like(b)
    if(A[0,0] == 0):
        print('Elemento nulo en la diagonal de A')
        return x
    x[0] = b[0]/A[0,0]
    for i in range(1,n):
        if(A[i,i]==0):
            print('Elemento nulo en la diagonal de A')
            return
        x[i] = (b[i] - np.sum(A[i,0:i+1]*x[0:i+1]))/A[i,i]
    return x
```

Ejemplo 2.3. Ilustremos el uso de la función `Sust_Regresiva`. Sean A y $\mathbf{b}$ dados por

$$A = \begin{bmatrix} 1 & -1 & 2 & 0 & 3 \\ 0 & -3 & 5 & 0 & 0 \\ 0 & 0 & 10 & -7 & 0 \\ 0 & 0 & 0 & 8 & 1 \\ 0 & 0 & 0 & 0 & -6 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 5 \\ 4 \\ 3 \\ 2 \\ 1 \end{bmatrix}.$$

La solución del sistema $A\mathbf{x} = \mathbf{b}$ es $\mathbf{x} = \left(\frac{1153}{288}, -\frac{149}{288}, \frac{47}{96}, \frac{13}{48}, -\frac{1}{6}\right)$. Usando la función `Sust_Regresiva` se tiene:

```
>>> A = np.array([[1,-1,2,0,3],
... [0,-3,5,0,0],
... [0,0,10,-7,0],
... [0,0,0,8,1],
... [0,0,0,0,-6]])
>>> b = np.array([5,4,3,2,1])
>>> x = Sust_Regresiva(A,b)
>>> print(x)
[ 4.00347222 -0.51736111  0.48958333  0.27083333 -0.16666667]
```

Ahora, si resolvemos el sistema $Ax = b$ con el comando `solve` del módulo `linalg` de la librería `Numpy`, obtenemos:

```
>>> x = np.linalg.solve(A,b)
>>> print(x)
[ 4.00347222 -0.51736111  0.48958333  0.27083333 -0.16666667]
```

2.2 Eliminación Gaussiana

Sea $A\mathbf{x} = \mathbf{b}$ un sistema lineal, donde $A = (a_{ij})$ es una matriz cuadrada de orden n y $\mathbf{b}$ es un vector de n componentes. El método de eliminación gaussiana tiene por objetivo reducir el sistema $A\mathbf{x} = \mathbf{b}$ a un sistema equivalente (es decir, que tiene la misma solución) de la forma $U\mathbf{x} = \widehat{\mathbf{b}}$,

donde U es una matriz triangular superior y $\widehat{\mathbf{b}}$ es el vector del lado derecho actualizado (es decir, el vector $\mathbf{b}$ modificado). Entonces, este último sistema se puede resolver por el proceso de sustitución regresiva.

2.2.1 Eliminación Gaussiana Simple

Describamos el procedimiento de reducción de la eliminación gaussiana simple (sin pivoteo). Inicialmente se construye la matriz ampliada del sistema $A\mathbf{x} = \mathbf{b}$, que está dada por

$$[A|\mathbf{b}] = \left[\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} & b_n \end{array}\right].$$

La eliminación gaussiana realiza transformaciones a la matriz ampliada, que no cambian el conjunto de soluciones del sistema de ecuaciones lineales, con el objeto de construir la matriz ampliada $[U|\widehat{\mathbf{b}}]$ del sistema reducido. En el siguiente teorema se resumen las transformaciones elementales de filas que se emplean en la eliminación gaussiana.

Teorema 2.2 (Transformaciones elementales de fila)**.** *Sea f_i la fila i de la matriz ampliada $[A|\mathbf{b}]$, para $i = 1,2,\ldots,n$. Cualquiera de las siguientes transformaciones elementales aplicada a la matriz ampliada produce un sistema equivalente:*

i) Intercambio: $f_i \leftrightarrow f_j$, las filas i y j se intercambian.

ii) Escalado: αf_i, la fila i se multiplica por el escalar α.

iii) Sustitución: $f_i + \alpha f_j \to f_i$, la fila i se reemplaza por la suma de esa fila más un múltiplo de la fila j.

Para describir el proceso de eliminación gaussiana es necesario introducir los conceptos de pivotes y multiplicadores.

Definición 2.1 (Pivotes y multiplicadores)**.** El elemento a_{jj} de la matriz ampliada $[A|\mathbf{b}]$ se llama j-ésimo *pivote*, la fila j-ésima se llama *fila pivote* y la columna j-ésima se llama *columna pivote*. Los números $m_{ij} = a_{ij}/a_{jj}$, para $a_{jj} \neq 0$ y $i = j+1, j+2, \ldots, n$, se llaman *multiplicadores*.

Asumamos que el pivote $a_{11} \neq 0$, en caso contrario el proceso de eliminación gaussiana se detiene. Inicialmente calculamos los multiplicadores

$$m_{i1} = \frac{a_{i1}}{a_{11}}, \quad i = 2,3,\ldots,n.$$

Aplicando las transformaciones elementales

$$f_i - m_{i1} f_1 \to f_i, \quad i = 2,3,\ldots,n,$$

a la matriz ampliada, se hacen cero todos los elementos de la columna 1 que están debajo del pivote a_{11}. Así obtenemos una nueva matriz ampliada, cuyos elementos, sin pérdida de generalidad, se

denotarán nuevamente como a_{ij} y b_j (en los siguientes pasos también usaremos esta notación):

$$\left[\begin{array}{ccccc|c} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} & b_1 \\ 0 & a_{22} & a_{23} & \cdots & a_{2n} & b_2 \\ 0 & a_{32} & a_{33} & \cdots & a_{3n} & b_3 \\ \vdots & \vdots & \vdots & & \vdots & \\ 0 & a_{n2} & a_{n3} & \cdots & a_{nn} & b_n \end{array}\right].$$

Supongamos que el pivote $a_{22} \neq 0$. Calculamos ahora los multiplicadores

$$m_{i2} = \frac{a_{i2}}{a_2}, \quad i = 3,4,\ldots,n.$$

Aplicando las transformaciones elementales

$$f_i - m_{i2} f_2 \to f_i, \quad i = 3,4,\ldots,n,$$

a la matriz ampliada, se hacen cero todos los elementos de la columna 2 que están debajo del pivote a_{22}, con lo cual obtenemos una nueva matriz ampliada:

$$\left[\begin{array}{ccccc|c} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} & b_1 \\ 0 & a_{22} & a_{23} & \cdots & a_{2n} & b_2 \\ 0 & 0 & a_{33} & \cdots & a_{3n} & b_3 \\ \vdots & \vdots & \vdots & & \vdots & \\ 0 & 0 & a_{n3} & \cdots & a_{nn} & b_n \end{array}\right].$$

Asumamos que al hacer cero los elementos de la $(j-1)$-ésima columna debajo del $(j-1)$-ésimo pivote $a_{j-1,j-1}$, se obtiene que $a_{jj} \neq 0$ para $j = 2,\ldots,n-1$. En este caso, calculando secuencialmente los multiplicadores, para $j = 2,\ldots,n-1$,

$$m_{ij} = \frac{a_{ij}}{a_{jj}}, \quad i = j+1, j+2,\ldots,n$$

y luego aplicando las siguientes transformaciones elementales, para $j = 2,\ldots,n-1$,

$$f_i - m_{ij} f_j \to f_i, \quad i = j+1, j+2,\ldots,n \tag{2.4}$$

obtenemos la matriz ampliada

$$[U,\widehat{\mathbf{b}}] = \left[\begin{array}{ccccc|c} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} & b_1 \\ 0 & a_{22} & a_{23} & \cdots & a_{2n} & b_2 \\ 0 & 0 & a_{33} & \cdots & a_{3n} & b_3 \\ \vdots & \vdots & \ddots & & \vdots & \\ 0 & 0 & \cdots & 0 & a_{nn} & b_n \end{array}\right].$$

Esta última matriz ampliada está asociada a un sistema lineal triangular superior, cuya solución se halla utilizando el proceso de sustitución regresiva.

Note que si en algún paso de la eliminación gaussiana se obtiene que el j-ésimo pivote es cero, $a_{jj} = 0$, entonces el proceso se detiene. Esto no necesariamente significa que el sistema lineal no tiene solución, sino que con el método de la eliminación gaussian simple no se puede hallar la solución. De esta forma, en caso de que $a_{jj} = 0$, es necesario introducir una estrategia que sirva para escoger otro elemento no nulo de la columna j que tome el lugar del actual pivote a_{jj} y luego nos permita hacer cero los elementos de la j-ésima columna debajo del pivote a_{jj}. Tal estrategia se denomina *pivoteo,* que describiremos más adelante.

Ejemplo 2.4. Usaremos la eliminación gaussiana simple para hallar un sistema triangular superior equivalente a un sistema de ecuaciones dado. Sea el siguiente sistema de ecuaciones:

$$\begin{cases} x_1+2x_2+3x_3+5x_4 &= 4\\ 2x_1+9x_2-4x_3-x_4 &= 1\\ x_1+8x_2+9x_3 &= 3\\ x_1+x_2+x_3+x_4 &= 0 \end{cases}$$

La matriz ampliada de este sistema es:

$$[A|\mathbf{b}] = \left[\begin{array}{cccc|c} 1 & 2 & 3 & 5 & 4\\ 2 & 9 & -4 & -1 & 1\\ 1 & 8 & 9 & 0 & 3\\ 1 & 1 & 1 & 1 & 0 \end{array}\right].$$

Apliquemos la eliminación gaussiana:

- Hacer cero los elementos de la primera columna debajo de la diagonal principal. Calculamos los multiplicadores:

$$m_{21}=\frac{a_{21}}{a_{11}}=2, \quad m_{31}=\frac{a_{31}}{a_{11}}=1, \quad m_{41}=\frac{a_{41}}{a_{11}}=1.$$

Realizamos las transformaciones elementales de fila:

$$\left[\begin{array}{cccc|c} 1 & 2 & 3 & 5 & 4\\ 2 & 9 & -4 & -1 & 1\\ 1 & 8 & 9 & 0 & 3\\ 1 & 1 & 1 & 1 & 0 \end{array}\right] \xrightarrow{f_2-m_{21}f_1\to f_2} \left[\begin{array}{cccc|c} 1 & 2 & 3 & 5 & 4\\ 0 & 5 & -10 & -11 & -7\\ 1 & 8 & 9 & 0 & 3\\ 1 & 1 & 1 & 1 & 0 \end{array}\right]$$

$$\left[\begin{array}{cccc|c} 1 & 2 & 3 & 5 & 4\\ 0 & 5 & -10 & -11 & -7\\ 1 & 8 & 9 & 0 & 3\\ 1 & 1 & 1 & 1 & 0 \end{array}\right] \xrightarrow{f_3-m_{31}f_1\to f_3} \left[\begin{array}{cccc|c} 1 & 2 & 3 & 5 & 4\\ 0 & 5 & -10 & -11 & -7\\ 0 & 6 & 6 & -5 & -1\\ 1 & 1 & 1 & 1 & 0 \end{array}\right]$$

$$\left[\begin{array}{cccc|c} 1 & 2 & 3 & 5 & 4\\ 0 & 5 & -10 & -11 & -7\\ 0 & 6 & 6 & -5 & -1\\ 1 & 1 & 1 & 1 & 0 \end{array}\right] \xrightarrow{f_4-m_{41}f_1\to f_4} \left[\begin{array}{cccc|c} 1 & 2 & 3 & 5 & 4\\ 0 & 5 & -10 & -11 & -7\\ 0 & 6 & 6 & -5 & -1\\ 0 & -1 & -2 & -4 & -4 \end{array}\right]$$

- Hacer cero los elementos de la segunda columna debajo de la diagonal principal. Calculamos los multiplicadores:

$$m_{32}=\frac{a_{32}}{a_{22}}=\frac{6}{5}, \quad m_{42}=\frac{a_{42}}{a_{22}}=-\frac{1}{5}.$$

Realizamos las transformaciones elementales de fila:

$$\left[\begin{array}{cccc|c} 1 & 2 & 3 & 5 & 4\\ 0 & 5 & -10 & -11 & -7\\ 0 & 6 & 6 & -5 & -1\\ 0 & -1 & -2 & -4 & -4 \end{array}\right] \xrightarrow{f_3-m_{32}f_2\to f_3} \left[\begin{array}{cccc|c} 1 & 2 & 3 & 5 & 4\\ 0 & 5 & -10 & -11 & -7\\ 0 & 0 & 18 & \frac{41}{5} & \frac{37}{5}\\ 0 & -1 & -2 & -4 & -4 \end{array}\right]$$

$$\left[\begin{array}{cccc|c} 1 & 2 & 3 & 5 & 4 \\ 0 & 5 & -10 & -11 & -7 \\ 0 & 0 & 18 & \frac{41}{5} & \frac{37}{5} \\ 0 & -1 & -2 & -4 & -4 \end{array}\right] \xrightarrow{f_4 - m_{42} f_2 \to f_4} \left[\begin{array}{cccc|c} 1 & 2 & 3 & 5 & 4 \\ 0 & 5 & -10 & -11 & -7 \\ 0 & 0 & 18 & \frac{41}{5} & \frac{37}{5} \\ 0 & 0 & -4 & -\frac{31}{5} & -\frac{27}{5} \end{array}\right]$$

- Hacer cero los elementos de la tercera columna debajo de la diagonal principal. Calculamos el multiplicador:

$$m_{43} = \frac{a_{43}}{a_{33}} = -\frac{2}{9}.$$

Realizamos la transformación elemental de fila:

$$\left[\begin{array}{cccc|c} 1 & 2 & 3 & 5 & 4 \\ 0 & 5 & -10 & -11 & -7 \\ 0 & 0 & 18 & \frac{41}{5} & \frac{37}{5} \\ 0 & 0 & -4 & -\frac{31}{5} & -\frac{27}{5} \end{array}\right] \xrightarrow{f_4 - m_{43} f_3 \to f_4} \left[\begin{array}{cccc|c} 1 & 2 & 3 & 5 & 4 \\ 0 & 5 & -10 & -11 & -7 \\ 0 & 0 & 18 & \frac{41}{5} & \frac{37}{5} \\ 0 & 0 & 0 & -\frac{197}{45} & -\frac{169}{45} \end{array}\right]$$

La última matriz ampliada obtenida está asociada al siguiente sistema triangular superior

$$\begin{cases} x_1 + 2x_2 + 3x_3 + 5x_4 &= 4 \\ 5x_2 - 10x_3 - 11x_4 &= -7 \\ 18x_3 + \frac{41}{5}x_4 &= \frac{37}{5} \\ -\frac{197}{45}x_4 &= -\frac{169}{45} \end{cases}$$

que es equivalente al sistema de ecuaciones lineales dado inicialmente. La solución del sistema reducido se puede obtener usando el procedimiento de Sustitución Regresiva. En este caso, la solución del sistema es: $x = \left(-\frac{277}{197}, \frac{104}{197}, \frac{4}{197}, \frac{169}{197}\right)$.

En la función `gaussiana` que se muestra en el Programa 2.3, implementamos la eliminación gaussiana simple.

Programa 2.3. Eliminación Gaussiana Simple

```
U = np.copy(np.asarray(A, dtype=float))
b = np.copy(np.asarray(b, dtype=float))
n = A.shape[0]
for j in range(0,n-1):
    if U[j,j]==0:
        print('Elemento nulo en la diagonal de A')
        return
    for i in range(j+1,n):
        mu = U[i,j]/U[j,j]
        U[i,0:n] = U[i,0:n] - mu*U[j,0:n]
        b[i] = b[i] - mu*b[j]
return U,b
```

Ejemplo 2.5. Sea $A\mathbf{x} = \mathbf{b}$ un sistema lineal con

$$A = \begin{bmatrix} 1 & 0 & 2 & 0 & 3 \\ 0 & -1 & -1 & 0 & 0 \\ 2 & 0 & 10 & -10 & 0 \\ 0 & 3 & 0 & 8 & 1 \\ 4 & 0 & 8 & 0 & -6 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 5 \\ 4 \\ 3 \\ 2 \\ 1 \end{bmatrix}.$$

Aplicando la función `gaussiana` a la matriz A y al vector $\mathbf{b}$ se tiene:

```
>>> A = np.array([[1,0,2,0,3],
...               [0,-1,-1,0,0],
...               [2,0,10,-10,0],
...               [0,3,0,8,1],
...               [4,0,8,0,-6]])
>>> b = np.array([5,4,3,2,1])
>>> U,bn = gaussiana(A,b)
>>> print(U)
[[ 1.  0.  2.  0.  3.]
 [ 0. -1. -1.  0.  0.]
 [ 0.  0.  6. -10. -6.]
 [ 0.  0.  0.  3. -2.]
 [ 0.  0.  0.  0. -18.]]
>>> print(bn)
[ 5.  4. -7.  10.5 -19. ]
```

De esta forma, se obtiene la matriz U y el vector del lado derecho **bn** del sistema triangular superior $U\mathbf{x} = \mathbf{bn}$, que es equivalente al sistema $A\mathbf{x} = \mathbf{b}$. Ahora, aplicando la función `Sust_Regresiva` a las matrices `U` y `bn` obtenidas previamente, se halla la solución del sistema $A\mathbf{x} = \mathbf{b}$:

```
>>> x = Sust_Regresiva(U,bn)
>>> print(x)
[-11.95679012 -10.89506173   6.89506173   4.2037037    1.05555556]
```

El número de operaciones punto-flotante de la eliminación gaussiana es aproximadamente $\frac{2}{3}n^3$, es decir, es de orden $\mathbf{O}(n^3)$. La combinación de la eliminación gaussiana y la sustitución regresiva se denomina *método de eliminación gaussiana con sustitución regresiva*, cuyo número total de operaciones punto-flotante es $\frac{2}{3}n^3 + n^2$ (la sustitución regresiva realiza n^2 operaciones punto-flotante). Se deja como ejercicio para el lector determinar cuánto tiempo le llevaría a una PC normal hallar la solución de un sistema de ecuaciones de dimensión $n = 10^8$, si utiliza la eliminación gaussiana con sustitución regresiva.

2.2.2 Eliminación Gaussiana con Pivoteo

En caso de que en algún paso de la eliminación gaussiana simple se tenga que el j-ésimo pivote es cero, entonces el proceso se detiene. Aquí es necesario introducir una estrategia de pivoteo, es decir, una técnica que sirve para escoger otro elemento de la columna j que tome el lugar del actual pivote a_{jj}, que se usa para hacer cero los elementos de la columna j-ésima que están debajo de la diagonal principal.

Cuando el pivote $a_{jj} = 0$, la fila j-ésima no puede usarse para hacer cero los elementos de la columna pivote que están debajo de la diagonal principal. En este caso, es necesario hallar una

fila posterior, digamos la k-ésima con $k > j$, en la que $a_{kj} \neq 0$, y luego intercambiarlas (realizar la transformación elemental $f_j \leftrightarrow f_k$) para obtener un pivote no nulo. Este proceso se denomina *pivoteo simple*. La técnica de pivoteo simple es la más elemental y no se usa mucho en la práctica. Pero, ¿por qué es necesario aplicar una técnica de pivoteo? Daremos una respuesta con un ejemplo sencillo.

Ejemplo 2.6. Consideremos el sistema lineal

$$\begin{bmatrix} \delta & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix},$$

donde $0 < \delta \ll 1$ es un número positivo muy pequeño. Al aplicar la Eliminación Gaussiana sin pivoteo, se obtiene el siguiente sistema reducido:

$$\begin{bmatrix} \delta & 1 \\ 0 & 1-\frac{1}{\delta} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2-\frac{1}{\delta} \end{bmatrix},$$

de donde obtenemos la solución, en aritmética exacta,

$$x_2 = \frac{2-\frac{1}{\delta}}{1-\frac{1}{\delta}} = 1 + \frac{\delta}{\delta - 1} \approx 1 - \delta,$$
$$x_1 = \frac{1-x_2}{\delta} = \frac{-1}{\delta - 1} = 1 - \frac{\delta}{\delta - 1} \approx 1 + \delta.$$

Sin embargo usando aritmética de punto-flotante de precisión finita para $\delta = 10^{-20}$, se obtiene $x_1 = 0$ y $x_2 = 1$. En efecto, resolvamos el sistema con `Python`.

```
>>> delta = 1e-20
>>> A,b=np.array([[delta,1],[1,1]]),np.array([1,2])
>>> U,bnew=gaussiana(A, b)
>>> x = Sust_Regresiva(U,bnew)
>>> print(x)
[0. 1.]
```

El ejemplo anterior muestra la inestabilidad de la eliminación gaussiana sin pivoteo y con pivoteo simple. La técnica de pivoteo con mayor uso en la práctica es el *pivoteo parcial*, con lo cual se obtiene estabilidad en el proceso de eliminación gaussiana (ver [6]) y cuyo propósito es usar como pivote el elemento de mayor magnitud de los elementos de la columna pivote que están debajo de la diagonal principal; en otras palabras, si

$$|a_{kj}| = \max\{|a_{jj}|, |a_{j+1,j}|, \ldots, |a_{n-1,j}|, |a_{nj}|\},$$

entonces se intercambia la fila j-ésima con la fila k-ésima. Si en la columna pivote hay más de un elemento con la mayor magnitud, entonces se toma el primero de ellos. En el siguiente ejemplo mostramos el funcionamiento del pivoteo parcial.

Ejemplo 2.7. En este ejemplo usaremos la eliminación gaussiana con pivoteo parcial para hallar un sistema triangular superior equivalente a un sistema de ecuaciones dado. Consideremos el siguiente sistema de ecuaciones:

$$\begin{cases} 2x_1 + x_2 + x_3 & = & 1 \\ 4x_1 + 3x_2 + 3x_3 + x_4 & = & 1 \\ 8x_1 + 7x_2 + 9x_3 + 5x_4 & = & 0 \\ 6x_1 + 7x_2 + 9x_3 + 8x_4 & = & 0 \end{cases}$$

La matriz ampliada de este sistema es:

$$[A|\mathbf{b}] = \left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 1 \\ 4 & 3 & 3 & 1 & 1 \\ 8 & 7 & 9 & 5 & 0 \\ 6 & 7 & 9 & 8 & 0 \end{array}\right].$$

Como $|a_{31}| \geq |a_{i1}|$ para $i = 1,2,3,4$, entonces aplicamos la operación $f_1 \leftrightarrow f_3$ para obtener el primer pivote:

$$\left[\begin{array}{cccc|c} 8 & 7 & 9 & 5 & 0 \\ 4 & 3 & 3 & 1 & 1 \\ 2 & 1 & 1 & 0 & 1 \\ 6 & 7 & 9 & 8 & 0 \end{array}\right].$$

Los multiplicadores que emplearemos para hacer cero los elementos debajo del pivote a_{11} son:

$$m_{21} = \frac{4}{8}, \quad m_{31} = \frac{2}{8}, \quad m_{41} = \frac{6}{8}.$$

Ahora aplicando las operaciones $f_i - m_{i1} f_1 \to f_i$ para $i = 2,3,4$, obtenemos la siguiente matriz ampliada:

$$\left[\begin{array}{cccc|c} 8 & 7 & 9 & 5 & 0 \\ 0 & -1/2 & -3/2 & -3/2 & 1 \\ 0 & -3/4 & -5/4 & -5/4 & 1 \\ 0 & 7/4 & 9/4 & 17/4 & 0 \end{array}\right].$$

Como $|a_{42}| \geq |a_{i2}|$ para $i = 2,3,4$, entonces aplicamos la operación $f_2 \leftrightarrow f_4$ y obtenemos:

$$\left[\begin{array}{cccc|c} 8 & 7 & 9 & 5 & 0 \\ 0 & 7/4 & 9/4 & 17/4 & 0 \\ 0 & -3/4 & -5/4 & -5/4 & 1 \\ 0 & -1/2 & -3/2 & -3/2 & 1 \end{array}\right].$$

Los multiplicadores que emplearemos para hacer cero los elementos debajo del pivote a_{22} son:

$$m_{32} = -\frac{3}{7}, \quad m_{42} = -\frac{2}{7}.$$

Ahora aplicando las operaciones $f_i - m_{i2} f_2 \to f_i$ para $i = 3,4$, obtenemos la siguiente matriz ampliada:

$$\left[\begin{array}{cccc|c} 8 & 7 & 9 & 5 & 0 \\ 0 & 7/4 & 9/4 & 17/4 & 0 \\ 0 & 0 & -2/7 & 4/7 & 1 \\ 0 & 0 & -6/7 & -2/7 & 1 \end{array}\right].$$

Como $|a_{43}| \geq |a_{i3}|$ para $i = 3,4$, entonces aplicamos la operación $f_3 \leftrightarrow f_4$ y obtenemos:

$$\left[\begin{array}{cccc|c} 8 & 7 & 9 & 5 & 0 \\ 0 & 7/4 & 9/4 & 17/4 & 0 \\ 0 & 0 & -6/7 & -2/7 & 1 \\ 0 & 0 & -2/7 & 4/7 & 1 \end{array}\right].$$

El multiplicador que emplearemos para hacer cero el elemento debajo del pivote a_{33} es:

$$m_{43} = \frac{1}{3}.$$

Ahora aplicando la operación $f_4 - m_{42} f_3 \to f_4$ obtenemos la matriz ampliada:

$$\left[\begin{array}{cccc|c} 8 & 7 & 9 & 5 & 0 \\ 0 & 7/4 & 9/4 & 17/4 & 0 \\ 0 & 0 & -6/7 & -2/7 & 1 \\ 0 & 0 & 0 & 2/3 & 2/3 \end{array}\right]$$

Esta última matriz ampliada está asociada al siguiente sistema triangular superior

$$\left\{\begin{array}{rcl} 8x_1 + 7x_2 + 9x_3 + 5x_4 & = & 0 \\ \frac{7}{4}x_2 + \frac{9}{4}x_3 + \frac{17}{4}x_4 & = & 0 \\ -\frac{6}{7}x_3 - \frac{2}{7}x_4 & = & 1 \\ \frac{2}{3}x_4 & = & \frac{2}{3} \end{array}\right.$$

que es equivalente al sistema de ecuaciones lineales dado inicialmente. La solución del sistema reducido se puede obtener usando el procedimiento de sustitución regresiva. En este caso, la solución del sistema es: $\mathbf{x} = \left(\frac{3}{2}, -\frac{1}{2}, -\frac{3}{2}, 1\right)$.

El método *eliminación gaussiana con pivoteo parcial* consiste en utilizar el pivoteo parcial en el proceso de reducción de la eliminación gaussiana. En el Programa 2.4, se muestra la función `gaussiana_pp` donde hemos implementado la eliminación gaussiana con pivoteo parcial. Mostremos la aplicación de las funciones `gaussiana_pp` y `Sust_Regresiva` para resolver un sistema de ecuaciones lineales.

Programa 2.4. Eliminación Gaussiana con Pivoteo Parcial

```
def gaussiana_pp(A,b):
    U = np.copy(np.asarray(A, dtype=float))
    b = np.copy(np.asarray(b, dtype=float))
    n = A.shape[0]
    for j in range(0,n-1):
        k = np.argmax(abs(U[j:n,j]))
        k += j
        if U[k,j]==0:
            print('Pivote cero\n')
            return
        fila_aux = np.copy(U[j,:])
        U[j,:] = U[k,:]
        U[k,:] = fila_aux
        b_aux = np.copy(b[j])
        b[j] = b[k]
        b[k] = b_aux
        for i in range(j+1,n):
            mu = U[i,j]/U[j,j]
            U[i,:] = U[i,:] - mu*U[j,:]
            b[i] = b[i] - mu*b[j]
    return U,b
```

Ejemplo 2.8. Sea $A\mathbf{x} = \mathbf{b}$, donde la matriz A y el vector del lado derecho $\mathbf{b}$ están dados en el Ejemplo 2.5. Aplicando la función `gaussiana_pp` a la matriz A y al vector $\mathbf{b}$ se tiene:

```
>>> U,bn = gaussiana_pp(A,b)
>>> print(U)
[[  4.          0.          8.          0.         -6.        ]
 [  0.          3.          0.          8.          1.        ]
 [  0.          0.          6.        -10.          3.        ]
 [  0.          0.          0.          1.          0.83333333]
 [  0.          0.          0.          0.          4.5       ]]
>>> print(bn)
[ 1.          2.          2.5         5.08333333  4.75      ]
```

Ahora, aplicando la función `Sust_Regresiva` a las matrices `U` y `bn` obtenidas previamente, se halla la solución del sistema $A\mathbf{x} = \mathbf{b}$:

```
>>> x = Sust_Regresiva(U,bn)
>>> print(x)
[-11.95679012 -10.89506173   6.89506173   4.2037037    1.05555556]
```

La combinación de la eliminación gaussiana con pivoteo parcial y la sustitución regresiva se denomina *método de eliminación gaussiana con pivoteo parcial y sustitución regresiva*. Este método requiere el mismo número de operaciones punto-flotante que la eliminación gaussian simple, esto es, $\frac{2}{3}n^3 + n^2$; además, debe realizar n^2 comparaciones para identificar el pivote. Se deja como ejercicio para el lector determinar cuánto tiempo le llevaría a una PC normal hallar la solución de un sistema de ecuaciones de dimensión $n = 10^8$, si utiliza la eliminación gaussiana con pivoteo parcial y sustitución regresiva.

2.3 Factorización LU

2.3.1 Factorización $A = LU$

Sea $A \in \mathbb{R}^{n\times n}$ una matriz cuadrada. La idea es encontrar matrices $L \in \mathbb{R}^{n\times n}$ triangular inferior y $U \in \mathbb{R}^{n\times n}$ triangular superior de manera que

$$A = LU. \tag{2.5}$$

Especificando los valores de la diagonal de las matrices U y L se tienen distintas factorizaciones triangulares, mencionemos dos de ellas:

- Si L es triangular inferior unitaria ($l_{ii} = 1$, $i = 1,2,\ldots,n$), la factorización LU se denomina *factorización de Doólittle*.
- Si la matriz U es triangular superior unitaria ($u_{ii} = 1$, $i = 1,2,\ldots,n$), la factorización LU se denomina *factorización de Crout*.

Nos centraremos en la factorización de Doólittle. También existe otra factorización triangular denominada factorización de Cholesky, que estudiaremos en la Sección 2.4. El procedimiento que utilizaremos para hallar la factorización (2.5) será la eliminación gaussiana. Tal procedimiento se describe en el siguiente ejemplo.

Ejemplo 2.9. Sea A una matriz cuadrada dada por

$$A = \begin{bmatrix} 4 & 3 & -1 \\ -2 & -4 & 5 \\ 1 & 2 & 6 \end{bmatrix}$$

Apliquemos la eliminación gaussiana simple para obtener la factorización (2.5). Inicialmente tomamos $U = A$ y $L = I$, con lo cual la ecuación $A = LU$ es cierta, es decir,

$$\begin{bmatrix} 4 & 3 & -1 \\ -2 & -4 & 5 \\ 1 & 2 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4 & 3 & -1 \\ -2 & -4 & 5 \\ 1 & 2 & 6 \end{bmatrix}.$$

Apliquemos el primer paso de la eliminación gaussiana a la matriz U, esto es, hacer cero los elementos que están debajo del pivote u_{11}. Para ello necesitamos los siguientes multiplicadores:

$$m_{21} = -\frac{1}{2}, \quad m_{31} = \frac{1}{4}.$$

Aplicando las operaciones

$$f_2 - m_{21} f_1 \to f_2 \quad \text{y} \quad f_3 - m_{31} f_1 \to f_3,$$

a la matriz U y colocando

$$l_{21} = m_{21} \quad \text{y} \quad l_{31} = m_{31},$$

obtenemos la siguiente identidad (se deja al lector comprobarla):

$$\begin{bmatrix} 4 & 3 & -1 \\ -2 & -4 & 5 \\ 1 & 2 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ 1/4 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4 & 3 & -1 \\ 0 & -5/2 & 9/2 \\ 0 & 5/4 & 25/4 \end{bmatrix}.$$

Ahora, el multiplicador para hacer cero los elementos debajo del pivote $u_{22} = -5/2$ es:

$$m_{32} = -\frac{1}{2}.$$

Aplicando la operación $f_3 - m_{32} f_2 \to f_3$ a la matriz U y colocando $l_{32} = m_{32}$, obtenemos:

$$\begin{bmatrix} 4 & 3 & -1 \\ -2 & -4 & 5 \\ 1 & 2 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ 1/4 & -1/2 & 1 \end{bmatrix} \begin{bmatrix} 4 & 3 & -1 \\ 0 & -5/2 & 9/2 \\ 0 & 0 & 17/2 \end{bmatrix}.$$

Por lo tanto,

$$L = \begin{bmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ 1/4 & -1/2 & 1 \end{bmatrix} \quad \text{y} \quad U = \begin{bmatrix} 4 & 3 & -1 \\ 0 & -5/2 & 9/2 \\ 0 & 0 & 17/2 \end{bmatrix}.$$

Tomando en cuenta el Ejemplo 2.9, se tiene que la matriz L se calcula empleando los multiplicadores; por ejemplo, para matrices de orden 4 la matriz L tiene la forma

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ m_{21} & 1 & 0 & 0 \\ m_{31} & m_{32} & 1 & 0 \\ m_{41} & m_{42} & m_{43} & 1 \end{bmatrix}.$$

En cambio, la matriz U es la última matriz triangular superior obtenida en la eliminación gaussiana. En el Algoritmo 2.1 se resume el procedimiento para el cálculo de las matrices L y U de la factorización (2.5). En el Programa 2.5 se muestra la implementación del Algoritmo 2.1.

Algoritmo 2.1 (Algoritmo de la Factorización $A = LU$)**.**

1: Sean $U = A$, $L = I$
2: **for** $j = 1 : n-1$ **do**
3: **for** $i = j+1 : n$ **do**
4: $l_{ij} = u_{ij}/u_{jj}$
5: $u_{i,j:n} = u_{i,j:n} - l_{ij}\, u_{i,j:n}$ ▷ Actualización de la fila i de U
6: **end for**
7: **end for**

Programa 2.5. Factorización LU

```
def lu(A):
    U = np.copy(np.asarray(A, dtype=float))
    n,m = U.shape
    if(n!=m):
        print('La matriz A no es cuadrada')
        return False
    L = np.eye(n)
    for j in range(0,n-1):
        if U[j,j]==0:
            print('Elemento nulo en la diagonal de A')
            return False
        for i in range(j+1,n):
            L[i,j] = U[i,j]/U[j,j]
            U[i,0:n] = U[i,0:n] - L[i,j]*U[j,0:n]
    return L,U
```

Cabe preguntar si toda matriz cuadrada tiene factorización LU (2.5). Lamentablemente la respuesta es no; por ejemplo, la matriz

$$A = \begin{bmatrix} 1 & 2 & 6 \\ 4 & 8 & -1 \\ -2 & 3 & 5 \end{bmatrix}$$

no posee factorización (2.5). (Se deja como ejercicio para el lector comprobar esta afirmación. Tenga presente que la factorización $A = LU$ se obtiene usando la eliminación gaussiana sin pivoteo.) Es claro que para la existencia de la factorización LU los pivotes utilizados en la eliminación gaussiana deben ser distintos de cero. A continuación damos un teorema que establece la existencia y unicidad de la factorización LU, cuya demostración se puede ver en [6]. Antes de mostrar el teorema, recordemos el concepto de *menores principales*.

Definición 2.2 (Menores principales)**.** Se define como el k-ésimo menor principal de una matriz A de orden n al determinante de la submatriz principal $k \times k$, denotada por A_k, que consiste de las primeras k filas y de las primeras k columnas de A.

Teorema 2.3 (Existencia y unicidad de la factorización LU)**.**

i) Una matriz A de orden n tiene una factorización LU si y sólo si las submatrices principales A_k, $k = 1, 2, \ldots, n-1$, *son no singulares.*

ii) Si A es no singular y la factorización LU existe, entonces es única.

Veamos un ejemplo del uso del Teorema 2.3.

Ejemplo 2.10. Verifiquemos que la matriz

$$A = \begin{bmatrix} 1 & 2 & 6 \\ 4 & 8 & -1 \\ -2 & 3 & 5 \end{bmatrix}$$

no tiene factorización LU. Las submatrices principales de A son:

$$A_1 = \begin{bmatrix} 1 \end{bmatrix}, \quad A_2 = \begin{bmatrix} 1 & 2 \\ 4 & 8 \end{bmatrix}.$$

Ahora, los menores principales son:

$$\det(A_1) = 1, \qquad \det(A_2) = 0 \quad \Rightarrow \quad A_2 \text{ es singular.}$$

Por el Teorema 2.3, la matriz A no tiene factorización LU.

■

El uso del Teorema 2.3 es estrictamente teórico, ya que es necesario verificar que las $(n-1)$ submatrices principales son no singulares, para lo cual se deben calcular respectivamente, por ejemplo, determinantes de orden $1, 2, \ldots, n-1$, que podría ser sumamente costoso cuando n es grande.

El comando `lu` del módulo `linalg` de la librería `Scipy`, se puede utilizar para hallar la factorización LU (2.5). En el siguiente ejemplo comparamos la función `lu` del Programa 2.5 y el comando `lu` de `Scipy`.

Ejemplo 2.11. Sea A la matriz cuadrada del Ejemplo 2.9. Entonces, las matrices L y U de la factorización (2.5) halladas con el comando `lu` de `Scipy` son:

```
>>> import scipy.linalg as LA
>>> A = np.array([[4,3,-1],[-2,-4,5],[1,2,6]])
>>> L, U = LA.lu(A, True)

>>> print(L)
[[ 1.    0.    0.  ]
 [-0.5   1.    0.  ]
 [ 0.25 -0.5   1.  ]]
>>> print(U)
[[ 4.   3.  -1. ]
 [ 0.  -2.5  4.5]
 [ 0.   0.   8.5]]
```

Ahora, las matrices L y U de la factorización (2.5) halladas con la función `lu` del Programa 2.5 son:

```
>>> L, U = lu(A)
>>> print(L)
[[ 1.    0.    0.  ]
 [-0.5   1.    0.  ]
 [ 0.25 -0.5   1.  ]]
>>> print(U)
[[ 4.   3.  -1. ]
 [ 0.  -2.5  4.5]
 [ 0.   0.   8.5]]
```

Ambas funciones hallan la misma factorización LU. Veamos ahora el comportamiento de estas funciones en una matriz que no tiene factorización LU. Consideremos la matriz

$$A = \begin{bmatrix} 1 & 2 & 6 \\ 4 & 8 & -1 \\ -2 & 3 & 5 \end{bmatrix}$$

que no tiene factorización LU. En este caso, las matrices L y U de la factorización (2.5) halladas con el comando `lu` de `Scipy` son:

```
>>> A = np.array([[1,2,6],[4,8,-1],[-2,3,5]])
>>> L, U = scl.lu(A, True)
>>> print(L)
[[ 0.25  0.    1.  ]
 [ 1.    0.    0.  ]
 [-0.5   1.    0.  ]]
>>> print(U)
[[ 4.    8.   -1.  ]
 [ 0.    7.    4.5 ]
 [ 0.    0.    6.25]]
```

Las matrices L y U de la factorización (2.5) halladas con la función `lu` del Programa 2.5 son:

```
>>> L, U = lu(A)
Elemento nulo en la diagonal de A
```

Note que el comando `lu` de `Scipy` halla matrices L y U, aunque la matriz L hallada no es una matriz triangular inferior. En cambio, la función `lu` del Programa 2.5 no halla ninguna matriz, ya que la matriz A no tiene factorización LU (2.5).

Hay matrices especiales para las que siempre existe la factorización LU (2.5). Una clase de matrices que aparecen en muchas aplicaciones y que siempre poseen factorización LU (2.5), es la formada por las matrices *estrictamente diagonal dominantes*.

Definición 2.3 (Matriz estrictamente diagonal dominante)**.** Una matriz $A \in \mathbb{R}^{n\times n}$ cuadrada se dice estrictamente diagonal dominante si se cumple la siguiente desigualdad

$$|a_{jj}| > \sum_{\substack{i=1 \\ i\neq j}}^{n} |a_{ij}| \quad j = 1,2,\ldots,n.$$

Teorema 2.4. *Toda matriz estrictamente diagonal dominante $A \in \mathbb{R}^{n\times n}$ es no singular y tiene una factorización $A = LU$, donde $L \in \mathbb{R}^{n\times n}$ es una matriz triangular inferior unitaria y $U \in \mathbb{R}^{n\times n}$ es una matriz triangular superior.*

Como una matriz cuadrada A, en general, no siempre tiene factorización LU (2.5), entonces es válido preguntarse: ¿A tendrá otro tipo de factorización que involucre matrices triangulares inferiores y superiores? Afortunadamente, sí existen tales factorizaciones. A continuación describiremos una factorización de A, que involucra matrices triangulares y que hallaremos usando la eliminación gaussiana con pivoteo parcial.

2.3.2 Factorización $PA = LU$

Dada una matriz $A \in \mathbb{R}^{n\times n}$ cuadrada. Pasemos ahora a encontrar matrices $P \in \mathbb{R}^{n\times n}$, $L \in \mathbb{R}^{n\times n}$ y $U \in \mathbb{R}^{n\times n}$ tales que

$$PA = LU, \tag{2.6}$$

donde P es una matriz de permutación, L es una matriz triangular inferior unitaria y U es una matriz triangular superior. Usaremos un ejemplo para explicar el uso de la eliminación gaussiana con pivoteo parcial en el cálculo de las matrices P, L y U.

Ejemplo 2.12. Hallemos la factorización (2.6) de la matriz

$$A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}.$$

Inicialmente tomamos $P = I$, $U = A$ y $L = I$. Por tanto, la ecuación $PA = LU$ es cierta. La idea es aplicar la eliminación gaussiana con pivoteo parcial a la matriz U conservando la identidad $PA = LU$. Tenemos:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}$$
$$\overset{P}{\downarrow} \qquad\qquad \overset{A}{\downarrow} \qquad\qquad \overset{L}{\downarrow} \qquad\qquad \overset{U}{\downarrow}$$

Como $|u_{31}| \geq |u_{r1}|$ para $r = 1,2,3,4$, entonces aplicamos la operación $f_1 \leftrightarrow f_3$ en las matrices P y U. En este primer paso no se altera la matriz L. Así, obtenemos la ecuación (cuya comprobación se deja para el lector):

$$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 8 & 7 & 9 & 5 \\ 4 & 3 & 3 & 1 \\ 2 & 1 & 1 & 0 \\ 6 & 7 & 9 & 8 \end{bmatrix}$$
$$\overset{P}{\downarrow} \qquad\qquad \overset{A}{\downarrow} \qquad\qquad \overset{L}{\downarrow} \qquad\qquad \overset{U}{\downarrow}$$

Ahora pasamos hacer cero los elementos debajo del pivote u_{11}. Para ello utilizamos los multiplicadores:

$$m_{21} = \frac{1}{2}, \quad m_{31} = \frac{1}{4}, \quad m_{41} = \frac{3}{4}.$$

Aplicando las operaciones

$$f_k - m_{k1} f_1 \to f_k \quad \text{para } k = 2,3,4$$

a la matriz U y colocando $l_{k1} = m_{k1}$ para $k = 2,3,4$, obtenemos:

$$\overset{P}{\underset{\downarrow}{\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}} \overset{A}{\underset{\downarrow}{\begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}}} = \overset{L}{\underset{\downarrow}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{1}{2} & 1 & 0 & 0 \\ \frac{1}{4} & 0 & 1 & 0 \\ \frac{3}{4} & 0 & 0 & 1 \end{bmatrix}}} \overset{U}{\underset{\downarrow}{\begin{bmatrix} 8 & 7 & 9 & 5 \\ 0 & -\frac{1}{2} & -\frac{3}{2} & -\frac{3}{2} \\ 0 & -\frac{3}{4} & -\frac{5}{4} & -\frac{5}{4} \\ 0 & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \end{bmatrix}}}$$

Como $|u_{42}| \geq |u_{r2}|$ para $r = 2,3,4$, entonces aplicamos la operación $f_2 \leftrightarrow f_4$ a las matrices P y U. En la matriz L se intercambian los elementos $l_{21} \leftrightarrow l_{41}$. Así, obtenemos:

$$\overset{P}{\underset{\downarrow}{\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}}} \overset{A}{\underset{\downarrow}{\begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}}} = \overset{L}{\underset{\downarrow}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{3}{4} & 1 & 0 & 0 \\ \frac{1}{4} & 0 & 1 & 0 \\ \frac{1}{2} & 0 & 0 & 1 \end{bmatrix}}} \overset{U}{\underset{\downarrow}{\begin{bmatrix} 8 & 7 & 9 & 5 \\ 0 & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ 0 & -\frac{3}{4} & -\frac{5}{4} & -\frac{5}{4} \\ 0 & -\frac{1}{2} & -\frac{3}{2} & -\frac{3}{2} \end{bmatrix}}}$$

Los multiplicadores para hacer cero los elementos debajo del pivote u_{22} son:

$$m_{32} = -\frac{3}{7}, \quad m_{42} = -\frac{2}{7}.$$

Aplicando las operaciones

$$f_k - m_{k2} f_2 \to f_k \quad \text{para } k = 3,4$$

a la matriz U y colocando $l_{k2} = m_{k2}$ para $k = 3,4$, obtenemos:

$$\overset{P}{\underset{\downarrow}{\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}}} \overset{A}{\underset{\downarrow}{\begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}}} = \overset{L}{\underset{\downarrow}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{3}{4} & 1 & 0 & 0 \\ \frac{1}{4} & -\frac{3}{7} & 1 & 0 \\ \frac{1}{2} & -\frac{2}{7} & 0 & 1 \end{bmatrix}}} \overset{U}{\underset{\downarrow}{\begin{bmatrix} 8 & 7 & 9 & 5 \\ 0 & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ 0 & 0 & -\frac{2}{7} & \frac{4}{7} \\ 0 & 0 & -\frac{6}{7} & -\frac{2}{7} \end{bmatrix}}}$$

Como $|u_{43}| \geq |u_{r3}|$ para $r = 3,4$, entonces aplicamos la operación $f_3 \leftrightarrow f_4$ a las matrices P y U. En la matriz L se intercambian los elementos $l_{31} \leftrightarrow l_{41}$ y $l_{32} \leftrightarrow l_{42}$. Así, obtenemos:

$$\overset{P}{\underset{\downarrow}{\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}}} \overset{A}{\underset{\downarrow}{\begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}}} = \overset{L}{\underset{\downarrow}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{3}{4} & 1 & 0 & 0 \\ \frac{1}{2} & -\frac{2}{7} & 1 & 0 \\ \frac{1}{4} & -\frac{3}{7} & 0 & 1 \end{bmatrix}}} \overset{U}{\underset{\downarrow}{\begin{bmatrix} 8 & 7 & 9 & 5 \\ 0 & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ 0 & 0 & -\frac{6}{7} & -\frac{2}{7} \\ 0 & 0 & -\frac{2}{7} & \frac{4}{7} \end{bmatrix}}}$$

El multiplicador para hacer cero los elementos debajo del pivote u_{33} es:

$$m_{43} = \frac{1}{3}.$$

Aplicando la operación

$$f_4 - m_{43} f_3 \to f_4$$

a la matriz U y colocando $l_{43} = m_{43}$, obtenemos:

$$\overset{P}{\downarrow}\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \overset{A}{\downarrow}\begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix} = \overset{L}{\downarrow}\begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{3}{4} & 1 & 0 & 0 \\ \frac{1}{2} & -\frac{2}{7} & 1 & 0 \\ \frac{1}{4} & -\frac{3}{7} & \frac{1}{3} & 1 \end{bmatrix} \overset{U}{\downarrow}\begin{bmatrix} 8 & 7 & 9 & 5 \\ 0 & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ 0 & 0 & -\frac{6}{7} & -\frac{2}{7} \\ 0 & 0 & 0 & \frac{2}{3} \end{bmatrix}$$

De esta forma,

$$P = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}, \quad L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{3}{4} & 1 & 0 & 0 \\ \frac{1}{2} & -\frac{2}{7} & 1 & 0 \\ \frac{1}{4} & -\frac{3}{7} & \frac{1}{3} & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 8 & 7 & 9 & 5 \\ 0 & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ 0 & 0 & -\frac{6}{7} & -\frac{2}{7} \\ 0 & 0 & 0 & \frac{2}{3} \end{bmatrix}.$$

En general, una matriz cuadrada A, singular o no singular, tiene una factorización (2.6), donde P es una matriz de permutación, L es una matriz unitaria triangular inferior y U es un matriz triangular superior. El pivoteo parcial es una práctica tan universal que esta factorización se conoce usualmente como *factorización LU* de A.

La famosa fórmula (2.6) tiene una interpretación simple. La eliminación gaussiana con pivoteo parcial es equivalente al siguiente procedimiento:

1. Permutar las columnas de A acorde a la matriz P.

2. Aplicar la eliminación gaussiana simple a la matriz PA.

El pivoteo parcial no se lleva a cabo de esta manera en la práctica, por supuesto, ya que P no se conoce de antemano. En el Algoritmo 2.2 se resume el procedimiento para el cálculo de las matrices P, L y U de la factorización (2.6), cuya implementación se muestra en el Programa 2.6.

Algoritmo 2.2 (Algoritmo de la Factorización $PA = LU$)**.**

Sean $U = A$, $L = I$, $P = I$
for $j = 1 : n-1$ **do**
  Seleccione $k \geq j$ que maximice $|u_{kj}|$
  $u_{j,j:n} \leftrightarrow u_{k,j:n}$ ▷ Intercambio de las filas j y k de U
  $l_{j,1:j-1} \leftrightarrow l_{k,1:j-1}$ ▷ Intercambio de los elementos de L
  $p_{j,:} \leftrightarrow p_{k,:}$ ▷ Intercambio de las filas j y k de P
  for $i = j+1 : n$ **do**
    $l_{ij} = u_{ij}/u_{jj}$ ▷ Multiplicador m_{ij}
    $u_{i,j:n} = u_{i,j:n} - l_{ij}\, u_{i,j:n}$
  end for
end for

El número de operaciones punto-flotante para obtener la factorización $PA = LU$ es $\frac{2}{3}n^3$ y se realizan $\frac{1}{2}n(n-1)$ comparaciones para escoger los pivotes. El comando `lu` del módulo `linalg` de la librería `Scipy`, también se puede usar para hallar las matrices P, L y U de la factorización (2.6). En este caso, se utiliza el comando `lu(A, False)`. Observe el siguiente ejemplo donde comparamos `lu` con `plu`.

Programa 2.6. Factorización $PA = LU$

```
def plu(A):
    U = np.copy(np.asarray(A, dtype=float))
    n, m = U.shape
    if n != m:
        print('La matriz A no es cuadrada')
        return False
    L,P = np.eye(n),np.eye(n)
    for j in range(0,n-1):
        k = np.argmax(abs(U[j:n,j]))
        k += j
        if U[k,j]==0:
            print('Pivote cero')
            return False
        fila_aux = np.copy(U[j,j:n])
        U[j,j:n] = U[k,j:n]
        U[k,j:n] = fila_aux
        p_aux = np.copy(P[j,:])
        P[j,:] = P[k,:]
        P[k,:] = p_aux
        if j>=1:
            l_aux = np.copy(L[j,0:j])
            L[j,0:j] = L[k,0:j]
            L[k,0:j] = l_aux
        for i in range(j+1,n):
            L[i,j] = U[i,j]/U[j,j]
            U[i,:] = U[i,:] - L[i,j]*U[j,:]
    return P,L,U
```

Ejemplo 2.13. Sea A la matriz cuadrada del Ejemplo 2.12. Entonces, las matrices P, L y U de la factorización (2.5) halladas con `lu` son:

```
>>> import scipy.linalg as LA
>>> A = np.array([[2,1,1,0],[4,3,3,1],[8,7,9,5],[6,7,9,8]])
>>> P, L, U = LA.lu(A, False)
>>> print(L)
[[ 1.          0.          0.          0.        ]
 [ 0.75        1.          0.          0.        ]
 [ 0.5        -0.28571429  1.          0.        ]
 [ 0.25       -0.42857143  0.33333333  1.        ]]
>>> print(U)
[[ 8.          7.          9.          5.        ]
 [ 0.          1.75        2.25        4.25      ]
 [ 0.          0.         -0.85714286 -0.28571429]
 [ 0.          0.          0.          0.66666667]]
>>> print(P)
[[ 0.  0.  0.  1.]
 [ 0.  0.  1.  0.]
```

```
 [ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]]
```

Ahora, las matrices P, L y U de la factorización (2.5) halladas con `plu` son:

```
>>> P,L,U = plu(A)
>>> print(L)
[[ 1.          0.          0.          0.        ]
 [ 0.75        1.          0.          0.        ]
 [ 0.5        -0.28571429  1.          0.        ]
 [ 0.25       -0.42857143  0.33333333  1.        ]]
>>> print(U)
[[ 8.          7.          9.          5.        ]
 [ 0.          1.75        2.25        4.25      ]
 [ 0.          0.         -0.85714286 -0.28571429]
 [ 0.          0.          0.          0.66666667]]
>>> print(P)
[[ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]
 [ 0.  1.  0.  0.]
 [ 1.  0.  0.  0.]]
```

2.3.3 Solución de un Sistema Lineal usando LU

Supongamos que la matriz A del sistema lineal $A\mathbf{x} = \mathbf{b}$ admite la factorización (2.5). Entonces, el sistema $A\mathbf{x} = \mathbf{b}$ se puede expresar como $LU\mathbf{x} = \mathbf{b}$. De esta forma, la solución de éste último sistema se puede obtener definiendo $\mathbf{y} = U\mathbf{x}$ y resolviendo secuencialmente los siguientes dos sistemas lineales:

$$L\mathbf{y} = \mathbf{b} \quad \text{y} \quad U\mathbf{x} = \mathbf{y}.$$

El primer sistema triangular inferior, $L\mathbf{y} = \mathbf{b}$, se resuelve usando el método de sustitución progresiva. Una vez obtenido $\mathbf{y}$, el segundo sistema triangular superior, $U\mathbf{x} = \mathbf{y}$, se resuelve usando el método de sustitución regresiva. Mostremos con un ejemplo el procedimiento descrito.

Ejemplo 2.14. Sea el sistema lineal

$$\begin{cases} x_1 + 2x_2 + 4x_3 + x_4 &= 21 \\ 2x_1 + 8x_2 + 6x_3 + 4x_4 &= 52 \\ 3x_1 + 10x_2 + 8x_3 + 8x_4 &= 79 \\ 4x_1 + 12x_2 + 10x_3 + 6x_4 &= 82 \end{cases}$$

Usando la eliminación gaussiana simple obtenemos la factorización (2.5) de A:

$$A = \begin{bmatrix} 1 & 2 & 4 & 1 \\ 2 & 8 & 6 & 4 \\ 3 & 10 & 8 & 8 \\ 4 & 12 & 10 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 3 & 1 & 1 & 0 \\ 4 & 1 & 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 4 & 1 \\ 0 & 4 & -2 & 2 \\ 0 & 0 & -2 & 3 \\ 0 & 0 & 0 & -6 \end{bmatrix} = LU.$$

Resolvamos el primer sistema $L\mathbf{y} = \mathbf{b}$. Como este sistema es triangular inferior, podemos utilizar la función `Sust_Progresiva` para resolverlo:

```
>>> A = np.array([[1, 2, 4, 1],[2, 8, 6, 4],[3, 10, 8, 8],[4, 12, 10, 6]])
>>> b = np.array([21,52,79,82])
>>> L,U = lu(A)
>>> print(L)
[[ 1.  0.  0.  0.]
 [ 2.  1.  0.  0.]
 [ 3.  1.  1.  0.]
 [ 4.  1.  2.  1.]]
>>> print(U)
[[ 1.  2.  4.  1.]
 [ 0.  4. -2.  2.]
 [ 0.  0. -2.  3.]
 [ 0.  0.  0. -6.]]
>>> y = Sust_Progresiva(L,b)
>>> print(y)
[ 21.  10.   6. -24.]
```

Luego, la solución del sistema $L\mathbf{y} = \mathbf{b}$ es: $\mathbf{y} = (21, 10, 6, -24)^T$. Ahora resolvamos el sistema $U\mathbf{x} = \mathbf{y}$, por supuesto, lo haremos con la función `Sust_Regresiva`:

```
>>> x = Sust_Regresiva(U,y)
>>> print(x)
[ 1.  2.  3.  4.]
```

Así, la solución del sistema inicial es: $\mathbf{x} = (1, 2, 3, 4)$.

Supongamos ahora que la matriz A no admite la factorización (2.5), sino que admite la factorización (2.6), es decir, $PA = LU$. Así, $A = P^{-1}LU$ y el sistema $A\mathbf{x} = \mathbf{b}$ adquiere la forma $P^{-1}LU\mathbf{x} = \mathbf{b}$, que es equivalente al sistema $LU\mathbf{x} = P\mathbf{b}$. En este caso, la solución del sistema $LU\mathbf{x} = P\mathbf{b}$ se obtiene definiendo $\mathbf{y} = U\mathbf{x}$ y resolviendo secuencialmente los siguientes sistemas lineales:

$$L\mathbf{y} = P\mathbf{b} \quad \text{y} \quad U\mathbf{x} = \mathbf{y}.$$

Análogamente, el primer sistema triangular inferior, $L\mathbf{y} = P\mathbf{b}$, se resuelve usando el método de sustitución progresiva. Una vez obtenido $\mathbf{y}$, el segundo sistema triangular superior, $U\mathbf{x} = \mathbf{y}$, se resuelve usando el método de sustitución regresiva. Mostremos un ejemplo.

Ejemplo 2.15. Sea el sistema $A\mathbf{x} = \mathbf{b}$ del Ejemplo 2.14. La matriz A admite la factorización (2.6), esto es:

$$PA = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 2 & 4 & 1 \\ 2 & 8 & 6 & 4 \\ 3 & 10 & 8 & 8 \\ 4 & 12 & 10 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1/2 & 1 & 0 & 0 \\ 1/4 & -1/2 & 1 & 0 \\ 3/4 & 1/2 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4 & 12 & 10 & 6 \\ 0 & 2 & 1 & 1 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 3 \end{bmatrix} = LU.$$

Resolvamos el primer sistema $L\mathbf{y} = P\mathbf{b}$, utilizando la función `Sust_Progresiva`:

```
>>> A = np.array([[1, 2, 4, 1],[2, 8, 6, 4],[3, 10, 8, 8],[4, 12, 10, 6]])
>>> b = np.array([21,52,79,82])
>>> P,L,U = plu(A)
```

```
>>> print(L)
[[ 1.    0.    0.    0.  ]
 [ 0.5   1.    0.    0.  ]
 [ 0.25 -0.5   1.    0.  ]
 [ 0.75  0.5   0.    1.  ]]
>>> print(U)
[[  4.  12.  10.   6.]
 [  0.   2.   1.   1.]
 [  0.   0.   2.   0.]
 [  0.   0.   0.   3.]]
>>> print(P)
[[ 0.  0.  0.  1.]
 [ 0.  1.  0.  0.]
 [ 1.  0.  0.  0.]
 [ 0.  0.  1.  0.]]
>>> y = Sust_Progresiva(L,P.dot(b))
>>> print(y)
[ 82.  11.   6.  12.]
```

Ahora resolvamos el sistema $U\mathbf{x} = \mathbf{y}$, utilizando la función `Sust_Regresiva`:

```
>>> x = Sust_Regresiva(U,y)
>>> print(x)
[ 1.  2.  3.  4.]
```

2.3.4 Cálculo de la Inversa y el Determinante usando LU

Pasemos ahora a calcular el determinante y la inversa de una matriz A que admite una de las factorizaciones (2.5) o (2.6). Note que el determinante de una matriz triangular (inferior o superior) es el producto de los elementos de su diagonal principal. De esta forma, si A admite algunas de las factorizaciones triangulares, entonces el determinante o la inversa se calculan de la siguiente forma:

- Si la matriz A admite la factorización (2.5), esto es, $A = LU$, entonces el determinante de A está dado por

$$\det(A) = \det(L)\det(U) = \det(U) = \prod_{i=1}^{n} u_{ii}$$

  y la inversa de A es

$$A^{-1} = U^{-1}L^{-1}.$$

- Si la matriz A admite la factorización (2.6), esto es, $PA = LU$, entonces el determinante de A está dado por

$$\det(A) = \frac{\det(L)\det(U)}{\det(P)} = \det(U)\det(P)^{-1} = \det(P)^{-1}\prod_{i=1}^{n} u_{ii},$$

  donde $\det(P) = (-1)^r$ y r es el número de permutaciones realizadas en la matriz identidad para obtener la matriz P. Ahora, la inversa de A es

$$A^{-1} = U^{-1}L^{-1}P.$$

Ejemplo 2.16. Calculemos el determinante y la inversa de la matriz

$$A = \begin{bmatrix} 1 & 2 & 4 & 1 \\ 2 & 8 & 6 & 4 \\ 3 & 10 & 8 & 8 \\ 4 & 12 & 10 & 6 \end{bmatrix}.$$

La matriz A admite factorización $A = LU$, donde

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 3 & 1 & 1 & 0 \\ 4 & 1 & 2 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 1 & 2 & 4 & 1 \\ 0 & 4 & -2 & 2 \\ 0 & 0 & -2 & 3 \\ 0 & 0 & 0 & -6 \end{bmatrix}.$$

Así, el determinante de A es

$$\det(A) = \det(U) = 1 \cdot 4 \cdot (-2) \cdot (-6) = 48.$$

Ahora, calculemos la inversa de las matrices L y U utilizando el comando `inv` del módulo `linalg` de la librería `numpy`, luego calcularemos $A^{-1} = U^{-1}L^{-1}$.

```
>>> from numpy.linalg import inv
>>> A = np.array([[1, 2, 4, 1],[2, 8, 6, 4],[3, 10, 8, 8],[4, 12, 10, 6]])
>>> L,U = lu(A)
>>> print(L)
[[ 1.  0.  0.  0.]
 [ 2.  1.  0.  0.]
 [ 3.  1.  1.  0.]
 [ 4.  1.  2.  1.]]
>>> print(U)
[[ 1.  2.  4.  1.]
 [ 0.  4. -2.  2.]
 [ 0.  0. -2.  3.]
 [ 0.  0.  0. -6.]]
>>> B = inv(U).dot(inv(L))
>>> print(B)
[[-0.5         -1.75        0.          1.25      ]
 [-0.25         0.45833333 -0.16666667 -0.04166667]
 [ 0.5          0.25        0.         -0.25      ]
 [ 0.          -0.16666667  0.33333333 -0.16666667]]
```

En otras palabras, la inversa de A es

$$A^{-1} = \begin{bmatrix} -\frac{1}{2} & -\frac{7}{4} & 0 & \frac{5}{4} \\ -\frac{1}{4} & \frac{11}{24} & -\frac{1}{6} & -\frac{1}{24} \\ \frac{1}{2} & \frac{1}{4} & 0 & -\frac{1}{4} \\ 0 & -\frac{1}{6} & \frac{1}{3} & -\frac{1}{6} \end{bmatrix}.$$

2.4 Factorización de Cholesky

La factorización de Cholesky es un caso particular de una factorización triangular que se aplica a matrices simétricas y definidas positivas. Recordemos que una matriz A es definida positiva si para todo vector distinto de cero $\mathbf{x}$ se cumple $\mathbf{x}^T A\mathbf{x} > 0$. Ahora, si $\mathbf{x}^T A\mathbf{x} \geq 0$ para todo vector $\mathbf{x}$, la matriz se dice semidefinida positiva. Cuando una matriz A es simétrica y definida positiva (semidefinida positiva), diremos que A es *simétrica definida positiva* (*simétrica semidefinida positiva*). Algunas de las propiedades de las matrices definidas positivas son las siguientes:

i) Una matriz simétrica A es definida positiva si y sólo si todos sus autovalores son reales y positivos.

ii) Si $A = (a_{ij})$ es simétrica definida positiva, entonces $a_{ii} > 0$ para todo i.

iii) Una matriz simétrica A es definida positiva si y sólo si todos sus menores principales son positivos.

iv) Si $A = (a_{ij})$ es simétrica definida positiva, entonces el elemento más grande en módulo en toda la matriz se encuentra en la diagonal.

v) La suma de dos matrices simétricas definidas positivas es una matriz simétrica definida positiva.

vi) La suma de todas las entradas (sin módulos) de una matriz simétrica definida positiva es un número positivo.

Es de notar que las propiedades ii), iv) y vi) son condiciones necesarias para que una matriz sea simétrica definida positiva. Tales condiciones permiten descartar matrices que no son definidas positivas. Por ejemplo, las matrices

$$A = \begin{bmatrix} 3 & 0 & 4 & 7 & 9 \\ 0 & 7 & 0 & 6 & -1 \\ 4 & 0 & -2 & 0 & 1 \\ 7 & 6 & 0 & 7 & 9 \\ 9 & -1 & 1 & 9 & 9 \end{bmatrix}, \qquad B = \begin{bmatrix} 5 & 88 & 0 \\ 88 & 6 & 2 \\ 0 & 2 & 10 \end{bmatrix}, \qquad C = \begin{bmatrix} 3 & -2 & 1 \\ -2 & 1 & -2 \\ 1 & -2 & 1 \end{bmatrix},$$

no pueden ser definidas positivas, ya que A tiene un elemento negativo en la diagonal, el elemento más grande en módulo de la matriz B no está en la diagonal y la suma de todas las entradas de la matriz C no es un número positivo.

Volvamos a la factorización de Cholesky. Dada una matriz $A \in \mathbb{R}^{n\times n}$ simétrica definida positiva, el objetivo de la factorización de Cholesky es hallar una matriz $R = (r_{ij}) \in \mathbb{R}^{n\times n}$ triangular inferior tal que

$$A = RR^T, \quad r_{ii} > 0. \tag{2.7}$$

En el siguiente teorema se establece que toda matriz simétrica definida positiva tiene una única factorización de Cholesky.

Teorema 2.5. *Toda matriz simétrica definida positiva $A \in \mathbb{R}^{n\times n}$ tiene una única factorización de Cholesky, esto es, $A = RR^T$ donde $R = (r_{ij})$ es triangular inferior. Los elementos de R se calculan como sigue: $r_{11} = \sqrt{a_{11}}$ y, para $i = 2,\ldots,n$,*

$$r_{ii} = \left(a_{ii} - \sum_{k=1}^{i-1} r_{ki}^2 \right)^{1/2}, \quad r_{ij} = \frac{a_{ij} - \sum_{k=1}^{j-1} r_{ik} r_{jk}}{r_{jj}}, \quad j = i+1,\ldots,n.$$

En el teorema anterior, $\sum_{k=1}^{0}() \equiv 0$.

Algoritmo 2.3 (Algoritmo de la Factorización de Cholesky)**.**

1: Sea $A \in \mathbb{R}^{n\times n}$ simétrica definida positiva.
2: $R = 0_{n\times n}$
3: **for** $j = 1 : n$ **do**
4: $\quad r_{jj} = \sqrt{a_{ij} - \sum_{k=1}^{j-1} r_{kj}^2}$
5: $\quad$ **for** $i = j+1 : n$ **do**
6: $\quad\quad r_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} r_{ik} r_{jk}\right) / r_{jj}$
7: $\quad$ **end for**
8: **end for**

Programa 2.7. Factorización de Cholesky

```
def chol(A):
    A = np.asarray(A, dtype=float)
    n,m = A.shape
    if n!=m:
        print('La matriz A no es cuadrada')
        return
    if np.linalg.norm(A-A.T,1) > 1.0e-6:
        print('La matriz A no es simétrica')
        return
    R = np.zeros_like(A)
    for j in range(n):
        temp = A[j,j]-np.sum(R[j,0:j]**2)
        if temp <= 0.0:
            print('La matriz no es definida positiva')
            return
        R[j,j] = np.sqrt(temp)
        for i in range(j+1,n):
            R[i,j] = (A[i,j]-np.sum(R[i,0:j]*R[j,0:j]))/R[j,j]
    return R
```

El Algoritmo 2.3 es una de las muchas implementaciones de la factorización de Cholesky. El número de operaciones punto flotante que se realizan en la factorización de Cholesky es aproximadamente $n^3/3$, es decir, es de orden $\mathbf{O}(n^3)$. En el Programa 2.7 se implementa la factorización de Cholesky. Una de las formas mas efectivas computacionalmente de verificar si una matriz dada es definida positiva es usar la factorización de Cholesky y ver si el algoritmo termina satisfactoriamente.

El comando `cholesky` del módulo `linalg` de la librería `numpy`, permite calcular la factorización de Cholesky. La función `R=cholesky(A)` arroja uno de los siguientes resultados:

- una matriz R triangular inferior tal que $A = RR^T$, si A es simétrica definida positiva;
- imprime un mensaje de error, si A no es simétrica definida positiva.

Veamos un ejemplo.

Ejemplo 2.17. Sean las siguientes tres matrices:

$$A = \begin{bmatrix} 4 & 2 & 6 \\ -2 & 2 & 5 \\ 6 & 5 & 29 \end{bmatrix}, \quad B = \begin{bmatrix} 4 & 4 & 8 \\ 4 & -4 & 1 \\ 8 & 1 & 9 \end{bmatrix}, \quad C = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 2 & 3 \end{bmatrix}.$$

Primero carguemos las matrices A, B y C, y la función `cholesky`.

```
>>> import numpy as np
>>> from numpy.linalg import cholesky
>>> A = np.array([[4,2,6],[-2,2,5],[6,5,29]])
>>> B = np.array([[4,4,8],[4,-4,1],[8,1,9]])
>>> C = np.array([[1,1,1],[1,2,2],[1,2,3]])
```

Al ejecutar el comando `cholesky` con la matriz A se obtiene:

```
>>> R = cholesky(A)
...
numpy.linalg.linalg.LinAlgError: Matrix is not positive definite
```

En este caso, `cholesky` nos indica que la matriz A no es simétrica definida positiva. Al ejecutar `chol` se obtiene:

```
>>> R = chol(A)
La matriz no es simétrica
```

Ejecutamos `cholesky` con la matriz B:

```
>>> R = cholesky(B)
    raise LinAlgError("Matrix is not positive definite")
numpy.linalg.linalg.LinAlgError: Matrix is not positive definite
```

La matriz B tampoco es definida positiva, aunque ella sea simétrica. Al ejecutar el comando `chol` con la matriz B obtenemos:

```
>>> R = chol(B)
La matriz no es definida positiva
```

Finalmente, para la matriz C ejecutamos `cholesky`:

```
>>> R = cholesky(C)
>>> print(R)
[[ 1.  0.  0.]
 [ 1.  1.  0.]
 [ 1.  1.  1.]]
```

También ejecutamos el comando `chol`:

```
>>> R = chol(C)
>>> print(R)
[[ 1.  0.  0.]
 [ 1.  1.  0.]
 [ 1.  1.  1.]]
```

La matriz R obtenida con `cholesky` satisface que $C = RR^T$, en otras palabras, `cholesky` cal-culó la factorización de Cholesky (2.7) de C. El comando `chol` también arroja la matriz R. De esta forma, C es una matriz simétrica definida positiva, cuya factorización de Cholesky es $C = RR^L$, con

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

2.4.1 Solución de un Sistema Lineal usando Cholesky

Si A es simétrica definida positiva, la forma estándar de resolver el sistema de ecuaciones $A\mathbf{x} = \mathbf{b}$ (de dimensión moderada) es usando la factorización de Cholesky. En este caso, como $A = RR^T$ con R triangular inferior, el sistema $A\mathbf{x} = \mathbf{b}$ adquiere la forma $RR^T\mathbf{x} = \mathbf{b}$. De esta forma, para hallar la solución del sistema original, se resuelven dos sistemas triangulares: primero el sistema triangular inferior $R\mathbf{y} = \mathbf{b}$ para hallar $\mathbf{y}$, y después el sistema triangular superior $R^T\mathbf{x} = \mathbf{y}$, para hallar la solución $\mathbf{x}$ del sistema $A\mathbf{x} = \mathbf{b}$.

Ejemplo 2.18. Sea el sistema lineal

$$\begin{cases} 3x_1 - x_2 + x_4 &= 1 \\ -x_1 + 4x_2 + x_4 &= 1 \\ 5x_3 + x_4 &= 1 \\ x_1 + x_2 + x_3 + 6x_4 &= 1. \end{cases}$$

La forma matricial de este sistema de ecuaciones es

$$A\mathbf{x} = \begin{bmatrix} 3 & -1 & 0 & 1 \\ -1 & 4 & 0 & 1 \\ 0 & 0 & 5 & 1 \\ 1 & 1 & 1 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \mathbf{b}.$$

Emplearemos el comando `chol` para calcular la factorización de Cholesky de la matriz A. Después utilizaremos las funciones `Sust_Progresiva` y `Sust_Regresiva` para resolver los sistemas $R\mathbf{y} = \mathbf{b}$ y $R^T\mathbf{x} = \mathbf{y}$, respectivamente.

```
>>> import numpy as np
>>> A = np.array([[3,-1,0,1],[-1,4,0,1],[0,0,5,1],[1,1,1,6]])
>>> b = np.array([1,1,1,1])
>>> R = chol(A)
>>> y = Sust_Progresiva(R,b)
>>> x = Sust_Regresiva(R.T,y)
>>> print(x)
[ 0.45620438  0.3649635   0.20072993 -0.00364964]
```

De esta forma, la solución del sistema de ecuaciones $A\mathbf{x} = \mathbf{b}$ es: $\mathbf{x} = \left(\frac{125}{274}, \frac{50}{137}, \frac{55}{274}, -\frac{1}{274}\right)$.

2.4.2 Cálculo del Determinante y la Inversa usando Cholesky

Cuando A es una matriz simétrica definida positiva, ella admite factorización de Cholesky $A = RR^T$. Luego, el determinante de A es

$$\det(A) = \det(RR^T) = \det(R)\det(R^T) = \det(R)^2 = \left(\prod_{i=1}^{n} r_{ii}\right)^2.$$

y la inversa de A es

$$A^{-1} = R^{-T}R^{-1}.$$

Veamos un ejemplo.

Ejemplo 2.19. Sea A la matriz cuadrada del sistema $A\mathbf{x} = \mathbf{b}$ del Ejemplo 2.18. La matriz A es simétrica definida positiva, por ello admite factorización de Colesky $A = R^TR$. Calculemos el determinante de A usando la ecuación $\det(A) = \det(R)^2$. Se tiene:

```
>>> from numpy.linalg import det
>>> A = np.array([[3,-1,0,1],[-1,4,0,1],[0,0,5,1],[1,1,1,6]])
>>> R = chol(A)
>>> det(R**2)
274.00000000000006
>>> det(A)
274.00000000000006
```

El determinante de A también se puede calcular con la ecuación $\det(A) = (\prod_{i=1}^{n} r_{ii})^2$, a través de los comandos `diag` y `prod` de la librería `Numpy`:

```
>>> from numpy import diag, prod
>>> prod(diag(R))**2
274.0
```

Note que el comando `diag(A)` extrae la diagonal de la matriz A y el comando `prod(array)` arroja el producto de los elementos del arreglo `array`. Ahora calculemos la inversa de A:

```
>>> from numpy.linalg import inv
>>> inv_R = inv(R)
>>> inv_R.T.dot(inv_R)
array([[ 0.40510949,  0.12408759,  0.01824818, -0.09124088],
       [ 0.12408759,  0.29927007,  0.01459854, -0.0729927 ],
       [ 0.01824818,  0.01459854,  0.2080292 , -0.04014599],
       [-0.09124088, -0.0729927 , -0.04014599,  0.20072993]])
```

En otras palabras, la inversa de A es

$$A^{-1} = \begin{bmatrix} \frac{111}{274} & \frac{17}{137} & \frac{5}{274} & -\frac{25}{274} \\ \frac{17}{137} & \frac{41}{137} & \frac{2}{137} & -\frac{10}{137} \\ \frac{5}{274} & \frac{2}{137} & \frac{57}{274} & -\frac{11}{274} \\ -\frac{25}{274} & -\frac{10}{137} & -\frac{11}{274} & \frac{55}{274} \end{bmatrix}.$$

2.5 Factorización *QR*

El concepto de ortogonalidad es importante en muchas aplicaciones, particularmente en Álgebra Lineal.

Definición 2.4. Una matriz $Q \in \mathbb{R}^{n\times n}$ se dice *ortogonal*, si

$$Q^T Q = I.$$

Si Q es una matriz ortogonal, entonces $Q\mathbf{x}$ se denomina *transformación ortogonal* de $\mathbf{x}$.

Las matrices ortogonales poseen buenas propiedades que las hacen fácilmente invertibles, dado que la inversa de una matriz ortogonal es justamente su traspuesta; además, el determinante de una matriz ortogonal es 1 o -1. Las transformaciones ortogonales también preservan la norma de un vector. En efecto, tenemos que

Demuestre que si Q es una matriz ortogonal, entonces, $\det(Q)$ es 1 o -1.

$$\|Q\mathbf{x}\|_2^2 = (Q\mathbf{x})^T(Q\mathbf{x}) = \mathbf{x}^T Q^T Q\mathbf{x} = \mathbf{x}^T\mathbf{x} = \|\mathbf{x}\|_2^2.$$

Debido a esta propiedad los algoritmos que involucran matrices ortogonales tienden a ser numéricamente estables.

Definición 2.5. Una matriz $A \in \mathbb{R}^{m\times n}$ tiene *factorización QR*, si

$$A = QR, \tag{2.8}$$

donde $Q \in \mathbb{R}^{m\times m}$ es una matriz ortogonal y $R \in \mathbb{R}^{m\times n}$ es una matriz triangular superior.

El siguiente teorema establece que toda matriz posee factorización *QR*.

Teorema 2.6. *Sea A una matriz de orden $m \times n$ con $m \geq n$. Entonces, existen una matriz ortogonal Q de orden $m \times m$ y una matriz triangular superior R de orden $m \times n$ tales que $A = QR$.*

Estamos interesados en la factorización *QR* de matrices cuadradas. Cuando A es una matriz de orden $n \times n$, las matrices Q y R de la factorización *QR* de A son matrices cuadradas de orden $n \times n$.

Actualmente existen tres formas de construir la factorización *QR*:

i) Transformaciones de Householder

ii) Rotaciones de Givens

iii) Proceso de ortogonalización de Gram-Schmidt

Nosotros describiremos el proceso que utiliza las Transformaciones de Householder, que sin duda es el más popular y el más efectivo para resolver sistemas generales de ecuaciones lineales mediante la factorización *QR*. En esencia, el proceso de las transformaciones de Householder es una demostración del Teorema 2.6. Para conocer acerca del proceso de construcción de la factorización *QR* usando las Rotaciones de Givens o el Proceso de ortogonalización de Gram-Schmidt, se recomienda el libro [32].

Definición 2.6. Una matriz de la forma

$$Q = I - \left(\frac{2}{\mathbf{v}^T\mathbf{v}}\right)\mathbf{v}\mathbf{v}^T$$

donde $\mathbf{v} \in \mathbb{R}^n$ es un vector no nulo, se denomina *transformación de Householder*.

Demuestre que si Q es una transformación de Householder, entonces Q es simétrica y ortogonal.

Para construir la factorización QR de A se procede de la siguiente forma. Se debe generar una sucesión de matrices simétricas y ortogonales $Q_1, Q_2, \ldots, Q_n$ tales que

$$Q_n Q_{n-1} \cdots Q_1 A = R.$$

Supongamos que A está dada por

$$A = \begin{bmatrix} c_1 & * & * & \cdots & * \\ c_2 & * & * & \cdots & * \\ c_3 & * & * & \cdots & * \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ c_m & * & * & \cdots & * \end{bmatrix},$$

donde la primera columna de A, convenientemente, la hemos denotado como el vector $\mathbf{c} = (c_1, c_2, \ldots, c_m)$ y $*$ denota un posible elemento no nulo cuyo valor en particular no es de importancia inmediata. Al multiplicar por la izquierda por la matriz Q_i, se reducen a cero todos los elementos de la i-ésima columna debajo de la diagonal, sin alterar las columnas previamente reducidas, esto es,

$$Q_1 A = \begin{bmatrix} * & * & * & \cdots & * \\ 0 & c_2 & * & \cdots & * \\ 0 & c_3 & * & \cdots & * \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & c_m & * & \cdots & * \end{bmatrix} \quad \Rightarrow \quad Q_2 Q_1 A = \begin{bmatrix} * & * & * & \cdots & * \\ 0 & * & * & \cdots & * \\ 0 & 0 & c_3 & \cdots & * \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & c_m & \cdots & * \end{bmatrix} \quad \ldots$$

donde el vector $\mathbf{c} = (*, \ldots, *, c_{i+1}, \ldots, c_m)$ denota la columna $i+1$-ésima de la matriz $Q_i Q_{i-1} \cdots Q_2 Q_1 A$. Supongamos que después de la $(k-1)$-ésima reducción tenemos

$$Q_{k-1} \cdots Q_1 A = \begin{bmatrix} * & \cdots & * & \cdots & * \\ 0 & \ddots & * & \cdots & * \\ \vdots & 0 & c_k & * & * \\ & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & c_m & * & * \end{bmatrix},$$

En este punto, debemos escoger Q_k de manera que el vector

$$\mathbf{c} = (*, \ldots, *, c_k, c_{k+1}, \ldots, c_m)$$

se reduzca al vector

$$\mathbf{r} = (r_1, \ldots, r_k, 0, 0, \ldots, 0),$$

mientras que los ceros existentes en el triángulo inferior no se modifiquen. Si siempre podemos hacer esto, entonces claramente la matriz puede ser reducida a la forma deseada por una sucesión de tales operaciones. Las transformaciones de Householder realizan precisamente esto. Escogemos

$$Q_k = I - \left(\frac{2}{\mathbf{v}^T\mathbf{v}}\right)\mathbf{v}\mathbf{v}^T, \tag{2.9}$$

donde $\mathbf{v} = (0,\ldots,0,v_k,c_{k+1},\ldots,c_m)$, con

$$v_k = c_k + \text{signo}(c_k)\sqrt{\sum_{i=k}^{m} c_i^2}$$

y la función signo(t) definida por

$$\text{signo}(t) = \begin{cases} -1, & t < 0, \\ 1, & t \geq 0. \end{cases}$$

La transformación (2.9) produce el efecto deseado, de esta forma se tiene que

$$Q_n Q_{n-1} \cdots Q_2 Q_1 A = R.$$

Como cada Q_k es simétrica y ortogonal, entonces

$$Q = (Q_n Q_{n-1} \cdots Q_2 Q_1)^T = Q_1 Q_2 \cdots Q_{n-1} Q_n \qquad \Rightarrow \qquad A = QR.$$

En el Algoritmo 2.4 se muestra el procedimiento (sin refinamiento numérico) descrito arriba para calcular la factorización *QR* de una matriz, cuya implementación se aprecia en el Programa 2.8, que hemos denominado `qr_h`. Es de resaltar que existen implementaciones de la factorización *QR* más eficientes que no construyen la matriz Q (ver [29]). El número total de operaciones punto-flotante necesarias para hallar la factorización *QR* de una matriz $A \in \mathbb{R}^{m\times n}$ usando transformaciones de Householder, es de aproximadamente $2mn^2 - \frac{2}{3}n^3$; por tanto, el número de operaciones punto-flotante para obtener la factorización *QR* de matrices cuadradas de orden $n \times n$ es aproximadamente $\frac{4}{3}n^3$, es decir, es de orden $\mathbf{O}(n^3)$.

Algoritmo 2.4 (Algoritmo de la Factorización *QR* usando transformaciones de Householder)**.**

Entrada: $A \in \mathbb{R}^{m\times n}$

1: $R := A$
2: **for** $k = 1 : n$ **do**
3: Tome $\mathbf{v} = (0,\ldots,0,v_k,r_{k+1,k},\ldots,r_{mk})$, donde

$$v_k = r_{kk} + \text{signo}(r_{kk})\sqrt{\sum_{i=k}^{m} r_{ik}^2}$$

4: $Q_k = I_{m\times m} - \left(\frac{2}{\mathbf{v}^T\mathbf{v}}\right)\mathbf{v}\mathbf{v}^T$
5: $R := Q_k R$
6: **end for**
7: $Q = Q_1 Q_2 \cdots Q_{n-1} Q_n$

Programa 2.8. Factorización *QR* usando transformaciones de Householder

```
def qr_h(A):
    R = np.asarray(A, dtype=float)
    m,n = R.shape
    Q = np.eye(m)
    for i in range(n):
        v = np.zeros(m)
        v[i:] = R[i:,i]
        v[i] = v[i] + signo(v[i])*np.linalg.norm(v)
        v = v/np.linalg.norm(v)
        Q_k = np.eye(m) - 2*prod_vv(v)
        R = Q_k.dot(R)
        Q = Q.dot(Q_k)
    return Q,R
```

El Programa 2.8 requiere de las funciones `signo` y `prod_vv`, cuya implementación se muestra a continuación.

```
def signo(t):
    if t==0:
        return 1.
    else:
        return np.sign(t)

def prod_vv(v):
    v = np.asarray(v, dtype=float)
    n = len(v)
    vv = np.empty([n,n])
    for i in range(n):
        vv[i] = v[i]*v
    return vv
```

En los siguientes ejemplos utilizamos la función `qr_h` para calcular la factorización *QR* de una matriz.

Ejemplo 2.20. Consideremos la matriz rectangular

$$A = \begin{bmatrix} 2 & 3 \\ 1 & -1 \\ 2 & 2 \\ 4 & 1 \end{bmatrix}$$

Empleemos `qr_h` para hallar la factorización *QR* de *A*.

```
>>> A = np.array([[2,3],[1,-1],[2,2],[4,1]])
>>> Q,R = qr_h(A)

>>> print(Q)
[[-0.4         0.68279835 -0.47560568 -0.38416877]
 [-0.2        -0.52951709  0.07140293 -0.82128757]
 [-0.4         0.33443185  0.85217857 -0.04412502]
 [-0.8        -0.37623583 -0.20613718  0.41946879]]
>>> print(R)
[[ -5.00000000e+00  -2.60000000e+00]
 [ -1.54705807e-17   2.87054002e+00]
 [ -1.09697352e-16  -1.66533454e-16]
 [ -7.28722533e-18   4.44089210e-16]]
```

Así, las matrices Q y R de la factorización QR de la matriz A obtenidas por `qr_h` son:

$$Q = \begin{bmatrix} -0.4 & 0.68279835 & -0.47560568 & -0.38416877 \\ -0.2 & -0.52951709 & 0.07140293 & -0.82128757 \\ -0.4 & 0.33443185 & 0.85217857 & -0.04412502 \\ -0.8 & -0.37623583 & -0.20613718 & 0.41946879 \end{bmatrix},$$

$$R = \begin{bmatrix} -5 & -2.6 \\ 0 & 2.87054002 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}.$$

Seguidamente se muestran el vector $\mathbf{v}$ normalizado, y las matrices Q_{k+1} y $Q_k \cdots Q_1 A$, obtenidos en cada iteración de la función `qr_h`.

$k = 0$:

$$\mathbf{v} = \begin{bmatrix} 0.83666003 \\ 0.11952286 \\ 0.23904572 \\ 0.47809144 \end{bmatrix}, \quad Q_1 = \begin{bmatrix} -0.4 & -0.2 & -0.4 & -0.8 \\ -0.2 & 0.97142857 & -0.05714286 & -0.11428571 \\ -0.4 & -0.05714286 & 0.88571429 & -0.22857143 \\ -0.8 & -0.11428571 & -0.22857143 & 0.54285714 \end{bmatrix},$$

$$Q_1 A = \begin{bmatrix} -5 & -2.6 \\ 0 & -1.8 \\ 0 & 0.4 \\ 0 & -2.2 \end{bmatrix}.$$

$k = 1$:

$$\mathbf{v} = \begin{bmatrix} 0. \\ -0.9019589 \\ 0.07724665 \\ -0.42485656 \end{bmatrix}, \quad Q_2 = \begin{bmatrix} 1. & 0. & 0. & 0. \\ 0. & -0.62705971 & 0.1393466 & -0.76640632 \\ 0. & 0.1393466 & 0.98806591 & 0.06563749 \\ 0. & -0.76640632 & 0.06563749 & 0.6389938 \end{bmatrix},$$

$$Q_2 Q_1 A = \begin{bmatrix} -5 & -2.6 \\ 0 & 2.87054002 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}.$$

Ejemplo 2.21. Consideremos la matriz cuadrada

$$A = \begin{bmatrix} -1 & -1 & 1 \\ 1 & 3 & 3 \\ -1 & -1 & 5 \end{bmatrix}$$

Hallemos la factorización QR de A usando `qr_h`.

```
>>> A = np.array([[-1,-1,1],[1,3,3],[-1,-1,5]])
>>> Q,R = qr_h(A)
>>> print(Q)
[[ -5.77350269e-01   -4.08248290e-01    7.07106781e-01]
 [  5.77350269e-01   -8.16496581e-01    2.77555756e-17]
 [ -5.77350269e-01   -4.08248290e-01   -7.07106781e-01]]
>>> print(R)
[[ 1.73205081   2.88675135  -1.73205081]
 [ 0.          -1.63299316  -4.89897949]
 [ 0.           0.          -2.82842712]]
```

Así, las matrices Q y R de la factorización QR de la matriz A obtenidas por `qr_h` son:

$$Q = \begin{bmatrix} -0.577350269 & -0.408248290 & 0.707106781 \\ 0.577350269 & -0.816496581 & 0 \\ -0.5.77350269 & -0.408248290 & -0.707106781 \end{bmatrix},$$

$$R = \begin{bmatrix} 1.73205081 & 2.88675135 & -1.73205081 \\ 0 & -1.63299316 & -4.89897949 \\ 0 & 0 & -2.82842712 \end{bmatrix}.$$

Seguidamente se muestran el vector $\mathbf{v}$ normalizado, y las matrices Q_{k+1} y $Q_k \cdots Q_1 A$, obtenidos en cada iteración de la función `qr_h`.

$k = 0$:

$$\mathbf{v} = \begin{bmatrix} -0.88807383 \\ 0.32505758 \\ -0.32505758 \end{bmatrix}, \quad Q_1 = \begin{bmatrix} -0.57735027 & 0.57735027 & -0.57735027 \\ 0.57735027 & 0.78867513 & 0.21132487 \\ -0.57735027 & 0.21132487 & 0.78867513 \end{bmatrix},$$

$$Q_1 A = \begin{bmatrix} 1.73205081 & 2.88675135 & -1.73205081 \\ 0 & 1.57735027 & 4 \\ 0 & 0.42264973 & 4 \end{bmatrix}.$$

$k = 1$:

$$\mathbf{v} = \begin{bmatrix} 0 \\ 0.99144486 \\ 0.13052619 \end{bmatrix}, \quad Q_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -0.96592583 & -0.25881905 \\ 0 & -0.25881905 & 0.96592583 \end{bmatrix},$$

$$Q_2 Q_1 A = \begin{bmatrix} 1.73205081 & 2.88675135 & -1.73205081 \\ 0 & -1.63299316 & -4.89897949 \\ 0 & 0 & 2.82842712 \end{bmatrix}.$$

$k = 2$:

$$\mathbf{v} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \quad Q_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix},$$

$$Q_3 Q_2 Q_1 A = \begin{bmatrix} 1.73205081 & 2.88675135 & -1.73205081 \\ 0 & -1.63299316 & -4.89897949 \\ 0 & 0 & -2.82842712 \end{bmatrix}.$$

2.5.1 Solución de un Sistema Lineal usando QR

Sea A una matriz cuadrada de orden $n \times n$. Sean $Q \in \mathbb{R}^{n\times n}$ y $R \in \mathbb{R}^{n\times n}$ las matrices de la factorización QR de A. Como $A = QR$ con Q ortogonal y R triangular superior, el sistema lineal $A\mathbf{x} = \mathbf{b}$ adquiere la forma

$$QR\mathbf{x} = \mathbf{b} \qquad \Leftrightarrow \qquad R\mathbf{x} = Q^T\mathbf{b}.$$

De esta forma, la solución del sistema triangular superior $R\mathbf{x} = Q^T\mathbf{b}$ también es la solución del sistema original.

Ejemplo 2.22. Consideremos el sistema $A\mathbf{x} = \mathbf{b}$ del Ejemplo 2.18. Emplearemos el comando `qr_h` para calcular la factorización de QR de la matriz A. Después utilizaremos la función `Sust_Regresiva` para resolver el sistema $R\mathbf{x} = Q^T\mathbf{b}$.

```
>>> import numpy as np
>>> A = np.array([[3,-1,0,1],[-1,4,0,1],[0,0,5,1],[1,1,1,6]])
>>> b = np.array([1,1,1,1])
>>> Q,R = qr_h(A)
>>> x = Sust_Regresiva(R, Q.T.dot(b))
>>> print(x)
[ 0.45620438  0.3649635   0.20072993 -0.00364964]
```

De esta forma, la solución del sistema de ecuaciones $A\mathbf{x} = \mathbf{b}$ es: $\mathbf{x} = \left(\frac{125}{274}, \frac{50}{137}, \frac{55}{274}, -\frac{1}{274}\right)$.

■

2.5.2 Cálculo del Determinante y la Inversa usando QR

Tome en cuenta que $Q^TQ = I$ y $\det(Q)$ es 1 o -1

Consideremos una matriz cuadrada $A \in \mathbb{R}^{n\times n}$. Ella admite factorización QR, esto es, $A = QR$, donde $Q \in \mathbb{R}^{n\times n}$ es una matriz ortogonal y $R \in \mathbb{R}^{n\times n}$ es una matriz triangular superior. Luego, el determinante de A es

$$\det(A) = \det(QR) = \det(Q)\det(R) = \det(Q)\prod_{i=1}^{n} r_{ii}.$$

y, si A es no singular, la inversa de A es

$$A^{-1} = R^{-1}Q^{-1} = R^{-1}Q^T.$$

2.6 Descomposición en Valores Singulares

En esta sección describiremos una factorización que tiene una gran variedad de aplicaciones, cuyo cálculo es un paso de muchos algoritmos. Se une a la factorización LU y a la factorización QR. No se adjunta el nombre de nadie; se conoce como *descomposición en valores singulares* y se denota por "SVD" (Singular Value Decomposition). Las factorizaciones hasta ahora estudiadas se calculan con un número finito de operaciones punto-flotante; en cambio, para calcular la factorización SVD se requiere de un proceso iterativo estrechamente relacionado con el cálculo de autovalores y autovectores (ver Capítulo 6). Una definición formal de la SVD es la siguiente.

Definición 2.7. Dada una matriz $A \in \mathbb{R}^{m\times n}$ con $m \geq n$, una SVD de A es una factorización

$$A = U\Sigma V^T, \tag{2.10}$$

donde

$$\begin{aligned} U &\in \mathbb{R}^{m\times m} \text{ es ortogonal,} \\ V &\in \mathbb{R}^{n\times n} \text{ es ortogonal,} \\ \Sigma &\in \mathbb{R}^{m\times n} \text{ es diagonal.} \end{aligned}$$

Además se asume que las entradas diagonales σ_j de Σ son no negativas y aparecen en orden decreciente, esto es, $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$. Las entradas $\sigma_1, \sigma_2, \ldots, \sigma_n$ se conocen como *valores singulares* de A. Las columnas $\mathbf{u}_j$ de U se conocen como los *vectores singulares por la izquierda* y las columnas $\mathbf{v}_j$ de V se conocen como los *vectores singulares por la derecha*.

En la Definición 2.7, cuando $m < n$, se construye la factorización SVD de A^T. Tenga presente que la matriz diagonal Σ tiene la misma forma de A incluso cuando A no sea una matriz cuadrada, pero las matrices U y V siempre son matrices ortogonales. Asimismo, los valores singulares de A se determinan de manera única, mientras que las matrices U y V no son únicas en general. Veamos la relación de los valores singulares de A con los autovalores de las matrices A^TA y AA^T. Al multiplicar (por la derecha) por la matriz ortogonal V en ambos lados de la ecuación (2.10) obtenemos

¿Por qué las matrices U y V no son únicas?

$$AV = U\Sigma,$$

lo que implica

$$A\mathbf{v}_j = \sigma_j \mathbf{u}_j, \quad j = 1, \ldots, n. \tag{2.11}$$

Por otra parte, tomando traspuesta en ambos lados de (2.10) obtenemos la ecuación

$$A^T = V\Sigma^T U^T.$$

Multiplicando por la matriz ortogonal U en ambos lados de la ecuación anterior obtenemos

$$A^T U = V\Sigma^T,$$

de la cual se deduce

$$A^T\mathbf{u}_j = \sigma_j \mathbf{v}_j, \quad j = 1, \ldots, n. \tag{2.12}$$

Considerando (2.11) y (2.12) se infiere la siguiente igualdad matricial:

$$\begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} \mathbf{u}_j \\ \mathbf{v}_j \end{bmatrix} = \sigma_j \begin{bmatrix} \mathbf{u}_j \\ \mathbf{v}_j \end{bmatrix}, \quad j = 1, \ldots, n. \tag{2.13}$$

En otras palabras, los valores singulares de A son los autovalores de una matriz simétrica y por tanto son números reales. Más aún, multiplicando A^T en (2.11) y usando apropiadamente a (2.11) obtenemos:

$$A^TA\mathbf{v}_j = \sigma_j A^T\mathbf{u}_j = \sigma_j^2 \mathbf{v}_j, \quad j = 1, \ldots, n.$$

De forma similar se obtiene

$$AA^T\mathbf{u}_j = \sigma_j A\mathbf{v}_j = \sigma_j^2 \mathbf{u}_j, \quad j = 1, \ldots, n.$$

Luego, los valores singulares de A al cuadrado son los autovalores de A^TA y AA^T. Generalmente los algoritmos para calcular los valores singulares son análogos a los algoritmos para calcular los autovalores de matrices simétricas. Ver Capítulo 6 para información adicional. Algunos algoritmos encuentran las raíces cuadradas de los autovalores de las matrices A^TA o AA^T (sin calcular realmente A^TA o AA^T), que en general pueden ser mal condicionadas. Otros algoritmos, aproximan los autovalores de la matriz $(m+n)\times(m+n)$ en bloques que aparece en (2.13). En todo caso, el cálculo de la SVD es más costoso que el cálculo de las factorizaciones directas (QR, LU, Cholesky). Todos los algoritmos suponen que $m \geq n$; si $m < n$, los algoritmos pueden aplicarse a A^T. Una implementación eficiente ha sido desarrollada en el módulo `linalg` de la librería `numpy` a través de la función `U,S,Vt=svd(A)`, que retorna las matrices U y V^T, y un vector S cuyas coordenadas son los valores singulares de A ordenados en forma decreciente.

Ejemplo 2.23. Sea $A \in \mathbb{R}^{3\times 2}$ dada por

$$A = \begin{bmatrix} -3\sqrt{2}/2 & -3\sqrt{2}/2 \\ 1 & -1 \\ 1 & -1 \end{bmatrix}.$$

Calculemos la factorización SVD usando la función `svd`.

```
>>> from numpy.linalg import svd
>>> A = np.array([[-3*(2**.5)/2,-3*(2**.5)/2],[1,-1],[1,-1]])
>>> U,S,Vt = svd(A)
>>> print(U)
[[-1.00000000e+00 -1.11022302e-16 -5.08496105e-17]
 [ 0.00000000e+00 -7.07106781e-01 -7.07106781e-01]
 [ 5.55111512e-17 -7.07106781e-01  7.07106781e-01]]
>>> print(S)
[3. 2.]
>>> print(Vt.T)
[[ 0.70710678 -0.70710678]
 [ 0.70710678  0.70710678]]
```

Los valores singulares de A obtenidos por `svd` son: $3 > 2 > 0$, además, las matrices U, Σ y V obtenidas son:

$$U = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -\sqrt{2}/2 & -\sqrt{2}/2 \\ 0 & -\sqrt{2}/2 & \sqrt{2}/2 \end{bmatrix}, \quad \Sigma = \begin{bmatrix} 3 & 0 \\ 0 & 2 \\ 0 & 0 \end{bmatrix}, \quad V = \begin{bmatrix} \sqrt{2}/2 & -\sqrt{2}/2 \\ \sqrt{2}/2 & \sqrt{2}/2 \end{bmatrix}.$$

■

Uno puede dar una interpretación geométricamente de la SVD, la cual proviene del siguiente hecho geométrico: "la imagen de la esfera unitaria en $\mathbb{R}^n$ bajo una matriz $m\times n$ es una hipereplipse en $\mathbb{R}^n$". El término hipereplipse quizás no sea familiar, pero éste es justamente la generalización de una elipse. Se puede definir una hiperelipse en $\mathbb{R}^m$ como la superficie obtenida estirando la esfera unitaria en $\mathbb{R}^m$ por algunos factores $\sigma_1, \sigma_2, \ldots, \sigma_m$ (posiblemente cero) en algunas direcciones ortogonales $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_m \in \mathbb{R}^m$. Por conveniencia, $\mathbf{u}_j$ son vectores unitarios, es decir, $\|\mathbf{u}_j\|_2 = 1$. Los vectores $\{\sigma_j \mathbf{u}_j\}$ son los ejes principales de la hiperelipse, con longitudes $\sigma_1, \sigma_2, \ldots, \sigma_m$. Si A tiene rango r, entonces exactamente r de las longitudes σ_j resultarán ser no nulas, y en particular, si $m \geq n$, al menos n de ellas deben ser no cero (para más detalles ver [32]).

Ejemplo 2.24. Consideremos la matriz

$$A = \begin{bmatrix} 1 & 2 \\ 1 & 1 \end{bmatrix}.$$

Sea $S = \{\mathbf{x} \in \mathbb{R}^2 : \|\mathbf{x}\|_2 = 1\}$ la esfera unitaria en $\mathbb{R}^2$. Entonces, AS es una elipse en $\mathbb{R}^2$. Construyamos esta elipse usando la descomposición SVD de A. Se tiene:

```
>>> from numpy.linalg import svd
>>> A = np.array([[1,2],[2,1]])
>>> U,S,Vt = svd(A)
>>> U
array([[-0.70710678, -0.70710678],
       [-0.70710678,  0.70710678]])
>>> S
array([3., 1.])
>>> Vt.T
array([[-0.70710678,  0.70710678],
       [-0.70710678, -0.70710678]])
```

Los valores singulares de A son: $\sigma_1 = 3$ y $\sigma_2 = 1$; además, los vectores

$$\mathbf{v}_1 = (-0.70710678, -0.70710678), \quad \mathbf{v}_2 = (0.70710678, -0.70710678)$$

satisfacen $\|\mathbf{v}_1\|_2 = \|\mathbf{v}_2\|_2 = 1$. Ahora, los vectores

$$A\mathbf{v}_1 = \sigma_1\mathbf{u}_1 = (-2.12132034, -2.12132034), \quad A\mathbf{v}_2 = \sigma_2\mathbf{u}_2 = (-0.70710678, 0.70710678)$$

son los semiejes principales de la elipse que se muestra en la Figura 2.1.

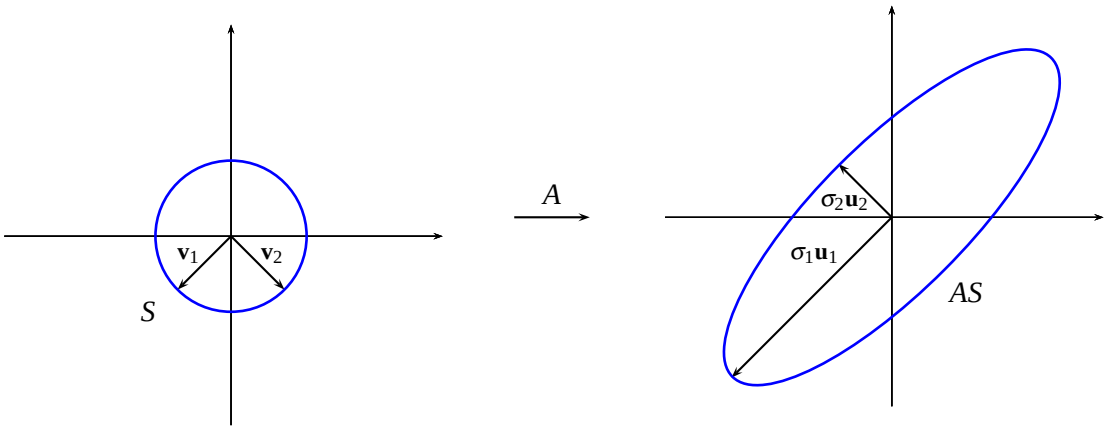


Figura 2.1. Interpretación geométrica de la SVD para una matriz 2×2

Seguidamente mostramos las propiedades teórico-prácticas más importantes de la SVD. Sin pérdida de generalidad, supondremos que la matriz $A \in \mathbb{R}^{m\times n}$ posee factorización SVD $A = U\Sigma V^T$, con $m \geq n$ y $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$ sus valores singulares. Entonces, las siguientes propiedades se cumplen (para su demostración ver [32]):

i) El rango de A es: $\text{rang}(A) = r$, donde r es el número de valores singulares estrictamente positivos.

ii) $\|A\|_2 = \sigma_1$.

iii) $\|A\|_F = \left(\sigma_1^2 + \sigma_2^2 + \cdots + \sigma_n^2\right)^{1/2}$.

iv) Los valores singulares no nulos de A son las raíces cuadradas de los autovalores de A^TA o AA^T. (Estas matrices poseen los mismos autovalores no nulos.)

v) Si A es cuadrada y además simétrica, entonces los valores singulares de A son los valores absoluto de los autovalores de A.

vi) Si A es cuadrada y no singular, entonces $\|A^{-1}\|_2 = 1/\sigma_n$ y además el número de condición de A está dado por $\kappa_2(A) = \|A\|_2\|A^{-1}\|_2 = \sigma_1/\sigma_n$.

vii) $|\det(A)| = \prod_{j=1}^n \sigma_j$.

viii) Si $r = \text{rang}(A)$, entonces la matriz A es la suma de r matrices de rango 1,

$$A = \sum_{j=1}^{r} \sigma_j \mathbf{u}_j \mathbf{v}_j^T.$$

ix) Sea $r = \text{rang}(A)$. Sea $1 \leq k \leq r$ y definamos $A_k = \sum_{j=1}^{k} \sigma_j \mathbf{u}_j \mathbf{v}_j^T$. Entonces, A_k es la matriz de rango k más cercana a A y la distancia es

$$\|A - A_k\|_2 = \sigma_{k+1} \qquad \text{y} \qquad \|A - A_k\|_F = (\sigma_{k+1}^2 + \cdots + \sigma_r^2)^{1/2}.$$

2.7 Métodos Iterativos

Sea $A\mathbf{x} = \mathbf{b}$ un sistema lineal, donde $A \in \mathbb{R}^{n\times n}$ es un matriz cuadrada y $\mathbf{b} \in \mathbb{R}^n$ es el vector del lado derecho. La eliminación gaussiana y sus variantes son conocidos como métodos *directos* para resolver el sistema lineal $A\mathbf{x} = \mathbf{b}$. Se ejecutan a través de un número finito de pasos y generan una solución exacta $\mathbf{x}^*$ (por supuesto, en aritmética exacta). En cambio, un método *iterativo* genera una sucesión de vectores que converge a la solución. El cálculo se detiene cuando se alcanza un máximo de iteraciones o cuando se cuenta con una solución aproximada con cierto grado de precisión especificado de antemano. Para sistemas lineales grandes ($n \gg 1000$), los métodos iterativos regularmente son más ventajosos que los métodos directos, en cuanto a velocidad y requerimiento de memoria para almacenamiento. En esta sección estudiaremos los métodos iterativos estacionarios básicos: *Jacobi*, *Gauss-Seidel* y *SOR*. De los métodos no estacionarios nos centraremos en los métodos basados en subespacios de Krylov.

2.7.1 Jacobi, Gauss-Seidel y SOR

En general, un método iterativo estacionario toma un vector $\mathbf{x}_0 \in \mathbb{R}^n$ como aproximación inicial y genera una sucesión de vectores $\{\mathbf{x}_k\}$ que converge a la solución $\mathbf{x}^*$. Un tipo general de método iterativo estacionario se puede describir de la siguiente forma. La sucesión de vectores $\{\mathbf{x}_k\}$ se genera como:

$$\mathbf{x}_{k+1} = M\mathbf{x}_k + \mathbf{v}, \quad k = 0, 1, \ldots \tag{2.14}$$

donde la matriz $M \in \mathbb{R}^{n\times n}$ y el vector $\mathbf{v} \in \mathbb{R}^n$, se escogen de tal manera que la ecuación

$$\mathbf{x} = M\mathbf{x} + \mathbf{v}$$

es equivalente al sistema $A\mathbf{x} = \mathbf{b}$, es decir, tienen la misma solución. Según sea la manera de escoger la matriz M y el vector $\mathbf{v}$, se define un método iterativo estacionario particular. La ecuación (2.14) normalmente se denomina *forma vectorial* del método iterativo.

Para la descripción de los métodos iterativos estacionarios básicos: *Jacobi*, *Gauss-Seidel* y *SOR*, es necesario introducir una notación. Sea la descomposición de la matriz A dada por:

$$A = D - (E + F), \tag{2.15}$$

donde D es una matriz diagonal, E es una matriz triangular inferior estricta y F es una matriz triangular superior estricta, definidas respectivamente como:

$$D = \begin{bmatrix} a_{11} & 0 & 0 & \cdots & 0 \\ 0 & a_{22} & 0 & \cdots & 0 \\ 0 & 0 & a_{33} & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ 0 & 0 & \cdots & 0 & a_{nn} \end{bmatrix},$$

$$E = \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 \\ -a_{21} & 0 & 0 & \cdots & 0 \\ -a_{31} & -a_{32} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ -a_{n1} & -a_{n2} & \cdots & -a_{n,n-1} & 0 \end{bmatrix}, \quad F = \begin{bmatrix} 0 & -a_{12} & -a_{13} & \cdots & -a_{1n} \\ 0 & 0 & -a_{23} & \cdots & -a_{2n} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & -a_{n-1,n} \\ 0 & 0 & \cdots & 0 & 0 \end{bmatrix}.$$

Las matrices D, E y F de la descomposición $A = D - (E + F)$, se pueden hallar respectivamente usando los comandos `diag`, `tril` y `triu` de la librería `Numpy`. Veamos un ejemplo.

Ejemplo 2.25. Considere la matrix

$$A = \begin{bmatrix} 4 & 3 & 2 & 1 \\ 5 & 3 & 1 & 2 \\ 6 & 4 & 2 & 2 \\ 7 & 5 & 1 & 1 \end{bmatrix}$$

Se tiene:

```
>>> import numpy as np
>>> A = np.array([[4,3,2,1],[5,3,1,2],[6,4,2,2],[7,5,1,1]])
>>> D = np.diag(np.diag(A))
>>> print(D)
[[4 0 0 0]
 [0 3 0 0]
 [0 0 2 0]
 [0 0 0 1]]
>>> E = -np.tril(A,-1)
>>> print(E)
[[ 0  0  0  0]
 [-5  0  0  0]
 [-6 -4  0  0]
 [-7 -5 -1  0]]
```

```
>>> F = -np.triu(A,1)
>>> print(F)
[[ 0 -3 -2 -1]
 [ 0  0 -1 -2]
 [ 0  0  0 -2]
 [ 0  0  0  0]]
```

Sustituyendo la descomposición de A dada en (2.15) en el sistema $A\mathbf{x} = \mathbf{b}$, obtenemos:

$$(D-(E+F))\mathbf{x} = \mathbf{b}$$

Los métodos iterativos clásicos se basan en despejar convenientemente $\mathbf{x}$ de la ecuación anterior, por ejemplo, el método de Jacobi se obtiene despejando $\mathbf{x}$ de la siguiente manera:

$$\mathbf{x} = D^{-1}(E+F)\mathbf{x} + D^{-1}\mathbf{b}$$

o, el método de Gauss-Seidel, despejando $\mathbf{x}$ como

$$\mathbf{x} = (D-E)^{-1}F\mathbf{x} + (D-E)^{-1}\mathbf{b}.$$

Cualquier despeje de este tipo, puede ser utilizado para generar un método iterativo, pero se debe tener en cuenta lo siguiente: *garantizar la convergencia teórica del método* (ver Teorema 2.7). Los métodos iterativos clásicos han sido sumamente estudiados y en la literatura se consiguen distintos resultados de convergencia. A continuación describimos los métodos iterativos clásicos y algunos resultados de convergencia.

Método de Jacobi

La forma vectorial del método de Jacobi es la siguiente. Dado un vector inicial $\mathbf{x}_0 \in \mathbb{R}^n$, el método de Jacobi genera la aproximación $k+1$-ésima como:

$$\mathbf{x}_{k+1} = D^{-1}(E+F)\,\mathbf{x}_k + D^{-1}\mathbf{b}, \quad k = 0, 1, \ldots \tag{2.16}$$

En este caso, la matriz M y el vector $\mathbf{v}$ de (2.14) se escogen como:

$$M_J = D^{-1}(E+F), \qquad \mathbf{v}_J = D^{-1}\mathbf{b}.$$

La forma vectorial del método de Jacobi se puede obtener manipulando la forma escalar del mismo. La forma escalar del método de Jacobi se obtiene con el siguiente razonamiento. Sea la i-ésima ecuación del sistema lineal $A\mathbf{x} = \mathbf{b}$, esto es,

$$a_{i1}x_1 + \cdots + a_{ii}x_i + \cdots a_{in}x_n = b_i.$$

Asumamos que $a_{ii} \neq 0$. Ahora, despejando la incógnita x_i en la ecuación anterior, obtenemos:

$$x_i = \frac{1}{a_{ii}}\left(b_i - \sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij}x_i\right).$$

Considerando la ecuación anterior se define la forma escalar del método de Jacobi: las componentes de la aproximación $\mathbf{x}_{k+1} = (x_1^{k+1}, x_2^{k+1}, \ldots, x_n^{k+1})$ están dadas por:

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_i^k \right), \quad i = 1, 2, \ldots, n. \tag{2.17}$$

Note que la ecuación (2.17) está bien definida cuando $a_{ii} \neq 0$ para todo $i = 1, \ldots, n$, lo cual indica que el método de Jacobi no se puede usar cuando la matriz A tiene elementos nulos en la diagonal.

Método de Gauss-Seidel

El método de Gauss-Seidel corrige la i-ésima componente del iterado

$$\mathbf{x}_{k+1} = (x_1^{k+1}, x_2^{k+1}, \ldots, x_n^{k+1})$$

del método de Jacobi, esto es:

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_i^{k+1} - \sum_{j=i+1}^{n} a_{ij} x_i^k \right), \quad i = 1, 2, \ldots, n. \tag{2.18}$$

En (2.18), $\sum_{j=1}^{0}() \equiv 0$.

Gauss-Seidel incorpora las componentes $x_1^{k+1}, \ldots, x_{i-1}^{k+1}$ ya calculadas, en la aproximación de la componente x_i^{k+1}. El método de Gauss-Seidel se puede ver como una "mejora" del método de Jacobi. Manipulando la forma escalar (2.18) de Gauss-Seidel se obtiene su forma vectorial:

$$\mathbf{x}_{k+1} = (D - E)^{-1} F \mathbf{x}_k + (D - E)^{-1} \mathbf{b}. \tag{2.19}$$

En este caso, la matriz M y el vector $\mathbf{v}$ de (2.14) se escogen como:

$$M_{GS} = (D - E)^{-1} F, \qquad \mathbf{v}_{GS} = (D - E)^{-1} \mathbf{b}.$$

Note que en el método de Gauss-Seidel, igual que en el método Jacobi, se debe asumir que la matriz A no tiene elementos nulos en su diagonal principal.

Método SOR

Considerando la forma escalar del método de Gauss-Seidel introducimos la forma escalar del método SOR (successive over-relaxation):

$$x_i^{k+1} = \frac{\omega}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_i^{k+1} - \sum_{j=i+1}^{n} a_{ij} x_i^k \right) + (1 - \omega) x_i^k, \quad i = 1, 2, \ldots, n. \tag{2.20}$$

El método SOR introduce el parámetro real ω en la forma escalar del método de Gauss-Seidel. La forma vectorial del método SOR es:

$$\mathbf{x}_{k+1} = (D - \omega E)^{-1} [\omega F + (1 - \omega) D] \mathbf{x}_k + (D - \omega E)^{-1} \omega \mathbf{b} \tag{2.21}$$

En este caso, la matriz M y el vector $\mathbf{v}$ de (2.14) se escogen como:

$$M_{SOR} = (D - \omega E)^{-1} [\omega F + (1 - \omega) D], \qquad \mathbf{v}_{SOR} = (D - \omega E)^{-1} \omega \mathbf{b}.$$

Note que si $\omega = 1$, el método SOR coincide con el método de Gauss-Seidel.

Los Programas 2.9, 2.10 y 2.11, muestran la implementación de la forma vectorial de los métodos de Jacobi, Gauss-Seidel y SOR, respectivamente. Observe que el criterio de parada de todas estas funciones es:

$$\|\mathbf{r}_k\|_1 \leq \texttt{tol},$$

donde $\mathbf{r}_k = A\mathbf{x}_k - \mathbf{b}$ es el k-ésimo residual y $0 < \texttt{tol} < 1$. El proceso también se detiene cuando se alcanza un máximo de iteraciones, `maxit`. Además, `tol=1.0e-4` y `maxit=100` son valores por defecto.

Programa 2.9. Método de Jacobi

```
def jacobi(A,b,x0,maxit=100,tol=1.0e-4):
    b = np.asarray(b,dtype=float)
    n,m = A.shape
    if n!=m:
        print('La matriz no es cuadrada')
        return
    if len(b) != n:
        print('Dimensión incorrecta del vector b')
        return
    res = []
    D = np.diag(A)
    xk = np.copy(np.asarray(x0, dtype=float))
    rk = A.dot(xk) - b
    Nrk = np.linalg.norm(rk, 1)
    res.append(Nrk)
    k = 0
    while True:
        if Nrk<=tol:
            print('Terminó satisfactoriamente')
            break
        if k>=maxit:
            print('Alcanzó el máximo de iteraciones')
            break
        xk = xk - rk/D
        rk = A.dot(xk) - b
        Nrk = np.linalg.norm(rk, 1)
        res.append(Nrk)
        k += 1
    return xk,k,np.array(res)
```

Programa 2.10. Método de Gauss-Seidel

```
def gauss_seidel(A,b,x0,maxit=100,tol=1.0e-4):
    b = np.asarray(b,dtype=float)
    n,m = A.shape
    if n!=m:
        print('La matriz no es cuadrada')
        return
    if len(b) != n:
        print('Dimension incorrecta del vector b')
        return
    res = []
    DE = np.tril(A)
    xk = np.copy(np.asarray(x0, dtype=float))
    rk = A.dot(xk) - b
    Nrk = np.linalg.norm(rk, 1)
    res.append(Nrk)
    k = 0
    while True:
        if Nrk<=tol:
            print('Terminó satisfactoriamente')
            break
        if k>=maxit:
            print('Alcanzó el máximo de iteraciones')
            break
        xk = xk - Sust_Progresiva(DE,rk)
        rk = A.dot(xk) - b
        Nrk = np.linalg.norm(rk, 1)
        res.append(Nrk)
        k += 1
    return xk,k,np.array(res)
```

Programa 2.11. Método SOR

```
def sor(A,b,x0,w,maxit=100,tol=1.0e-4):
    b = np.asarray(b,dtype=float)
    n,m = A.shape
    if n!=m:
        print('La matriz no es cuadrada')
        return
    if len(b) != n:
        print('Dimension incorrecta del vector b')
        return
    res = []
    DE = w*np.tril(A,-1)
    for i in range(n):
        DE[i,i] = A[i,i]
    xk = np.copy(np.asarray(x0, dtype=float))
    rk = A.dot(xk) - b
    Nrk = np.linalg.norm(rk, 1)
    res.append(Nrk)
    k = 0
    while True:
        if Nrk<=tol:
            print('Terminó satisfactoriamente')
            break
        if k>=maxit:
            print('Alcanzó el máximo de iteraciones')
            break
        xk = xk - w*Sust_Progresiva(DE, rk)
        rk = A.dot(xk) - b
        Nrk = np.linalg.norm(rk, 1)
        res.append(Nrk)
        k += 1
    return xk,k,np.array(res)
```

Convergencia

El siguiente teorema establece algunas condiciones para que un método iterativo descrito por la ecuación (2.14) converja a la solución del sistema $A\mathbf{x} = \mathbf{b}$.

Teorema 2.7. *Sea M una matriz cuadrada tal que $\|M\| < 1$ para alguna norma matricial $\|\cdot\|$. Entonces, la iteración* (2.14) *converge a la solución del sistema $A\mathbf{x} = \mathbf{b}$, para algún vector inicial $\mathbf{x}_0 \in \mathbb{R}^n$.*

Un resultado de convergencia general de los métodos de Jacobi, Gauss-Seidel y SOR, se establece en el siguiente teorema. Este teorema es una consecuencia inmediata del Teorema 2.7.

Teorema 2.8. *Sea $\|\cdot\|$ una norma matricial. Los métodos de Jacobi, Gauss-Seidel y SOR satisfacen:*

- *Si $\|D^{-1}(E+F)\| < 1$, entonces la forma iterativa del método de Jacobi dada por* (2.16)*, converge a la solución del sistema $A\mathbf{x} = \mathbf{b}$, para algún vector inicial $\mathbf{x}_0 \in \mathbb{R}^n$.*
- *Si $\|(D-E)^{-1}F\| < 1$, entonces la forma iterativa del método de Gauss-Seidel dada por* (2.19)*, converge a la solución del sistema $A\mathbf{x} = \mathbf{b}$, para algún vector inicial $\mathbf{x}_0 \in \mathbb{R}^n$.*
- *Si $\|(D-\omega E)^{-1}[\omega F + (1-\omega)D]\| < 1$ para algún $\omega > 0$, entonces la forma iterativa del método SOR dada por* (2.21)*, converge a la solución del sistema $A\mathbf{x} = \mathbf{b}$, para algún vector inicial $\mathbf{x}_0 \in \mathbb{R}^n$.*

Ahora bien, existen matrices A tales que los métodos Jacobi, Gauss-Seidel y SOR siempre convergen a la solución del sistema $A\mathbf{x} = \mathbf{b}$, para cualquier iterado inicial $\mathbf{x}_0 \in \mathbb{R}^n$. Tales matrices se indican en los siguientes teoremas.

Teorema 2.9. *Si $A \in \mathbb{R}^{n\times n}$ es una matriz estrictamente diagonal dominante, entonces la sucesión $\{\mathbf{x}_k\}$ generada por el método de Jacobi converge a la solución de $A\mathbf{x} = \mathbf{b}$, para cualquier iterado inicial $\mathbf{x}_0 \in \mathbb{R}^n$.*

Teorema 2.10. *Si $A \in \mathbb{R}^{n\times n}$ es una matriz estrictamente diagonal dominante, entonces la sucesión $\{\mathbf{x}_k\}$ generada por el método de Gauss-Seidel converge a la solución de $A\mathbf{x} = \mathbf{b}$, para cualquier iterado inicial $\mathbf{x}_0 \in \mathbb{R}^n$.*

Teorema 2.11. *Si $A \in \mathbb{R}^{n\times n}$ es simétrica con elementos de la diagonal positivos y $0 < \omega < 2$, entonces el método SOR converge a la solución de $A\mathbf{x} = \mathbf{b}$ para cualquier iterado inicial $\mathbf{x}_0 \in \mathbb{R}^n$ si, y sólo si A es definida positiva.*

En el siguiente ejemplo verificamos las condiciones de convergencia de los métodos Jacobi, Gauss-Seidel y SOR.

Ejemplo 2.26. Demostremos que los métodos Jacobi, Gauss-Seidel y SOR convergen a la solución del sistema $A\mathbf{x} = \mathbf{b}$ para cualquier iterado inicial $\mathbf{x}_0 \in \mathbb{R}^5$, donde

$$A = \begin{bmatrix} 5 & -1 & 1 & 0 & 1 \\ -1 & 4 & 0 & -2 & 0 \\ 1 & 0 & 3 & 0 & -1 \\ 0 & -2 & 0 & 6 & 0 \\ 1 & 0 & -1 & 0 & 8 \end{bmatrix}, \qquad b = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}.$$

Probaremos la convergencia de los métodos usando dos razonamientos distintos: (i) aplicando el Teorema 2.8 y (ii) aplicando los Teoremas 2.9, 2.10 o 2.11.

(i) Se tiene que:

$$D = \begin{bmatrix} 5 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 & 8 \end{bmatrix}, \quad E = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 \end{bmatrix}, \quad F = \begin{bmatrix} 0 & 1 & -1 & 0 & -1 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

De esta forma, aplicando el Teorema 2.8 obtenemos:

- como $\|D^{-1}(E+F)\|_\infty = 0.75 < 1$, entonces el método de Jacobi converge a la solución del sistema $A\mathbf{x} = \mathbf{b}$, para algún vector inicial $\mathbf{x}_0 \in \mathbb{R}^5$;
- como $\|(D-E)^{-1}F\|_\infty = 0.65 < 1$, entonces el método de Gauss-Seidel converge a la solución del sistema $A\mathbf{x} = \mathbf{b}$, para algún vector inicial $\mathbf{x}_0 \in \mathbb{R}^5$;
- como $\|(D-\omega E)^{-1}[\omega F + (1-\omega)D]\|_\infty < 1$ para $0 < \omega \leq 1.2$, entonces el método SOR converge a la solución del sistema $A\mathbf{x} = \mathbf{b}$, para algún vector inicial $\mathbf{x}_0 \in \mathbb{R}^5$.

Se deja como ejercicio para el lector verificar los valores de las normas $\|\cdot\|_\infty$ obtenidos anteriormente; para ello se recomienda usar el comando `norm`.

(ii) Se tiene que:

$$|a_{11}| = 5 > 3 = \sum_{\substack{i=1 \\ i\neq 1}}^{5} |a_{i1}|, \quad |a_{22}| = 4 > 3 = \sum_{\substack{i=1 \\ i\neq 2}}^{5} |a_{i2}|, \quad |a_{33}| = 3 > 2 = \sum_{\substack{i=1 \\ i\neq 3}}^{5} |a_{i3}|,$$

$$|a_{44}| = 6 > 2 = \sum_{\substack{i=1 \\ i\neq 4}}^{5} |a_{i4}|, \quad |a_{55}| = 8 > 2 = \sum_{\substack{i=1 \\ i\neq 5}}^{5} |a_{i5}|.$$

Por tanto, A es una matriz estrictamente diagonal dominante. De esta forma, por los Teoremas 2.9, 2.10, los métodos Jacobi y Gauss-Seidel convergen a la solución del sistema $A\mathbf{x} = \mathbf{b}$, para cualquier iterado inicial $\mathbf{x}_0 \in \mathbb{R}^5$. Ahora, como A es simétrica definida positiva, entonces aplicando el Teorema 2.11 obtenemos que el método SOR converge a la solución del sistema $A\mathbf{x} = \mathbf{b}$, para $0 < \omega < 2$ y cualquier iterado inicial $\mathbf{x}_0 \in \mathbb{R}^5$. Se deja como ejercicio para el lector comprobar que A es definida positiva; para ello use el comando `cholesky(A)`.

Pasemos ahora a ver el comportamiento computacional de los métodos Jacobi, Gauss-Seidel y SOR.

Ejemplo 2.27. Consideremos el sistema $A\mathbf{x} = \mathbf{b}$ con la matriz A y el vector $\mathbf{b}$ dados en el Ejemplo 2.26. La solución de este sistema es:

```
>>> import numpy as np
>>> A = np.array([[5,-1,1,0,1],
...               [-1,4,0,-2,0],
...               [1,0,3,0,-1],
...               [0,-2,0,6,0],
...               [1,0,-1,0,8]])
>>> b = np.ones(5)
>>> from numpy.linalg import solve
>>> sol = np.linalg.solve(A,b)
>>> print(sol)
[ 0.20189274  0.46056782  0.31230284  0.32018927  0.13880126]
```

Empleemos las funciones `jacobi`, `gauss_seidel` y `sor`, para encontrar una aproximación de la solución del sistema, tomando $\mathbf{x}_0 = (0,0,0,0,0)$ como aproximación inicial. Para `sor`, tomamos $\omega = 1.2$. Definimos el iterado inicial.

```
>>> x0 = np.zeros(5)
```

Resolvemos el sistema con Jacobi:

```
>>> sol_j,it_j,res_j = jacobi(A,b,x0)
Terminó satisfactoriamente
>>> print('Solución: {0}\nIteraciones: {1}'.format(sol_j.T,it_j))
Solución: [ 0.20188685  0.46056149  0.31230786  0.32018455  0.13880381]
Iteraciones: 16
```

Resolvemos el sistema con Gauss-Seidel:

```
>>> sol_gs,it_gs,res_gs = gauss_seidel(A,b,x0)
Terminó satisfactoriamente
>>> print('Solución: {0}\nIteraciones: {1}'.format(sol_gs.T,it_gs))
Solución: [ 0.20187623  0.46055745  0.31231221  0.32018582  0.1388045 ]
Iteraciones: 8
```

Resolvemos el sistema con SOR:

```
>>> sol_s,it_s,res_s = sor(A,b,x0,1.2)
SOR termino satisfactoriamente
>>> print('Solución: {0}\nIteraciones: {1}'.format(sol_s.T,it_s))
Solución: [ 0.20189032  0.46056776  0.31230324  0.32018879  0.13880176]
Iteraciones: 8
```

En la Tabla 2.1 mostramos los resultados obtenidos. Reportamos para cada método el número de iteraciones (`it_j`, `it_gs`, `it_s`) y la solución hallada (`sol_j`, `sol_gs`, `sol_s`).

Tabla 2.1. Resultados obtenidos de Jacobi, Gauss-Seidel y SOR

Método	Iteraciones	Solución	e_m
Jacobi	16	(0.20188685, 0.46056149, 0.31230786, 0.32018455, 0.13880381)	2.453e-05
Gauss-Seidel	8	(0.20187623, 0.46055745, 0.31231221, 0.32018582, 0.1388045)	4.295e-05
SOR	8	(0.20189032, 0.46056776, 0.31230324, 0.32018879, 0.13880176)	3.870e-06

En la Tabla 2.1 también reportamos el error

$$e_m = \|\texttt{sol} - \texttt{solm}\|_1,$$

donde `sol` es la solución obtenida con el comando `solve` y `solm` es la solución hallada por el método. Este error lo calculamos de la siguiente manera:

```
>>> print('{0:.3e}'.format(np.linalg.norm(sol-sol_j,1)))
2.453e-05
>>> print('{0:.3e}'.format(np.linalg.norm(sol-sol_gs,1)))
4.295e-05
>>> print('{0:.3e}'.format(np.linalg.norm(sol-sol_s,1)))
3.870e-06
```

En la Figura 2.2 mostramos las gráficas de *Iteraciones* vs. $\|A\mathbf{x}_k - \mathbf{b}\|_1$ (en escala logarítmica del eje *y*) para cada uno de los métodos. Para la generación de estas gráficas, utilizamos el módulo `pyplot` de la librería `Matplotlib`:

```
>>> import matplotlib.pyplot as plt
>>> plt.semilogy(range(0,it_j+1),res_j, label='Jacobi')
>>> plt.semilogy(range(0,it_gs+1),res_gs, label='Gauss-Seidel')
>>> plt.semilogy(range(0,it_s+1),res_s, label='SOR')
>>> plt.legend()
>>> plt.xlabel('Iteraciones')
>>> plt.ylabel(r'$\Vert Ax_{k}-b\Vert_{1}$')
>>> plt.show()
```

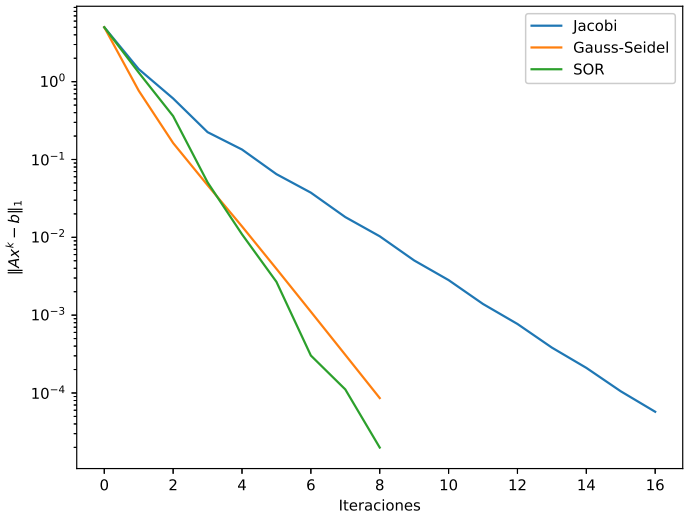


Figura 2.2. Convergencia de los métodos Jacobi, Gauss-Seidel y SOR

2.7.2 Métodos Basados en Subespacios de Krylov

Los métodos iterativos no estacionarios para sistemas de ecuaciones lineales $A\mathbf{x} = \mathbf{b}$ poseen la siguiente forma iterativa general. Dado $\mathbf{x}_0 \in \mathbb{R}^n$ un iterado inicial, los métodos no estacionarios generan sus iterados por la fórmula:

$$\mathbf{x}_k = \mathbf{x}_{k-1} + \alpha_{k-1}\mathbf{s}_{k-1}, \quad k = 1, 2, \ldots$$

donde el escalar $\alpha_k \in \mathbb{R}$ y el vector $\mathbf{s}_k \in \mathbb{R}^n$ se definen en cada iteración. Para el caso especial en que A es simétrica definida positiva, existen métodos no estacionarios muy conocidos como el método de Cauchy y el método de gradientes conjugados. Estos métodos los trataremos en el Capítulo 8. En esta sección describiremos métodos para el caso general que A es cualquier matriz no singular. Específicamente, estudiaremos métodos basados en subespacios de Krylov o, simplemente, métodos de Krylov. Cabe resaltar que el método de gradientes conjugados también es un método de Krylov, pero nosotros lo describiremos utilizando conceptos de optimización. Para un estudio más detallado de los métodos de Krylov se recomiendan los libros [29], [16], [10], [25].

Definición 2.8 (Subespacio de Krylov)**.** Sean A una matriz cuadrada de orden n, $\mathbf{v}$ un vector en $\mathbb{R}^n$ y m un entero positivo. El subespacio generado por el sistema de vectores $\{\mathbf{v}, A\mathbf{v}, A^2\mathbf{v}, \ldots, A^{m-1}\mathbf{v}\}$ se denomina subespacio de Krylov de orden m, denotado por $\mathscr{K}_m(A, \mathbf{v})$. En otras palabras,

$$\mathscr{K}_m(A, \mathbf{v}) = \text{span}(\mathbf{v}, A\mathbf{v}, A^2\mathbf{v}, \ldots, A^{m-1}\mathbf{v}).$$

Los métodos que estudiaremos buscan minimizar el residual, $\mathbf{r}_k = \mathbf{b} - A\mathbf{x}_k$, en la k-ésima iteración sobre el espacio afín

$$\mathscr{V}_k = \mathbf{x}_0 + \mathscr{K}_k(A, \mathbf{r}_0), \tag{2.22}$$

donde $\mathbf{x}_0$ es el iterado inicial y $\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$. El espacio afín $\mathscr{V}_k$ está conformado por vectores $\mathbf{u} = \mathbf{x}_0 + p(A)\mathbf{r}_0$, donde p es un polinomio cuyo grado no excede a $k-1$. En sí, los métodos generan soluciones aproximadas de la forma

$$\mathbf{x}_k = \mathbf{x}_0 + q(A)\mathbf{r}_0, \quad k = 1, 2, \ldots \tag{2.23}$$

donde q es un polinomio de grado $k-1$ seleccionado de tal manera que $\mathbf{x}_k$ sea, en un sentido que debe ser preciso, la mejor aproximación de $\mathbf{x}$ en $\mathscr{V}_k$. Un método que busca una solución de la forma (2.23) con $\mathscr{V}_k$ definido como en (2.22) se llama método de Krylov.

Una primera pregunta relacionada con los métodos de Krylov es si la dimensión del subespacio $\mathscr{K}_m$ aumenta a medida que crece m. Una respuesta parcial la proporciona el siguiente resultado.

Teorema 2.12. *Sea $A \in \mathbb{R}^{n\times n}$ y $\mathbf{v} \in \mathbb{R}^n$. El subespacio de Krylov $\mathscr{K}_m(A, \mathbf{v})$ tiene dimensión igual a m si, y sólo si el grado de $\mathbf{v}$ con respecto a A, denotado por $\deg_A(\mathbf{v})$, no es menor que m, donde el grado de $\mathbf{v}$ se define como el mínimo grado de un polinomio mónico no nulo p, para el cual $p(A)\mathbf{v} = 0$.*

La dimensión de $\mathscr{K}_m(A, \mathbf{v})$ es, por lo tanto, igual a $\ell = \min(m, \deg_A(\mathbf{v}))$ y, como consecuencia, la dimensión de los subespacios de Krylov es ciertamente una función no decreciente de m. Por el Teorema de Cayley-Hamilton (Teorema 1.6), se tiene que $\deg_A(\mathbf{v})$ no puede ser mayor que n.

Por otra parte, las soluciones de la forma (2.23) se pueden expresar equivalentemente como

$$\mathbf{x}_k = \mathbf{x}_0 + \mathbf{u}_k, \quad k = 1, 2, \ldots \tag{2.24}$$

donde $\mathbf{u}_k \in \mathscr{K}_k(A, \mathbf{r}_0)$. Este vector, a su vez, se puede expresar como una combinación lineal de una base $\{\mathbf{v}_1, \ldots, \mathbf{v}_{\ell_k}\}$ de $\mathscr{K}_k(A, \mathbf{r}_0)$, donde $\ell_k = \min(k, \deg_A(\mathbf{v}))$.

Para m fijo, es posible generar una base ortonormal de $\mathscr{K}_m(A, \mathbf{v})$ usando el *algoritmo de Arnoldi*. Tomando $\mathbf{v}_1 = \mathbf{v}/\|\mathbf{v}\|$, este método genera una base ortonormal $\{\mathbf{v}_1, \ldots, \mathbf{v}_\ell\}$ de $\mathscr{K}_m(A, \mathbf{v})$ usando el proceso de Gram-Schmidt (Teorema 1.1). Para $k = 1, \ldots, m$, el algoritmo de Arnoldi calcula

$$\begin{aligned} h_{jk} &= \mathbf{v}_j^T A \mathbf{v}_k, \qquad j = 1, \ldots, k, \\ \mathbf{w}_k &= A\mathbf{v}_k - \sum_{j=1}^{k} h_{ik}\mathbf{v}_j, \\ h_{k+1,k} &= \|\mathbf{w}_k\|. \end{aligned}$$

Si $h_{k+1,k} = 0$, entonces el proceso termina y en tal caso decimos que se ha producido un "*breakdown*" (parada forzada) del algoritmo indicando que se ha producido convergencia; de lo contrario, definimos $\mathbf{v}_{k+1} = \mathbf{w}_k/h_{k+1,k}$ y el algoritmo realiza otra iteración, incrementando k por 1. En el Algoritmo 2.5 se presenta la descripción formal del algoritmo de Arnoldi-Gram-Schmidt modificado o, simplemente, algoritmo de Arnoldi.

Algoritmo 2.5 (Algoritmo de Arnoldi-Gram-Schmidt modificado)**.**

Entrada: $\mathbf{v}_1 \in \mathbb{R}^n$ un vector de norma 1

```
for k = 1 : m do
    w_k = A v_k
    for j = 1 : k do
        h_jk = w_k^T v_j
        w_k = w_k − h_jk v_j
    end for
    h_{k+1,k} = ||w_k||
    if h_{k+1,k} = 0 then
        parar el proceso.
    else
        v_{k+1} = w_k / h_{k+1,k}
    end if
end for
```

Se puede demostrar que si el Algoritmo 2.5 termina en el paso m, entonces los vectores $\mathbf{v}_1, \ldots, \mathbf{v}_m$ forman una base de $\mathscr{K}_m(A, \mathbf{v})$. En este caso, si denotamos con $V_m \in \mathbb{R}^{n\times m}$ la matriz cuyas columnas son los vectores $\mathbf{v}_j$, entonces obtenemos

$$V_m^T A V_m = H_m, \qquad V_{m+1}^T A V_m = \widehat{H}_m,$$

donde $\widehat{H}_m \in \mathbb{R}^{(m+1)\times n}$ es una matriz de Hessenberg superior cuyos elementos están dados por $\widehat{h}_{ij} = h_{ij} = \mathbf{v}_i^T A \mathbf{v}_j$ y $H_m \in \mathbb{R}^{m\times m}$ es la matriz formada por las primeras m filas y m columnas de $\widehat{H}_m$. El Algoritmo 2.5 termina en un paso intermedio $k < m$ si, y sólo si $\deg_A(\mathbf{v}_1) = k$.

En el Programa 2.12 se muestra la implementación del Algoritmo 2.5. Para variantes computacionales más eficientes y estables del algoritmo de Arnoldi, puede consultar [29]. La salida de la

función `arnoldi` son las matrices $V \in \mathbb{R}^{n\times(m+1)}$ y $H \in \mathbb{R}^{(m+1)\times m}$. Las columnas de V conforman la base generada, mientras que H almacena los coeficientes $\widehat{h}_{ij}$ calculados por el algoritmo. Si la ejecución se detiene en m pasos, entonces $V_m = V$ y $H_m = H$. El algoritmo de Arnoldi realiza $2m^2n$ operaciones punto-flotante, es decir, es de orden $\mathbf{O}(mn^2)$.

Programa 2.12. Algoritmo de Arnoldi-Gram-Schmidt modificado

```
def arnoldi(A,v,m,tol=1.0e-12):
    v = np.asarray(v,dtype=float)
    n = A.shape[0]
    H = np.zeros((m + 1, m))
    V = np.zeros((n, m + 1))
    y = v / np.linalg.norm(v)
    V[:, 0] = y
    for k in range(m):
        w = A.dot(y)
        for j in range(k + 1):
            H[j, k] = np.dot(V[:,j], w)
            w = w - H[j, k]*V[:,j]
        H[k+1, k] = np.linalg.norm(w)
        if H[k+1, k] > tol:
            y = w / H[k + 1, k]
            V[:, k+1] = y
        else:
            return V, H
    return V, H
```

Ejemplo 2.28. Consideremos la matriz

$$A = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -10 & \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix}.$$

Dado $\mathbf{v} = (1,1,1,1)$, construyamos las bases ortonormales de los subespacios de Krylov $\mathscr{K}_2(A,\mathbf{v})$, $\mathscr{K}_3(A,\mathbf{v})$ y $\mathscr{K}_4(A,\mathbf{v})$. Se tiene:

```
>>> A = np.array([[2,-1,0,0],[-1,2,-1,0],[0,-1,2,-1],[0,0,-1,2]])
>>> v = np.ones(4)
>>> V,H = arnoldi(A,v,2)
>>> V
array([[ 0.5,  0.5,  0. ],
       [ 0.5, -0.5,  0. ],
       [ 0.5, -0.5,  0. ],
       [ 0.5,  0.5,  0. ]])
```

```
>>> H
array([[0.5 , 0.5],
       [0.5 , 2.5],
       [0.  , 0.  ]])
```

De esta forma, los vectores $\mathbf{v}_1 = \left(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}\right)$ y $\mathbf{v}_2 = \left(\frac{1}{2}, -\frac{1}{2}, -\frac{1}{2}, \frac{1}{2}\right)$ forman una base de $\mathscr{K}_2(A, \mathbf{v})$.

```
>>> V,H = arnoldi(A,v,3)
>>> V
array([[ 0.5 ,  0.5 ,  0.  ,  0.  ],
       [ 0.5 , -0.5 ,  0.  ,  0.  ],
       [ 0.5 , -0.5 ,  0.  ,  0.  ],
       [ 0.5 ,  0.5 ,  0.  ,  0.  ]])
>>> H
array([[0.5 , 0.5 , 0.  ],
       [0.5 , 2.5 , 0.  ],
       [0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  ]])
```

Los vectores $\mathbf{v}_1$ y $\mathbf{v}_2$ forman una base de $\mathscr{K}_3(A, \mathbf{v})$.

```
>>> V,H = arnoldi(A,v,4)
>>> V
array([[ 0.5 ,  0.5 ,  0.  ,  0.  ,  0.  ],
       [ 0.5 , -0.5 ,  0.  ,  0.  ,  0.  ],
       [ 0.5 , -0.5 ,  0.  ,  0.  ,  0.  ],
       [ 0.5 ,  0.5 ,  0.  ,  0.  ,  0.  ]])
>>> H
array([[0.5 , 0.5 , 0.  , 0.  ],
       [0.5 , 2.5 , 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.  ]])
```

Los vectores $\mathbf{v}_1$ y $\mathbf{v}_2$ forman una base de $\mathscr{K}_4(A, \mathbf{v})$. Observe que para $m = 2$, la tercera columna de la matriz V es nula; para $m \geq 3$, las columnas de las matrices V y H son todas nulas a partir de la tercera. Este hecho se debe a que el vector $\mathbf{v} = (1,1,1,1)$ tiene grado 2 con respecto a A, ya que $p(A)\mathbf{v} = 0$ con $p(A) = I_4 - 3A + A^2$ y no existe un polinomio mónico q de grado 1 para el cual se cumpla $q(A)\mathbf{v} = 0$. Por tanto, todos los subespacios de Krylov $\mathscr{K}_m(A, \mathbf{v})$ con $m \geq 2$, tienen dimensión $\ell = 2$.

■

Habiendo introducido un algoritmo para generar una base para el subespacio de Krylov $\mathscr{K}_m(A, \mathbf{r}_0)$, ahora podemos resolver el sistema lineal $A\mathbf{x} = \mathbf{b}$ mediante un método de Krylov. Como ya se notó, para todos estos métodos, la iteración $\mathbf{x}_k$ siempre tiene la forma (2.23) y, para un $\mathbf{r}_0$ dado, el vector $\mathbf{x}_k$ se selecciona como el único elemento en $\mathscr{V}_k$ que satisface un criterio de distancia mínima a $\mathbf{x}$. Por lo tanto, la característica que distingue dos métodos de Krylov diferentes es el criterio para seleccionar $\mathbf{x}_k$.

La idea más natural consiste en buscar $\mathbf{x}_k \in \mathscr{V}_k$ como el vector que minimiza la norma del residual, $\|\mathbf{r}_k\|$. Sin embargo, este enfoque no funciona muy bien en la práctica, ya que $\mathbf{x}_k$ dependería de la solución $\mathbf{x}$, que por supuesto no es conocida. Se pueden considerar dos alternativas:

(i) Calcular $\mathbf{x}_k \in \mathscr{V}_k$ forzando que el residual $\mathbf{r}_k$ sea ortogonal a $\mathscr{K}_k(A,\mathbf{r}_0)$, es decir, buscamos $\mathbf{x}_k \in \mathscr{V}_k$ tal que

$$\mathbf{v}^T(\mathbf{b}-A\mathbf{x}_k)=0, \qquad \text{para todo } \mathbf{v}\in\mathscr{K}_k(A,\mathbf{r}_0); \tag{2.25}$$

(ii) Calcular $\mathbf{x}_k \in \mathscr{V}_k$ minimizando la norma del residual residual $\|\mathbf{r}_k\|$, es decir, debemos resolver el problema de minimización

$$\mathbf{x}_k = \underset{\mathbf{v}\in\mathscr{K}_k(A,\mathbf{r}_0)}{\operatorname{argmin}} \|\mathbf{b}-A\mathbf{v}\|. \tag{2.26}$$

La alternativa (i) conduce al método de Arnoldi para sistemas de ecuaciones lineales, más conocido como FOM (por sus siglas en inglés, Full Orthogonalization Method), mientras que la alternativa (ii) proporciona el método GMRES (Generalized Minimum Residual). A continuación presentamos los métodos de FOM y GMRES. No daremos una explicación detallada de su derivación. Para un estudio formal de estos métodos y sus variantes, se recomienda el libro [29].

Asumamos que se ha generado una base ortonormal $\{\mathbf{v}_1,\ldots,\mathbf{v}_{\ell_k}\}$ ($\ell_k = \min(k,\deg_A(\mathbf{r}_0))$ del subespacio de Krylov $\mathscr{K}_k(A,\mathbf{r}_0)$, construida con el algoritmo de Arnoldi, además, asumamos que los vectores de esta base son las columnas de la matriz $V_k \in \mathbb{R}^{n\times\ell_k}$, con $\mathbf{v}_1 = \mathbf{r}_0/\|\mathbf{r}_0\|$. En este caso, el iterado $\mathbf{x}_k$ del método de Krylov se puede escribir como

$$\mathbf{x}_k = \mathbf{x}_0 + V_k\mathbf{s}_k, \tag{2.27}$$

donde $\mathbf{s}_k \in \mathbb{R}^{\ell_k}$ debe ser seleccionado de manera que se satisfaga (2.25) o (2.26).

Método FOM

La condición (2.25) implica que el residual $\mathbf{r}_k$ debe ser ortogonal a los vectores $\mathbf{v}_j$, esto es,

$$V_k^T\mathbf{r}_k = 0. \tag{2.28}$$

Como $\mathbf{r}_k = b - A\mathbf{x}_k$ y $\mathbf{x}_k$ está dado por (2.27), entonces por (2.28) obtenemos:

$$\begin{aligned} V_k^T(b-A\mathbf{x}_k) &= V_k^T(b-A\mathbf{x}_0-AV_k\mathbf{s}_k)\\ &= V_k^T(\mathbf{r}_0-AV_k\mathbf{s}_k)\\ &= V_k^T\mathbf{r}_0 - V_k^TAV_k\mathbf{s}_k = 0. \end{aligned} \tag{2.29}$$

Ahora, la base $\{\mathbf{v}_1,\ldots,\mathbf{v}_{\ell_k}\}$ es ortogonal y $\mathbf{v}_1 = \mathbf{r}_0/\|\mathbf{r}_0\|$, así tenemos:

$$V_k^T\mathbf{r}_0 = \|\mathbf{r}_0\|\mathbf{e}_1, \tag{2.30}$$

donde $\mathbf{e}_1 = (1,0,\ldots,0) \in \mathbb{R}^{\ell_k}$. Luego, como $V_k^TAV_k = H_k \in \mathbb{R}^{\ell_k\times\ell_k}$, entonces por (2.29) y (2.30) obtenemos:

$$H_k\mathbf{s}_k = \|\mathbf{r}_0\|\mathbf{e}_1. \tag{2.31}$$

En otras palabras, $\mathbf{s}_k$ se puede obtener como la solución del sistema de ecuaciones lineales (2.31). La matriz H_k es una matriz de Hessenberg superior, por ello el sistema (2.31) se puede resolver fácilmente, por ejemplo, a través de la eliminación gaussiana con sus-titución regresiva. De esta forma, el método FOM genera una sucesión de iterados $\{\mathbf{x}_k\}$ de la forma (2.27), donde $\mathbf{s}_k$ es solución del sistema de ecuaciones lineales (2.31), para cada $k \geq 1$. En su forma básica, FOM no requiere un cálculo explícito de la aproximación $\mathbf{x}_k$ o del residual $\mathbf{r}_k$. Por lo tanto, monitorear su convergencia calculando el residual en cada iteración puede ser costoso. Sin embargo, el residual

está disponible sin necesidad de calcular $\mathbf{x}_k$. Se puede establecer que en la k-ésima iteración de FOM se tiene:

$$\mathbf{b} - A\mathbf{x}_k = -h_{k+1,k}\mathbf{e}_k^T\mathbf{s}_k\mathbf{v}_{k+1}$$

y

$$\|\mathbf{b} - A\mathbf{x}_k\| = h_{k+1,k}|\mathbf{e}_k^T\mathbf{s}_k|.$$

Por tanto, uno puede definir, por ejemplo, el siguiente criterio de parada

$$h_{k+1,k}|\mathbf{e}_k^T\mathbf{s}_k|/\|\mathbf{r}_0\| < \varepsilon, \tag{2.32}$$

con $\varepsilon > 0$ una tolerancia fija.

En el siguiente teorema se da un resultado de convergencia del método FOM.

Teorema 2.13. *Sea A una matriz no singular. Entonces, en aritmética exacta, el método FOM halla la solución del sistema de ecuaciones lineales $A\mathbf{x} = \mathbf{b}$ en un máximo de n iteraciones.*

Una consecuencia del Teorema 2.13 es que FOM puede ser considerado un método directo, es decir, halla la solución en un número finito de iteraciones. Esto está totalmente alejado de la realidad. En la práctica, cuando se utiliza aritmética punto-flotante, no existe garantía que FOM termine en n iteraciones. Aún más, si tomamos en cuenta que el número de operaciones punto-flotante de FOM en m iteraciones es de orden $\mathbf{O}(mn^3)$, y que el espacio en memoria necesario para almacenar la matriz V_m es elevado, podemos concluir que FOM no se puede utilizar en la práctica, excepto para valores pequeños de m. Entonces, ¿qué se debe hacer para utilizar FOM? Sabemos que el mayor valor de m es n, pero no es conveniente cuando n es grande. Hay dos enfoques posibles. El primero de ellos es reiniciar el algoritmo periódicamente y el segundo es "truncar" la ortogonalización en el algoritmo de Arnoldi. Nosotros consideraremos el primer enfoque; el segundo no lo trataremos aquí (ver [29]).

El procedimiento de reiniciar FOM periódicamente se denomina FOM(m) o FOM *reiniciado*, que consiste en: generar una base de m vectores usando el algoritmo de Arnoldi, tomando m un número fijo pequeño (usualmente, $m \simeq 10$); luego calcular $\mathbf{x}_m$ a partir de la forma básica de FOM, si el método no converge, se toma $\mathbf{x}_0 = \mathbf{x}_m$ y se aplica nuevamente el algoritmo de Arnoldi para calcular un nuevo $\mathbf{x}_m$. Este procedimiento se lleva a cabo hasta lograr la convergencia. El Algoritmo 2.6 muestra el método FOM reiniciado.

Algoritmo 2.6 (Método FOM reiniciado (FOM(m))).

1: Calcule $\mathbf{r}_0 = b - A\mathbf{x}_0$, $\beta = \|\mathbf{r}_0\|$ y $\mathbf{v}_1 = \mathbf{r}_0/\beta$.
2: Genere la base $\{\mathbf{v}_1, \ldots, \mathbf{v}_m\}$ y la matriz H_m, usando el algoritmo de Arnoldi comenzando con $\mathbf{v}_1$.
3: Calcule $\mathbf{s}_m = H_m^{-1}\beta\mathbf{e}_1$ y $\mathbf{x}_m = \mathbf{x}_0 + V_m\mathbf{s}_m$.
4: Detenga el proceso si se satisface el criterio de parada.
5: Tome $\mathbf{x}_0 = \mathbf{x}_m$ e ir a 1.

El Programa 2.13 proporciona una implementación del algoritmo FOM reiniciado, con un criterio de parada dado por (2.32). El parámetro de entrada m es el tamaño máximo del subespacio de Krylov que se está generando y representa, como consecuencia, el número máximo de iteraciones para el reinicio (restart). `FOM_m` utiliza la función `arnoldi` implementada en el Programa 2.12.

Programa 2.13. Método FOM reiniciado

```
def FOM_m(A, b, x0=None, m=10, tol=1.0e-8):
    if x0 == None:
        x0 = np.zeros(A.shape[0])
    else:
        x0 = np.asarray(x0, dtype=float)
    b = np.asarray(b, dtype=float)
    e1 = np.zeros(m); e1[0] = 1
    res = []
    r = b - A.dot(x0)
    k = 0
    nr_0 = np.linalg.norm(r)
    res.append(nr_0)
    if nr_0 < tol: return x0, k, res
    beta = nr_0
    while True:
        v = r / beta
        V,H = arnoldi(A, v, m)
        sm = np.linalg.solve(H[:m,:m], beta*e1)
        x = x0 + V[:,:m].dot(sm)
        beta = (H[m, m-1] * sm[m - 1])
        r = -beta*V[:,m]
        beta = np.abs(beta)
        res.append(beta)
        k += 1
        if beta/nr_0 > tol:
            x0 = x
        else:
            return x, k, res
```

Ejemplo 2.29. Consideremos la matriz

$$A = \begin{bmatrix} T & -I & & \\ -I & T & \ddots & \\ & \ddots & \ddots & -I \\ & & -I & T \end{bmatrix}$$

donde $I \in \mathbb{R}^{N\times N}$ es la matriz identidad y $T \in \mathbb{R}^{N\times N}$ es una matriz tridiagonal dada por

$$T = \begin{bmatrix} 4 & -1 & & \\ -1 & 4 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 4 \end{bmatrix}.$$

La matriz A es una matriz cuadrada de orden $n = N^2$ denominada *matriz de poisson*, que aparece en muchas aplicaciones relacionadas con ecuaciones diferenciales en derivadas parciales que involucran la discretización del Laplaciano en 2D. A es simétrica definida positiva, además, es una matriz rala (sparse), ya que posee muchos elementos nulos. Utilicemos el método FOM reiniciado para encontrar una solución del sistema de ecuaciones lineales $A\mathbf{x} = b$, donde $b = (1,1,\ldots,1) \in \mathbb{R}^n$. La siguiente función genera la matriz de poisson de orden N^2.

```
def matrix_poisson(N):
    from scipy.sparse import spdiags
    e = np.ones(N**2)
    e2 = ([1] * (N-1) + [0]) * N
    e3 = ([0] + [1]*(N-1)) * N
    return spdiags([-4*e, e2, e3, e, e], [0, -1, 1, -N, N], N**2, N**2)
```

La Tabla 2.2 muestra los resultados de aplicar `FOM_m`, con $m = 10$, para matrices de poisson de distintos tamaños. Por defecto la función `FOM_m` toma $\mathbf{x}_0 = 0$ como iterado inicial. La columna etiquetada con Iter muestra el número total de iteraciones requeridas para la convergencia. Se tomó el criterio de parada (2.32) con una tolerancia de 10^{-8}. La columna etiquetada con Residual muestra la norma del residual en la solución hallada. Note que el método requiere de mayor número de iteraciones a medida que el orden de A aumenta, además, la norma del residual crece cuando n aumenta.

n	Iter	Residual
100	3	9.310e-08
900	34	2.887e-07
2500	105	3.979e-07
10000	403	9.525e-07

Tabla 2.2. Una prueba del método FOM reiniciado

Método GMRES

El método GMRES se caracteriza por seleccionar $\mathbf{x}_k$ de manera que minimice la norma del residual, $\|b - A\mathbf{x}_k\|$, en la k-ésima iteración. Por (2.27), podemos escribir:

$$\mathbf{r}_k = \mathbf{r}_0 - AV_k\mathbf{s}_k, \tag{2.33}$$

pero $\mathbf{r}_0 = \mathbf{v}_1/\|\mathbf{r}_0\|$ y (2.26) se debe satisfacer, entonces la Ecuación (2.33) adquiere la forma

$$\mathbf{r}_k = V_{k+1}\left(\|\mathbf{r}_0\|\mathbf{e}_1 - \widehat{H}_k\mathbf{s}_k\right), \tag{2.34}$$

donde $\mathbf{e}_1 = (1,0,\ldots,0) \in \mathbb{R}^{\ell_k+1}$. Por tanto, en el método GMRES la aproximación $\mathbf{x}_k$ se debe calcular usando (2.27) tal que:

$$\mathbf{s}_k \text{ se escoge de manera que minimice la norma del residual } \|\|\mathbf{r}_0\|\mathbf{e}_1 - \widehat{H}_k\mathbf{s}_k\|.$$

Recuerde que la matriz V_{k+1} es ortogonal, por lo que se cumple $\|V_{k+1}\mathbf{y}\| = \|\mathbf{y}\|$, para todo vector $\mathbf{y}$. Como GMRES debe resolver en cada iteración un problema de cuadrados mínimos de tamaño $\ell_k \leq n$, éste será más efectivo cuanto más pequeño sea el número de iteraciones que realice. Igual que el método FOM, GMRES termina a lo sumo en n iteraciones, produciendo la solución exacta. Para ser más precisos, tenemos el siguiente resultado de convergencia.

Teorema 2.14. *Sea A una matriz no singular. Entonces, el método GMRES se detiene en una iteración m (con $m < n$) si, y sólo si la aproximación x_m hallada coincide con la solución exacta.*

Para mejorar la eficiencia del algoritmo GMRES es necesario idear un criterio de parada que no requiera la evaluación explícita del residuo en cada paso. Esto es posible, siempre que el sistema lineal con la matriz de Hessenberg superior $\widehat{H}_k$ se resuelva adecuadamente.

Como FOM, el número de operaciones punto-flotante de GMRES en m iteraciones es aproximadamente de orden $\mathbf{O}(mn^3)$, además, requiere de un elevado almacenamiento en memoria, a menos que se produzca una convergencia después de algunas iteraciones. Por este motivo, están disponibles dos variantes del algoritmo, una llamada GMRES reiniciado o GMRES(m); y la otra llamada Quasi-GMRES o QGMRES (ver [29]), que está basada en detener el proceso de ortogonalización de Arnoldi. Cabe resaltar que estas dos variantes no poseen la propiedad del Teorema 2.14.

El Algoritmo 2.7 muestra el método GMRES reiniciado, cuya implementación se proporciona en el Programa 2.14. Aquí se usa la función `lstsq` de la librería `Scipy` para resolver el problema de cuadrados mínimos

$$\min_{\mathbf{w}\in\mathscr{K}_m(A,\mathbf{r}_0)} \|\|\mathbf{r}_0\|\mathbf{e}_1 - \widehat{H}_m\mathbf{w}\|^2,$$

además, se emplea el criterio de parada

$$\frac{\|\mathbf{r}_k\|}{\|\mathbf{r}_0\|} \leq \varepsilon,$$

con $\varepsilon > 0$ una tolerancia fija y $\mathbf{r}_k$ dado por (2.34).

Algoritmo 2.7 (Método GMRES reiniciado (GMRES(m)))**.**

1: Calcule $\mathbf{r}_0 = b - A\mathbf{x}_0$, $\beta = \|\mathbf{r}_0\|$ y $\mathbf{v}_1 = \mathbf{r}_0/\beta$.
2: Genere la base $\{\mathbf{v}_1,\ldots,\mathbf{v}_m\}$ y la matriz $\widehat{H}_m$, usando el algoritmo de Arnoldi comenzando con $\mathbf{v}_1$.
3: Calcule $\mathbf{s}_m$ resolviendo el problema de cuadrados mínimos $\min_{\mathbf{s}\in\mathscr{K}_m(A,\mathbf{r}_0)} \|\|\mathbf{r}_0\|\mathbf{e}_1 - \widehat{H}_m\mathbf{s}\|^2$.
4: Calcule $\mathbf{x}_m = \mathbf{x}_0 + V_m\mathbf{s}_m$.
5: Detenga el proceso si se satisface el criterio de parada.
6: Tome $\mathbf{x}_0 = \mathbf{x}_m$ e ir a 1.

Programa 2.14. Método GMRES reiniciado

```
def gmres_m(A, b, x0=None, m=10, tol=1.0e-8):
    if x0 == None:
        x0 = np.zeros(A.shape[0])
    else:
        x0 = np.asarray(x0, dtype=float)
    b = np.asarray(b, dtype=float)
    e1 = np.zeros(m+1); e1[0] = 1
    res = []
    r = b - A.dot(x0)
    k = 0
    nr_0 = np.linalg.norm(r)
    res.append(nr_0)
    if nr_0 < tol:
        return x0, k, res
    beta = nr_0
```

```
    while True:
        v = r / beta
        be1 = beta*e1
        V,H = arnoldi(A, v, m)
        sal = np.linalg.lstsq(H, be1, rcond=None)
        sm = sal[0]
        x = x0 + V[:,:m].dot(sm)
        r = V.dot(be1-H.dot(sm))
        beta = np.linalg.norm(r)
        res.append(beta)
        k += 1
        if beta/nr_0 > tol:
            x0 = x
        else:
            return x, k, res
```

Ejemplo 2.30. Veamos el comportamiento del método GMRES en la resolución del sistema de ecuaciones lineales $A\mathbf{x} = \mathbf{b}$ del Ejemplo 2.29. La Tabla 2.3 muestra los resultados de aplicar `gmres_m`, con $\mathbf{x}_0 = 0$, $m = 10$ y `tol=1e-8`, para matrices de poisson de distintos tamaños. Observe que el método requiere de mayor número de iteraciones a medida que el orden de A aumenta, además, la norma del residual crece cuando n aumenta.

n	Iter	Residual
100	4	1.757e-09
900	40	2.035e-07
2500	101	4.343e-07
10000	380	9.532e-07

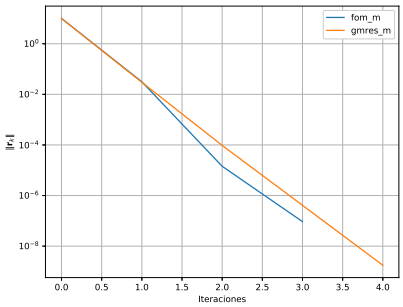
Tabla 2.3. Una prueba del método GMRES

En la Figura 2.3 se muestra el comportamiento del residual como una función del número de iteraciones para los métodos FOM(m) y GMRES(m) aplicados al sistema lineal $A\mathbf{x} = \mathbf{b}$.
Observe que, para n pequeño, FOM(m) converge más rápido que GMRES(m); en cambio, para $n > 900$, GMRES(m) converge más rápidamente. Además, note que la convergencia de ambos métodos es lineal.

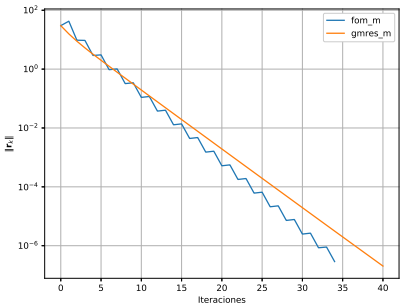
Una implementación eficiente del método GMRES está disponible en `sparse.linalg` de la librería `Scipy`, a través de la función:

```
x,info = gmres(A,b,x0=None,tol=1e-05,restart=None,maxiter=None,M=none),
```

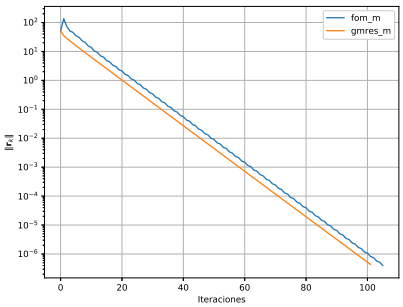
donde `x0=None` indica que el iterado inicial es opcional, es decir, si el usuario no suministra un iterado inicial, se escogerá $\mathbf{x}_0$ como el vector nulo; el parámetro `tol` (opcional) es la tolerancia, por defecto es 10^{-5}; el parámetro `restart` (opcional) es el número m de reinicio, por defecto se toma $m = 20$; el parámetro `M` (opcional) es la inversa del precondicionador de A (ver Sección 2.8); y `maxiter` (opcional) es el número máximo de iteraciones. La fun-ción `gmres` arroja como salida



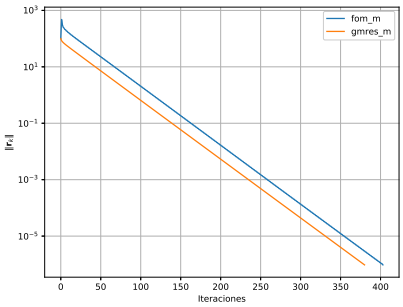
(a) Matriz de poisson de orden $n = 100$.



(b) Matriz de poisson de orden $n = 900$.



(c) Matriz de poisson de orden $n = 2500$.



(d) Matriz de poisson de orden $n = 10000$.

Figura 2.3. Comportamiento del residual como una función del número de iteraciones para los métodos FOM(m) y GMRES(m) aplicados al sistema lineal $A\mathbf{x} = \mathbf{b}$ del Ejemplo 2.29, con matrices de poisson de distintos tamaños. La línea azul representa FOM(m) y la línea naranja es GMRES(m).

la solución `x` y un entero `info` que provee información de la convergencia:

$$\texttt{info} \begin{cases} = 0, & \text{convergencia alcanzada;} \\ > 0, & \text{convergencia no alcanzada, número de iteraciones;} \\ < 0, & \text{breakdown (salida abrupta).} \end{cases}$$

Para mayor información de la función `gmres` consulte el link:
https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.sparse.linalg.gmres.html

2.8 Precondicionamiento

Como mencionamos en la Sección 1.3.3, el condicionamiento es una de las características que tiene un gran efecto en la eficiencia y precisión de un algoritmo. Si un problema está mal condicionado, un pequeño error en los datos produce un error relativo grande en la solución. Para sistemas de ecuaciones lineales $A\mathbf{x} = \mathbf{b}$, el condicionamiento depende del número de condición de la matriz A (ver Sección 1.3.3), $\kappa_2(A) = \|A\|\|A^{-1}\|$. Si $\kappa_2(A)$ es grande, se dice que el sistema lineal está mal-condicionado; de lo contrario, se dice que está bien-condicionado.

Cuando un sistema lineal está mal-condicionado, la velocidad de convergencia de los métodos iterativos es lenta y obtienen, generalmente, una solución de baja precisión. Existen técnicas que permiten mejorar el comportamiento de los métodos iterativos, una de ellas es el *precondicionamiento*, la cual se basa en lo siguiente. Sea $P \in \mathbb{R}^{n\times n}$ una matriz no singular. Multiplicando por la

inversa de P en ambos lados de la ecuación $A\mathbf{x} = \mathbf{b}$, obtenemos el sistema,

$$P^{-1}A\mathbf{x} = P^{-1}\mathbf{b}, \tag{2.35}$$

denominado *sistema precondicionado*, que tiene la misma solución del sistema original. La matriz P se denomina *matriz de precondicionamiento* o *precondicionador*. Se debe tener presente que el sistema precondicionado puede ser un sistema denso (la matriz $P^{-1}A$ pudiera tener muchos elementos no nulos). De hecho, no hay razón por la que P^{-1} deba ser una matriz dispersa (aunque P sea dispersa, esto es, con muchos elementos nulos), ya que la inversa de una matriz dispersa no es necesariamente dispersa. Esto limita el número de técnicas que se pueden aplicar para resolver el sistema precondicionado ([29]).

Construir un precondicionador no es una tarea fácil, depende generalmente de la estructura de la matriz A (en algunos casos también depende de las condiciones o leyes que rigen el modelo físico). Existen precondicionadores generales que se obtienen fácilmente y los mismos dependen de los elementos de la matriz y están estrechamente relacionados con los métodos iterativos Jacobi, Gauss-Seidel y SOR, a saber:

$$P_J = D, \qquad P_{GS} = D - E, \qquad P_{SOR} = \frac{1}{\omega}(D - \omega E),$$

denominados precondicionador de Jacobi, precondicionador de Gauss-Seidel y precondicionador SOR, respectivamente. El precondicionador de Jacobi es simplemente la diagonal de A (por supuesto, todos los elementos de la diagonal de A deben ser no nulos), el precondicionador de Gauss-Seidel es la parte inferior de A. El número ω del precondicionador SOR solo tiene el efecto de escalar las ecuaciones del sistema precondicionado de manera uniforme. Por lo tanto, no es importante en el contexto de precondicionamiento.

Existe otro grupo de precondicionadores que se obtienen a través de la *factorización LU incompleta* de la matriz A. No explicaremos su construcción porque está fuera del alcance de este libro. Para una descripción formal de la factorización LU incompleta se recomienda el libro [29]. La factorización LU incompleta de A consiste en hallar matrices L triangular inferior unitaria y U triangular superior, de manera que el residual $R = LU - A$ satisfaga ciertas restricciones. Con las matrices L y U se construye el precondicionador,

$$P_{ILU} = LU,$$

denominado *precondicionador ILU*.

Cuando A es una matriz simétrica definida positiva, se suele utilizar un precondicionador construido con la *factorización de Cholesky incompleta* (ver [29]), que consiste en hallar una matriz L triangular inferior tal que el residual $R = LL^T - A$ satisfaga ciertas restricciones. Con la matriz L se construye el precondicionador,

$$P_{IC} = LL^T,$$

denominado *precondicionador de Cholesky*. Para una descripción más extensa y formal de las técnicas de precondicionamiento se recomienda la referencia [29].

Mostremos con un ejemplo el efecto de un precondicionador en el comportamiento de los métodos iterativos. Cuando un método iterativo utiliza un técnica de precondicionamiento, se dice que el método está precondicionado.

Ejemplo 2.31. Consideremos el sistema de ecuaciones lineales $A\mathbf{x} = \mathbf{b}$ del Ejemplo 2.29. La matriz de poisson es una matrix mal-condicionada; por ejemplo, para $n = 2500$, el número de condición de A es $\kappa_2(A) \simeq 1.5315 \times 10^3$. En los experimentos numéricos utilizamos el vector del lado derecho $\mathbf{b} = (1, \ldots, 1) \in \mathbb{R}^n$, con $n = 900$. La siguiente función genera la inversa del precondicionador P_{ILU}, para una matriz A.

```
def inv_P_ILU(A, drop_tol=.25):
    from scipy.sparse.linalg import spilu
    from scipy.sparse import csc_matrix
    n = A.shape[0]
    I = np.eye(n)
    Inv_P_ILU = np.zeros((n,n))
    M2 = spilu(csc_matrix(A), drop_tol=drop_tol)
    M_x = lambda x: M2.solve(x)
    for i in range(n):
        Inv_P_ILU[:,i] = M_x(I[:,i])
    return Inv_P_ILU
```

Con el método GMRES implementado en el Programa 2.14, resolveremos el sistema original $A\mathbf{x} = \mathbf{b}$ y, para cada precondicionador P, también resolveremos el sistema precondicionado $P^{-1}A\mathbf{x} = P^{-1}\mathbf{b}$. La Tabla 2.4 muestra los resultados de la aplicación del método GMRES con los precondicionadores P_J, P_{GS}, P_{SOR} (con $\omega = 1.8$) y P_{ILU}. En esta tabla también se aprecian los resultados de GMRES sin precondicionamiento.

Método	Iter	Residual
GMRES	75	2.405e-12
GMRES-P_J	75	2.405e-12
GMRES-P_{GS}	31	1.617e-12
GMRES-P_{SOR}	31	1.600e-12
GMRES-P_{ILU}	11	2.029e-12

Tabla 2.4. Una prueba del método GMRES sin precondicionamiento y con los precondicionadores P_J, P_{GS}, P_{SOR} (con $\omega = 1.8$) y P_{ILU}.

Observe que GMRES converge con y sin precondicionador. El número de iteraciones se reduce sustancialmente cuando el método emplea un precondicionador, salvo para el precondicionador de Jacobi, con el cual se obtiene el mismo número de iteraciones que con la versión sin precondicionamiento. Note también que la calidad del residual hallado es prácticamente la misma en todos los casos.

Los métodos iterativos no resuelven directamente el sistema precondicionado, como se hizo en el ejemplo anterior. La mayoría de las implementaciones incorporan el precondicionamiento. En general los métodos descritos solo requieren productos matriz-vector; por ello, para calcular $\mathbf{w} = P^{-1}A\mathbf{v}$ para un vector dado $\mathbf{v}$, no es necesario calcular la inversa de P, sino que primero se calcula $\mathbf{y} = A\mathbf{v}$ y luego se resuelve el sistema $P\mathbf{w} = \mathbf{y}$. Las implementaciones de los métodos iterativos incorporan esta técnica para el uso de precondicionadores y el usuario sólo debe suministrar el precondicionador.

2.9 Sistemas Sobredeterminados

La mayoría de los métodos desarrollados hasta ahora están diseñados para resolver sistemas de ecuaciones lineales cuadrados, es decir, sistemas con matriz de coeficientes cuadrada. En esta sección le prestaremos atención a los sistemas de ecuaciones lineales

$$A\mathbf{x} = \mathbf{b}, \tag{2.36}$$

donde $A \in \mathbb{R}^{m\times n}$ con $m > n$, $\mathbf{x} \in \mathbb{R}^n$ y $\mathbf{b} \in \mathbb{R}^m$, es decir, sistemas que tienen más ecuaciones que incógnitas. Tales sistemas se denominan *sobredeterminados*. En general, con los n parámetros del vector solución $\mathbf{x}$, no es suficiente para expresar al vector $\mathbf{b}$ como una combinación lineal de las n columnas de A. En otras palabras, para un sistema sobredeterminado usualmente no se determina la solución en el sentido habitual (multiplicando por la inversa de A, que en este caso no está definida). En su lugar, se busca la solución $\mathbf{x}$ que minimiza la norma del vector residual $\mathbf{r} = \mathbf{b} - A\mathbf{x}$, para cierta norma $\|\cdot\|$. Cuando se usa la norma-2 y se minimiza la norma del residual al cuadrado, $\|\mathbf{b} - A\mathbf{x}\|_2^2$, se obtiene el método de *cuadrados mínimos*, que veremos con detalle en el Capítulo 3.

Existen otras técnicas, además de cuadrados mínimos, que permiten hallar la solución de (2.36). Supongamos que el sistema (2.36) tiene al menos una solución (esto es, $\mathbf{b} \in \text{alc}(A)$). Multiplicando por A^T en ambos lados de (2.36), obtenemos

$$(A^T A)\mathbf{x} = A^T \mathbf{b}. \tag{2.37}$$

Note que $A^T A \in \mathbb{R}^{n\times n}$ y $A^T\mathbf{b} \in \mathbb{R}^n$; además, $A^T A$ es una matriz simétrica semidefinida positiva. Luego, una solución del sistema (2.37) se puede hallar con cualquiera de los métodos vistos, pero existe un problema, la matrix $A^T A$ está mal-condicionada. Por ello, no es recomendable resolver directamente el sistema (2.37), cuando m y n son grandes.

Bajo ciertas condiciones, la matriz $A^T A$ es no singular. Luego, una solución del sistema (2.37) se obtiene por:

$$\mathbf{x} = (A^T A)^{-1} A^T \mathbf{b}.$$

En general, una solución del sistema sobredeterminado (2.36) se puede obtener como

$$\mathbf{x} = A^+ \mathbf{b},$$

donde $A^+ \in \mathbb{R}^{n\times n}$ se denomina *pseudoinversa* de A, la cual se define con la SVD de A.

Definición 2.9. Sea $A = U\Sigma V^T$ la SVD de $A \in \mathbb{R}^{m\times n}$, con $m \geq n$. La *pseudoinversa* de A es la matriz

$$A^+ = V\Sigma^+ U^T, \tag{2.38}$$

donde $\Sigma^+ \in \mathbb{R}^{n\times m}$ es una matriz diagonal, con elementos en la diagonal principal dados por:

$$\sigma_{jj}^+ = \begin{cases} 1/\sigma_j, & j = 1,\ldots,r, \\ 0, & j = r+1,\ldots,n, \end{cases}$$

y $r = \text{rang}(A) \leq n$.

Demostremos que si $\text{rang}(A) = n$, entonces $A^+ = (A^T A)^{-1} A^T$. En este caso, la matriz $A^T A$ es no singular. Como $U^T U = I$, tenemos

$$A^T A = (U\Sigma V^T)^T (U\Sigma V^T) = V\Sigma^T U^T U\Sigma V^T = V\Sigma^T \Sigma V^T.$$

De esta forma, empleando la ecuación anterior y el hecho que $V^T V = I$, podemos escribir:

$$\begin{aligned} (A^T A)^{-1} A^T &= (V\Sigma^T \Sigma V^T)^{-1} (U\Sigma V^T)^T \\ &= V(\Sigma^T \Sigma)^{-1} V^T V \Sigma^T U^T \\ &= V(\Sigma^T \Sigma)^{-1} \Sigma^T U^T. \end{aligned} \tag{2.39}$$

Como $(\Sigma^T\Sigma)^{-1}$ es una matriz diagonal de orden $n\times n$, con elementos en la diagonal principal dados por σ_j^{-2}, $j=1,\ldots,n$, entonces

$$(\Sigma^T\Sigma)^{-1}\Sigma^T=\Sigma^+. \tag{2.40}$$

Por (2.39) y (2.40), se concluye que $A^+=(A^TA)^{-1}A^T$, cuando $\text{rang}(A)=n$.

Ejemplo 2.32. Resolvamos el sistema sobredeterminado $A\mathbf{x}=\mathbf{b}$, donde

$$A=\begin{bmatrix}2&3\\1&-1\\2&2\\4&1\end{bmatrix},\qquad \mathbf{b}=\begin{bmatrix}1\\3\\2\\7\end{bmatrix}.$$

Se tiene:

$$A^T=\begin{bmatrix}2&1&2&4\\3&-1&2&1\end{bmatrix},\quad A^TA=\begin{bmatrix}25&13\\13&15\end{bmatrix}.$$

Como A^TA es no singular, entonces la pseudoinversa de A es

$$A^+=(A^TA)^{-1}A^T=\begin{bmatrix}-\frac{9}{206}&\frac{14}{103}&\frac{2}{103}&\frac{47}{206}\\[2mm]\frac{49}{206}&-\frac{19}{103}&\frac{12}{103}&-\frac{27}{206}\end{bmatrix}.$$

Por tanto, una solución del sistema es

$$\mathbf{x}=(A^TA)^{-1}A^T\mathbf{b}=\begin{bmatrix}-\frac{9}{206}&\frac{14}{103}&\frac{2}{103}&\frac{47}{206}\\[2mm]\frac{49}{206}&-\frac{19}{103}&\frac{12}{103}&-\frac{27}{206}\end{bmatrix}\begin{bmatrix}1\\3\\2\\7\end{bmatrix}=\begin{bmatrix}2\\-1\end{bmatrix}.$$

Por supuesto, la solución la hallamos en aritmética exacta, ya que el sistema es pequeño. Pero, en las aplicaciones esto, en general, no es posible. Hallemos la solución del sistema empleando la SVD de A. Se tiene:

```
>>> import numpy as np
>>> A = np.array([[2,3],[1,-1],[2,2],[4,1]])
>>> b = np.array([1,3,2,7])
>>> U,S,Vt = np.linalg.svd(A)
>>> print(S)
[5.824808    2.46406407]
```

Como los valores singulares de A son $\sigma_1=5.824808$ y $\sigma_2=2.46406407$, entonces el rango de A es $\text{rang}(A)=2$ y la matriz $\Sigma^+\in\mathbb{R}^{2\times 4}$ se define como:

```
>>> S_plus = np.zeros((2,4)); S_plus[:,0:2]=np.diag(1/S)
>>> print(S_plus)
[[0.17167948 0.         0.         0.        ]
 [0.         0.4058336  0.         0.        ]]
```

Ahora, la pseudoinversa de A, $A^+=V\Sigma^+U^T$, está dada por:

```
>>> A_plus = Vt.T.dot(S_plus.dot(U.T))
>>> print(A_plus)
[[-0.04368932  0.13592233  0.01941748  0.22815534]
 [ 0.23786408 -0.18446602  0.11650485 -0.13106796]]
```

Así, la solución del sistema sobredeterminado está dada por:

```
>>> x = A_plus.dot(b)
>>> print(x)
[ 2. -1.]
```

Calcular la pseudoinversa resulta costoso, ya que es necesario hallar la descomposición en valores singulares de A. Por ello, el uso de la pseudoinversa para resolver el sistema (2.36) dependerá de las propiedades de A (si es simétrica, diagonalemnte dominante, definida positiva, etc.). Existen otras técnicas para resolver sistemas sobredeterminados, pero están fuera del alcance de este libro. Para un estudio de estas técnicas se recomienda el libro [12].

2.10 Ejercicios

1. Resuelva los siguientes sistemas lineales usando (i) el método de eliminación Gaussiana y (ii) el método de eliminación Gaussiana con pivoteo parcial.

a)

$$\begin{cases} 2x_1 - 3x_2 + 100x_3 = 1 \\ x_1 + 10x_2 - 0.001x_3 = 0 \\ 3x_1 - 100x_2 + 0.01x_3 = 0 \end{cases}$$

b)

$$\begin{cases} x_1 + 20x_2 - x_3 + 0.001x_4 = 0 \\ 2x_1 - 5x_2 + 30x_3 - 0.1x_4 = 1 \\ 5x_1 + x_2 - 100x_3 - 10x_4 = 0 \\ 2x_1 - 100x_2 - x_3 + x_4 = 0 \end{cases}$$

c)

$$\begin{cases} x_1 - x_2 = -1 \\ x_1 + 2x_2 - x_3 = -2 \\ x_2 + 3x_3 - x_4 = -3 \\ x_3 + 4x_4 - x_5 = -4 \\ \ddots \quad \vdots \\ x_{48} + 49x_{49} - x_{50} = -49 \\ x_{49} + 50x_{50} = -50 \end{cases}$$

2. Resuelva los siguientes sistemas lineales dos veces. Primero, encuentre la factorización $A = LU$ y con ella resuelva el sistema. Después, encuentre la factorización $PA = LU$ y con ella resuelva el sistema.

a)

$$\begin{bmatrix} -1 & 1 & -4 \\ 2 & 2 & 0 \\ 3 & 3 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ \frac{1}{2} \end{bmatrix}$$

b)

$$\begin{bmatrix} 1 & 6 & 0 \\ 2 & 1 & 0 \\ 0 & 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 1 \\ 1 \end{bmatrix}$$

c)
$$\begin{bmatrix} -1 & 1 & 0 & -3 \\ 1 & 0 & 3 & 1 \\ 0 & 1 & -1 & -1 \\ 3 & 0 & 1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 4 \\ 0 \\ 3 \\ 1 \end{bmatrix}$$

d) Tome el sistema de ecuaciones del apartado c) de la Pregunta 1.

3. Utilice la factorización $PA = LU$ para encontrar la inversa y el determinante de
$$A = \begin{bmatrix} 0 & -1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 2 & 0 \\ 2 & 0 & 1 & 0 \end{bmatrix}$$

4. Considere el sistema
$$\begin{cases} & & x_2 & + & 2x_3 & = & 1 \\ 2x_1 & - & x_2 & & & = & 2 \\ & & 2x_2 & + & x_3 & = & 3 \end{cases}$$
Determine la factorización $PA = LU$ de la matriz asociada al sistema. Utilice la factorización para resolver el sistema.

5. a) Demuestre que la matriz
$$A = \begin{bmatrix} 4 & -1 & -1 & 0 \\ -1 & 4 & 0 & -1 \\ -1 & 0 & 4 & -1 \\ 0 & -1 & -1 & 4 \end{bmatrix}$$
es definida positiva, usando la factorización LU y la factorización de Cholesky.

b) Resuelva $A\mathbf{x} = \mathbf{b}$, donde $\mathbf{b} = (1,1,1,1)$, usando la factorización de Cholesky de A.

6. Usando el método de Householder, encuentre la factorización QR de la matriz
$$A = \begin{bmatrix} -1 & 2 & 3 \\ 4 & -5 & 6 \\ 7 & 8 & -9 \end{bmatrix}$$
y luego utilícela para resolver el sistema $A\mathbf{x} = \mathbf{b}$, donde $\mathbf{b} = (3,2,1)$.

7. Sea $A = \begin{bmatrix} C & B \\ B^T & 0 \end{bmatrix}$, donde $C \in \mathbb{R}^{n\times n}$ es simétrica definida positiva y $B \in \mathbb{R}^{n\times m}$ es de rango completo con $m \geq n$. Desarrolle un algoritmo para triangularizar A, y luego úselo para resolver $A \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{bmatrix}$, para C, B, $\mathbf{b}_1$ y $\mathbf{b}_2$, seleccionados por usted.

8. a) Use la función `svd` de la librería `Numpy` y encuentre la SVD de las siguientes matrices:

i) $A = \begin{bmatrix} 2 & 0 \\ 0 & -3 \\ 0 & 0 \end{bmatrix}$

ii) $A = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$

iii) $A = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$

iv) $A = \begin{bmatrix} 1 & & & & \\ & 0 & & & \\ & & 2 & & \\ & & & 0 & \\ & & & & -5 \end{bmatrix}$ v) $A = \begin{bmatrix} 1 & 1 \\ 10^{-1} & 0 \\ 0 & 10^{-5} \end{bmatrix}$

b) Para cada una de las matrices anteriores, use la parte a) y encuentre: rang(A), $\|A\|_2$ y $\|A\|_F$.

9. Sea A una matriz invertible. Establezca que $\|A\|_2 = 1$ si y sólo si A es un múltiplo de una matriz ortogonal.

10. Sea U una matriz ortogonal. Usando la SVD de A, demuestre que

$$\|UA\|_2 = \|A\|_2 \qquad \text{y} \qquad \|UA\|_F = \|A\|_F.$$

11. Considere el siguiente sistema sobredeterminado

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 9 \\ 7 & 11 & 18 \\ -2 & 3 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \mathbf{b}.$$

a) Encuentre una solución resolviendo las ecuaciones normales $A^T A\mathbf{x} = A^T\mathbf{b}$. Para ello, utilice la factorización de Cholesky.

b) Encuentre una solución utilizando la SVD de A.

¿Qué observa?

12. Verifique que las siguientes matrices

$$\begin{aligned} J &= I - D^{-1}A, \\ G &= I - (D-E)^{-1}A, \\ S &= I - \omega(D - \omega E)^{-1}A, \end{aligned}$$

son respectivamente las matrices M de la forma iterativa $\mathbf{x}_{k+1} = M\mathbf{x}_k + \mathbf{v}$, de los métodos Jacobi, Gauss-Seidel y SOR.

13. a) Escriba la forma vectorial del método de Jacobi aplicado al sistema

$$\begin{cases} 2x_1 + x_2 = 4 \\ x_1 + 2x_2 = 5 \end{cases}$$

y establezca que converge.

b) Intercambie las 2 ecuaciones y demuestre que ahora no converge. ¿Converge el método de Gauss-Seidel?

14. Considere una matriz $A = (a_{ij})$ 2×2 tal que $a_{11} \neq 0$ y $a_{22} \neq 0$.

a) Establezca que los métodos de Jacobi y Gauss-Seidel para A convergen o divergen simultáneamente.

b) Encuentre una condición necesaria y suficiente sobre las entradas de A para que ambos métodos converjan.

c) Si ambos convergen, ¿cuál de ellos lo hace más rápidamente? (Justifique su respuesta.)

15. Considere la matriz

$$A = \begin{bmatrix} 1 & a & a \\ a & 1 & a \\ a & a & 1 \end{bmatrix}.$$

Encuentre los valores de a para los cuales A es simétrica definida positiva, pero el método de Jacobi no converge.

16. Considere la forma iterativa $\mathbf{x}_{k+1} = M\mathbf{x}_k + \mathbf{v}$ de los métodos Jacobi, Gauss-Seidel y SOR. Sea $A \in \mathbb{R}^{n\times n}$ una matriz dada por

$$A = \begin{bmatrix} 2 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 2 & -1 \\ & & & & -1 & 2 \end{bmatrix}$$

Encuentre la forma explícita de la matriz M para cada uno de los métodos.

17. Caracterice a la familia de todas las matrices A no singulares $n \times n$ para las cuales un paso del método de Gauss-Seidel resuelve el sistema $A\mathbf{x} = b$, suponiendo que se inició el proceso con $\mathbf{x}_0 = 0$.

18. Dé un ejemplo de una matriz A que no sea estrictamente diagonal dominante, de manera que el método de Gauss-Seidel converja a la solución de $A\mathbf{x} = b$, para cualquier iterado inicial $\mathbf{x}_0 \in \mathbb{R}^n$.

19. Ilustre como influye el parámetro ω en la convergencia del método SOR, al resolver el sistema $A\mathbf{x} = \mathbf{b}$ desde el iterado inicial $\mathbf{x}_0 = 0$, donde $\mathbf{b} = (1,1,1,1)$ y

$$A = \begin{bmatrix} -4 & 1 & 1 & 1 \\ 1 & -4 & 1 & 1 \\ 1 & 1 & -4 & 1 \\ 1 & 1 & 1 & -4 \end{bmatrix}.$$

La solución exacta es $\mathbf{x} = (1,1,1,1)$. Intente con los valores $\omega = 1.0, 1.1, \ldots, 1.9$.

20. Supongamos que un objeto puede estar en cualquiera de los $n+1$ puntos igualmente espaciados $x_0, x_1, \ldots, x_n$. Cuando el objeto está en la ubicación x_i, es igualmente probable que se mueva a x_{i-1} o x_{i+1} y no se puede mover directamente a ninguna otra ubicación. Considere las probabilidades $\{p_i\}_{i=0}^n$ de que un objeto que comienza en la ubicación x_i alcance el punto final izquierdo x_0 antes de llegar al punto final derecho x_n. Claramente, $p_0 = 1$ y $p_n = 0$. Dado que el objeto puede moverse a x_i solo desde x_{i-1} o x_{i+1} y lo hace con probabilidad $1/2$ para cada una de estas ubicaciones, se tiene:

$$p_i = \frac{1}{2}p_{i-1} + \frac{1}{2}p_{i+2}, \quad \text{para cada } i = 1,2,\ldots,n-1.$$

a) Muestre que

$$\begin{bmatrix} 1 & -\frac{1}{2} & & & & \\ -\frac{1}{2} & 1 & -\frac{1}{2} & & & \\ & -\frac{1}{2} & 1 & -\frac{1}{2} & & \\ & & \ddots & \ddots & \ddots & \\ & & & -\frac{1}{2} & 1 & -\frac{1}{2} \\ & & & & -\frac{1}{2} & 1 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ \vdots \\ p_{n-1} \end{bmatrix} = \begin{bmatrix} \frac{1}{2} \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

b) Para $n = 25, 50$ y 100, y resuelva este sistema usando:

i) Factorización LU

ii) Factorización de Cholesky

iii) Los métodos Jacobi, Gauss-Seidel y SOR ($\omega = 1.8$), con $\mathbf{x}_0 = 0$.

iv) FOM(20), con $\mathbf{x}_0 = 0$.

v) GMRES(20), con $\mathbf{x}_0 = 0$.

¿Qué observa?

c) Cambie las probabilidades a α y $1-\alpha$ para el movimiento hacia la izquierda y hacia la derecha, respectivamente, y obtenga el sistema lineal similar al de la parte a).

d) Repita la parte b) con $\alpha = 1/3$ y el sistema obtenido en c).

21. Defina un sistema lineal $A\mathbf{x} = \mathbf{b}$ usando el Script

```
A = np.random.rand(n,n)
xt = np.random.rand(n)
b = A.dot(xt)
```

Se tiene que $\mathbf{x}_t$ es la solución del sistema $A\mathbf{x} = \mathbf{b}$. Tome $n = 1000$, una tolerancia de `tol=1e-6` y un máximo de 2000 iteraciones, y corra los métodos:

i) FOM(10)

ii) FOM(50)

iii) GMRES(10)

iv) GMRES(20)

Para cada método construya la gráfica iteraciones vs. $\|\mathbf{x}_t - \mathbf{x}_k\|_1$. Comente los resultados.

3 | Interpolación y Ajuste de Curvas

Conjuntos de datos discretos o tablas de la forma

t	t_0	t_1	$\cdots$	t_n
y	y_0	y_1	$\cdots$	y_n

aparecen con mucha frecuencia en cálculos técnicos. Las fuentes de los datos pueden ser observaciones experimentales o cálculos numéricos. Existe diferencia entre interpolación y ajuste de curvas. En la interpolación se construye una curva que pasa por los puntos. De este modo, es necesario suponer que los puntos dados son exactos y precisos. El objetivo del ajuste de curvas es encontrar una curva suave que se aproxima a los datos en algún sentido. La curva no tiene que pasar por los puntos dados. Es común aplicar el ajuste de curvas a datos que contienen dispersión (ruido), por lo general debido a errores de medición.

El problema de interpolación más simple en una dimensión tiene la siguiente forma: para datos dados (t_i, y_i) , con $t_0 < t_1 < t_2 < \cdots, t_n$, se busca una función $g : \mathbb{R} \to \mathbb{R}$ de manera que

$$g(t_i) = y_i, \quad i = 0, 1, \ldots, n.$$

La función g se denomina *función interpolante* para un conjunto de datos dado. A menudo se requiere que la función interpolante tenga propiedades deseables (continua, derivable, convexa, etc.). La interpolación tiene muchos usos, entre los cuales podemos mencionar:

- Construir la gráfica de una curva suave que pasa por puntos de datos discretos.
- Obtener información relevante de una tabla de datos.
- Aproximar la derivada o la integral de datos de una tabla.
- Evaluar una función matemática de forma rápida y sencilla.
- Reemplazar una función complicada por una sencilla.

Elegir que tipo de función interpolante a utilizar no es una tarea sencilla. La selección de una función de interpolación se basa generalmente en las respuestas de las siguientes interrogantes:

- ¿Qué forma debe tener la función? ¿Deben tomarse en cuenta consideraciones matemáticas o físicas relevantes que sugieran una forma particular de la función?
- ¿Debería la función heredar las propiedades de los datos, como la monotonicidad, la convexidad o la periodicidad?
- Si la función interpolante y los datos se trazan, ¿deberían los resultados ser agradables visualmente?

De esta forma, la elección de la función de interpolación depende de las respuestas a estas preguntas, así como de los datos considerados. Las funciones más comúnmente utilizadas para la interpolación son las siguientes:

- Polinomios
- Polinomios a trozos (Spline)
- Funciones trigonométricas
- Funciones exponenciales
- Funciones racionales

En este capítulo estudiaremos la interpolación por polinomios (interpolación polinomial) y la interpolación por funciones trigonométricas (interpolación trigonométrica). En esta parte describiremos la transformada rápida de Fourier, cuyo uso en aplicaciones como procesamiento de señales es de suma importancia. Finalizaremos el capítulo con el ajuste de curvas por cuadrados mínimos lineales. Comencemos el capítulo con la interpolación polinomial.

3.1 Interpolación Polinomial

Consideremos una función $f(t)$ definida por un conjunto de $n+1$ datos (t_i, y_i), donde $y_i = f(t_i)$, $i = 0, 1, \ldots, n$. Los puntos t_i se denominan *nodos de interpolación*. El objetivo de la interpolación polinomial es encontrar un polinomio $p_n(t)$ con el menor grado posible tal que satisfaga las ecuaciones de interpolación:

$$p_n(t_i) = f(t_i), \quad i = 0, 1, \ldots, n. \tag{3.1}$$

Se dice que tal polinomio hace una interpolación de los datos y se denomina *polinomio interpolante*. El primer procedimiento que estudiaremos para encontrar el polinomio interpolante es el *método de los coeficientes indeterminados*, que consiste en determinar los coeficientes de p_n a través de la solución de un sistema de ecuaciones lineales, que se construye con las ecuaciones dadas en (3.1). La forma canónica del polinomio $p_n(t)$ es:

$$p_n(t) = a_0 + a_1 t + a_2 t + \cdots + a_n t^n.$$

Ahora, forzando que se satisfagan las ecuaciones de interpolación (3.1), obtenemos el siguiente sistema de ecuaciones lineales:

$$\begin{cases} a_0 + a_1 t_0 + a_2 t_0^2 + \cdots + a_n t_0^n &= y_0 \\ a_0 + a_1 t_1 + a_2 t_1^2 + \cdots + a_n t_1^n &= y_1 \\ \qquad\qquad \vdots & \\ a_0 + a_1 t_n + a_2 t_n^2 + \cdots + a_n t_n^n &= y_n \end{cases} \tag{3.2}$$

Las incógnitas de este sistema de ecuaciones lineales son los coeficientes $a_0, a_1, \ldots, a_n$ del polinomio $p_n(t)$. La forma matricial de este sistema es $V\mathbf{a} = \mathbf{y}$, donde la matriz $V \in \mathbb{R}^{(n+1)\times(n+1)}$, el vector $\mathbf{a} \in \mathbb{R}^{n+1}$ y el vector $\mathbf{y} \in \mathbb{R}^{n+1}$, están dados por:

$$V = \begin{bmatrix} 1 & t_0 & t_0^2 & \cdots & t_0^n \\ 1 & t_1 & t_1^2 & \cdots & t_1^n \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & t_n & t_n^2 & \vdots & t_n^n \end{bmatrix}, \quad \mathbf{a} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

La matriz V se llama *matriz de Vandermonde*; además, se puede demostrar que

$$\det(V) = \prod_{0 \le j \le i \le n} (t_i - t_j) \neq 0,$$

siempre que $t_i \neq t_j$, para todo $i \neq j$. Por lo tanto, si los nodos t_i son todos distintos, el sistema de ecuaciones lineales (3.2) tiene solución única; en otras palabras, los coeficientes $a_0, a_1, \ldots, a_n$ de $p_n(t)$ se determinan en forma única. Entonces, el polinomio de interpolación $p_n(t)$ es único, siempre y cuando los nodos t_i sean todos distintos. Esto último se expresa en el siguiente teorema.

Teorema 3.1. *Si $t_0, t_1, \ldots, t_n$ son números reales distintos, entonces para valores arbitrarios $y_0, y_1, \ldots, y_n$, existe un único polinomio $p_n(t)$ de grado a lo sumo n tal que*

$$p_n(t_i) = y_i \quad \textit{para } i = 0, 1, \ldots, n.$$

Ejemplo 3.1. En este ejemplo se construye el polinomio interpolante resolviendo el sistema (3.2). Consideremos la siguiente tabla de datos.

t_i	-1	0	1
y_i	2	1	4

Note que el número de puntos es 3, lo que indica que el polinomio interpolante es de grado a lo sumo 2 y tiene la forma canónica

$$p_2(t) = a_0 + a_1 t + a_2 t^2.$$

Las ecuaciones de interpolación que se deben satisfacer son:

$$\begin{aligned} p_2(-1) &= a_0 + a_1(-1) + a_2(-1)^2 = 2, \\ p_2(0) &= a_0 + a_1 0 + a_2 0^2 = 1, \\ p_2(1) &= a_0 + a_1 1 + a_2 1^2 = 4, \end{aligned}$$

las cuales originan un sistema de ecuaciones lineales cuya forma matricial es:

$$\begin{bmatrix} 1 & -1 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 4 \end{bmatrix}.$$

La solución de este sistema de ecuaciones es: $a_0 = 1$, $a_1 = 1$ y $a_2 = 2$. Por lo tanto, el polinomio interpolante es $p_2(t) = 1 + t + 2t^2$, cuya gráfica se muestra en la Figura 3.1.

■

Es de subrayar que la matriz de Vandermonde está mal condicionada, y por ello los coeficientes a_i podrían no quedar determinados con precisión como resultado de resolver el sistema (3.2). Por consiguiente este enfoque no es recomendable su uso en la práctica, cuando el número de nodos es grande.

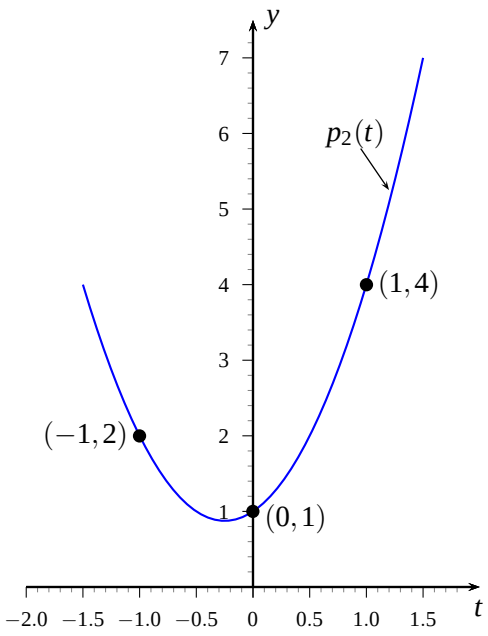


Figura 3.1. Polinomio interpolante del Ejemplo 3.1

Sabemos que el conjunto de los polinomios de grado menor o igual a n es un subespacio vectorial de dimensión $n+1$, que denotaremos por Π_n. La base canónica de Π_n es $\{1,t,t^2,\ldots,t^n\}$; por consiguiente, todo polinomio de grado menor o igual a n se puede expresar como una combinación lineal de los elementos de esta base, esto es,

$$p_n(t) = a_0 + a_1 t + a_2 t + \cdots + a_n t^n,$$

donde los coeficientes $a_0, a_1, \ldots, a_n$ son números reales, que representan las coordenadas de $p_n(t)$ en la base canónica. En otras palabras, el procedimiento de construcción del polinomio interpolante descrito arriba, consiste en encontrar las coordenadas de $p_n(t)$ en la base canónica de Π_n de manera que se satisfagan las condiciones de interpolación. Si se escoge otra base de Π_n y se calculan las nuevas coordenadas de $p_n(t)$ en esa base, estaríamos construyendo de una forma diferente a $p_n(t)$. Por tanto, escoger una base de Π_n produce una manera distinta de expresar el polinomio interpolante.

3.1.1 Polinomio Interpolante en la Forma de Lagrange

Presentamos ahora una forma alternativa del polinomio interpolante $p_n(t)$ asociado con los datos (t_i, y_i), para $i = 0, 1, \ldots, n$. Es importante entender que existe uno y sólo un polinomio interpolante de grado $\leq n$ asociado con los datos, por supuesto, suponiendo que los $n+1$ nodos t_i son distintos. Sin embargo, el polinomio interpolante se puede expresar de maneras distintas y también se puede construir a través de diferentes procedimientos.

Consideremos una base $\{\ell_0, \ell_1, \ldots, \ell_n\}$ de Π_n tal que para $i, j = 0, 1, \ldots, n$,

$$\ell_i(t_j) = \begin{cases} 1, & \text{si } i = j; \\ 0, & \text{si } i \neq j. \end{cases} \tag{3.3}$$

Calculemos las coordenadas de $p_n(t)$ en esta base, es decir, hallemos los números reales $b_0, b_1, \ldots, b_n$

tales que

$$p_n(t) = b_0\ell_0(t) + b_1\ell_1(t) + \cdots + b_n\ell_n(t). \tag{3.4}$$

Utilizando las condiciones de interpolación, $y_i = p_n(t_i)$, conjuntamente con (3.4) y (3.3), obtenemos que $b_i = y_i$, para $i = 0, 1, \ldots, n$. Luego, hemos encontrado otra manera de expresar a $p_n(t)$, que está dada por la siguiente expresión:

$$p_n(t) = y_0\ell_0(t) + y_1\ell_1(t) + \cdots + y_n\ell_n(t) = \sum_{k=0}^{n} y_k\ell_k(t). \tag{3.5}$$

Esta expresión se denomina *forma de Lagrange* del polinomio interpolante. En este caso, los polinomios $\ell_0, \ell_1, \ldots, \ell_n$ dependen de los nodos $t_0, t_1, \ldots, t_n$, pero no dependen de las ordenadas $y_0, y_1, \ldots, y_n$. Los polinomios ℓ_i se dominan *lagrangeanos* o *funciones cardinales* y están definidos como:

$$\ell_i(t) = \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{(t - t_j)}{(t_i - t_j)}, \quad i = 0, 1, \ldots, n.$$

La base formada por los polinomios $\ell_i(t)$ se denomina *base de Lagrange*. Se puede probar fácilmente que cada $\ell_i(t)$ es un polinomio de grado n que satisface (3.3).

Ejemplo 3.2. Construyamos la forma de Lagrange del polinomio interpolante del Ejemplo 3.1. Los nodos son: $t_0 = -1$, $t_1 = 0$ y $t_2 = 1$. Por tanto, los lagrangeanos son:

$$\ell_0(t) = \frac{(t-t_1)(t-t_2)}{(t_0-t_1)(t_0-t_2)} = \frac{(t-0)(t-1)}{(-1-0)(-1-1)} = \frac{t(t-1)}{2},$$
$$\ell_1(t) = \frac{(t-t_0)(t-t_2)}{(t_1-t_0)(t_1-t_2)} = \frac{(t-(-1))(t-1)}{(0-(-1))(0-1)} = -(t+1)(t-1),$$
$$\ell_2(t) = \frac{(t-t_0)(t-t_1)}{(t_2-t_0)(t_2-t_1)} = \frac{(t-(-1))(t-0)}{(1-(-1))(1-0)} = \frac{t(t+1)}{2}.$$

En la siguiente figura se muestran las gráficas de los lagrangeanos ℓ_0, ℓ_1 y ℓ_2.

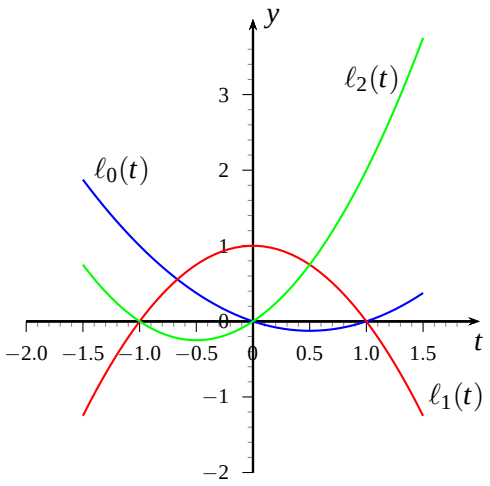


Figura 3.2. Lagrangeanos ℓ_0, ℓ_1 y ℓ_2.

Prosigamos con la construcción del polinomio interpolante. Como $y_0 = 2$, $y_1 = 1$ y $y_2 = 4$, entonces el polinomio interpolante en su forma de Lagrange es:

$$\begin{aligned} p_2(t) &= 2\ell_0(t) + \ell_1(t) + 4\ell_2(t) \\ &= 2\frac{t(t-1)}{2} - (t+1)(t-1) + 4\frac{t(t+1)}{2} \\ &= t(t-1) - (t+1)(t-1) + 2t(t+1). \end{aligned}$$

Note que realizando operaciones algebraicas en la forma de Lagrange, se obtiene la forma canónica del polinomio interpolante.

■

3.1.2 Polinomio Interpolante en la Forma de Newton

Anteriormente vimos que el polinomio interpolante también se puede expresar en la forma de Lagrange. Pero esta manera de expresarlo no es la más eficiente. A continuación deduciremos la forma de Newton del polinomio interpolante $p_n(t)$ de grado menor o igual a n asociado con los puntos (t_i, y_i), para $i = 0, 1, \ldots, n$.

Sea $p_{n-1}(t)$ el polinomio interpolante de grado menor o igual a $n-1$ asociado a los puntos (t_i, y_i) para $i = 0, 1, \ldots, n-1$, es decir, $p_{n-1}(t)$ satisface las siguientes ecuaciones de interpolación:

$$p_{n-1}(t_i) = y_i \quad \text{para } i = 0, 1, \ldots, n-1.$$

Sea $g(t)$ un polinomio de grado a lo sumo n definido como

$$g(t) = p_{n-1}(t) + c(t - t_0)(t - t_1)\cdots(t - t_{n-1}),$$

donde $c \in \mathbb{R}$. Afirmamos que $g(t)$ es el polinomio interpolante $p_n(t)$, para cierto valor de la constante c. Determinemos tal valor de c. Es claro que $g(t)$ interpola los n primeros puntos (t_i, y_i), para $i = 0, 1, \ldots, n-1$, es decir,

$$g(t_i) = y_i, \quad \text{para } i = 0, 1, \ldots, n-1.$$

Ahora, queremos hallar un cierto número real c tal que

$$y_n = g(t_n) = p_{n-1}(t_n) + c(t_n - t_0)(t_n - x_1)\cdots(t_n - x_{n-1}).$$

Despejando c en la última ecuación obtenemos:

$$c = \frac{y_n - p_{n-1}(t_n)}{\prod_{j=0}^{n-1}(t_n - t_j)}. \tag{3.6}$$

Así, con c dada por la ecuación (3.6) el polinomio $g(t)$ satisface las ecuaciones de interpolación $g(t_i) = y_i$, $i = 0, 1, \ldots, n$, es decir, $g(t)$ es el polinomio interpolante asociado con los datos (t_i, y_i) con $i = 0, 1, \ldots, n$. De esta forma, el polinomio interpolante $p_n(t)$ se obtiene a partir de $p_{n-1}(t)$ usando la recurrencia

$$p_n(t) = p_{n-1}(t) + (y_n - p_{n-1}(t_n)) \prod_{j=0}^{n-1} \frac{(t - t_j)}{(t_n - t_j)}.$$

La ecuación anterior nos permite construir el polinomio interpolante en la forma de Newton mediante el siguiente proceso recursivo: primero se construye el polinomio que interpola los puntos (t_0,y_0) y (t_1,y_1), esto es:

$$p_1(t) = c_0 + c_1(t-t_0),$$

donde las constantes c_0 y c_1 se calculan usando las ecuaciones: $p_1(t_0) = y_0$ y $p_1(t_1) = y_1$. Ahora, con $p_1(t)$ se construye el polinomio que interpola los puntos (t_0,y_0), (t_1,y_1) y (t_2,y_2),

$$p_2(t) = p_1(t) + c_2(t-t_0)(t-t_1) = c_0 + c_1(t-t_0) + c_2(t-t_0)(t-t_1),$$

donde la constante c_2 se calcula usando la ecuación $p_2(x_2) = y_2$. Siguiendo con este proceso constructivo obtenemos el polinomio interpolante $p_n(x)$ en la forma de Newton:

$$p_n(t) = c_0 + c_1(t-t_0) + c_2(t-t_0)(t-t_1) + \cdots + c_n \prod_{i=0}^{n-1}(t-t_i), \tag{3.7}$$

donde los coeficientes se determinan mediante la fórmula

$$\begin{cases} c_0 = y_0, \\ c_k = (y_k - p_{k-1}(t_k))/\prod_{i=0}^{k-1}(t_k - t_{k-1}), \quad \text{para } k = 1,2,\ldots,n. \end{cases} \tag{3.8}$$

Veamos con un ejemplo la construcción del polinomio interpolante en la forma de Newton.

Ejemplo 3.3. Consideremos la siguiente tabla de datos:

t_i	-1	0	1
y_i	2	1	4

Construyamos el polinomio interpolante $p_2(t)$ en la forma de Newton de grado menor o igual a 2 asociado a la tabla de datos dada. El polinomio $p_2(t)$ está dado por

$$p_2(t) = c_0 + c_1(t+1) + c_2(t+1)t.$$

Para determinar los coeficientes c_i, se emplean las condiciones de interpolación:

$$p_2(-1) = 2,\ p_2(0) = 1,\ p_2(1) = 4.$$

Así tenemos el siguiente sistema de ecuaciones, cuyas incógnitas son los coeficientes c_i:

$$\begin{cases} 2 = p_2(-1) = c_0 + c_1(-1+1) + c_2(-1+1)(-1) \\ 1 = p_2(0) = c_0 + c_1(0+1) + c_2(0+1)0 \\ 4 = p_2(1) = c_0 + c_1(1+1) + c_2(1+1)1 \end{cases} \Rightarrow \quad \begin{cases} c_0 = 2 \\ c_0 + c_1 = 1 \\ c_0 + 2c_1 + 2c_2 = 4 \end{cases}$$

La solución de este sistema es:

$$c_0 = 2, \quad c_1 = -1, \quad c_2 = 2.$$

De esta forma, el polinomio interpolante $p_2(t)$ en la forma de Newton es:

$$p_2(t) = 2 - (t+1) + 2(t+1)t.$$

Supongamos ahora que a la tabla inicial le agregamos el punto $(-2,2)$, es decir, tenemos la nueva tabla:

t_i	-2	-1	0	1
y_i	2	2	1	4

Construyamos ahora el polinomio interpolante $p_3(t)$ en la forma de Newton de grado menor o igual a 3 asociado a la nueva tabla de datos. Para ello, utilizamos el polinomio $p_2(t)$ obtenido anteriormente, esto es, usamos la fórmula:

$$p_3(t) = p_2(t) + c_3(t+1)t(t-1).$$

Ahora, para hallar el valor del coeficiente c_3, utilizamos la ecuación $p_3(-2) = 2$. Así obtenemos:

$$p_3(-2) = 2 \qquad \Rightarrow \qquad c_3 = \left.\frac{2 - p_2(t)}{(t+1)t(t-1)}\right|_{t=-2} = \frac{5}{6}.$$

Luego, el polinomio interpolante $p_3(t)$ expresado en la forma de Newton es:

$$p_3(t) = 2 - (t+1) + 2(t+1)t + \frac{5}{6}(t+1)t(t-1).$$

En la Figura 3.3 se muestran las gráficas de $p_2(t)$ y $p_3(t)$.

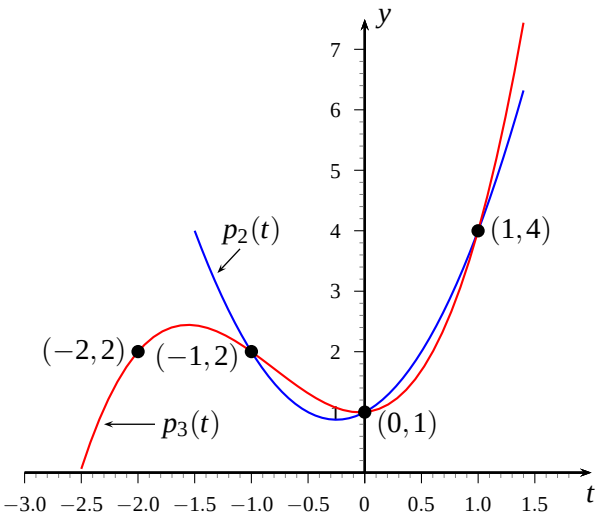


Figura 3.3. Polinomios interpolantes del Ejemplo 3.3

La ecuación (3.7) significa que el polinomio interpolante $p_n(t)$ se expresa como una combinación lineal de los elementos de la base,

$$\{1, (t-t_0), (t-t_0)(t-t_1), \ldots, (t-t_0)(t-t_1)\cdots(t-t_n)\}$$

donde los números reales $c_0, c_1, \ldots, c_n$ dados por (3.8) son las coordenadas de $p_n(t)$ en esta base. En otras palabras,

$$p_n(t) = \sum_{k=0}^{n} c_k \varphi_k(t), \tag{3.9}$$

donde

$$\varphi_k(t) = \begin{cases} 1, & \text{para } k = 0, \\ \prod_{i=0}^{k-1}(t - t_i), & \text{para } k = 1, \ldots, n. \end{cases} \tag{3.10}$$

y los números reales c_k están definidos en (3.8).

Al conjunto de polinomios $\varphi_k(t)$ definidos en (3.10) se le denomina con el nombre de *Base Triangular de Newton* de Π_n. Con respecto a la base de Lagrange, la base triangular tiene ventajas gracias a su construcción. Todos los lagrangeanos tienen grado n, mientras que el grado de los polinomios de la base triangular aumentan progresivamente; más aún, los polinomios $\varphi_k(t)$ pueden definirse de manera recursiva,

$$\varphi_k(t) = \begin{cases} 1, & \text{para } k = 0, \\ (t - t_k)\varphi_{k-1}(t), & \text{para } k = 1, \ldots, n. \end{cases} \tag{3.11}$$

Por (3.11) se deduce que $\varphi_k(t_j) = 0$ para $k > j$, que es una característica muy importante de la base triangular de Newton. Además, la ecuación (3.8) permite utilizar el método de Horner para evaluar el polinomio interpolante en su forma de Newton. Esto lo veremos más adelante.

Si consideramos las condiciones de interpolación $p_n(t_i) = y_i$, entonces por (3.9) y (3.11), obtenemos el siguiente sistema de ecuaciones lineales, cuyas incógnitas son los coeficientes c_k,

$$\begin{cases} y_0 = c_0 \\ y_1 = c_0 + c_1(t_1 - t_0) \\ y_2 = c_0 + c_1(t_2 - t_0) + c_2(t_2 - t_0)(t_2 - t_1) \\ \vdots \\ y_n = c_0 + c_1(t_n - t_0) + c_2(t_n - t_0)(t_n - t_1) + \cdots + c_n(t_n - t_0)\cdots(t_n - t_{n-1}) \end{cases} \tag{3.12}$$

Este sistema es triangular superior, lo cual facilita el cálculo de los coeficientes c_k a través de la sustitución progresiva. Esto es exactamente lo que hace la fórmula (3.8). Existe un procedimiento más eficiente para calcular los coeficientes c_k. En la siguiente sección se describe tal procedimiento, denominado diferencias divididas.

3.1.3 Tabla de Diferencias Divididas

Sea $f(t)$ una función cuyos valores son conocidos o calculables en un conjunto de nodos $t_0, t_1, \ldots, t_n$. Sabemos que hay un único polinomio $p_n(t)$ de grado menor o igual a n que interpola a $f(t)$ en los $n+1$ nodos, es decir, se satisfacen las ecuaciones de interpolación $p_n(t_i) = f(t_i)$, $i = 0, 1, \ldots, n$. Tales ecuaciones dan lugar al sistema de ecuaciones lineales (3.12), con el que se determinan los coeficientes c_k del polinomio interpolante en la forma de Newton dado por la ecuación (3.7). Por ejemplo, considere el caso en que hay tres nodos, t_0, t_1, t_2, y

$$p_2(t) = c_0 + c_1(t - t_0) + c_2(t - t_0)(t - t_1).$$

En este caso, la forma matricial del sistema (3.12) es:

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & (t_1 - t_0) & 0 \\ 1 & (t_2 - t_0) & (t_2 - t_0)(t_2 - t_1) \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} f(t_0) \\ f(t_1) \\ f(t_2) \end{bmatrix}.$$

Utilizando la sustitución progresiva, se tiene que

$$\begin{cases} c_0 = f(t_0), \\ c_1 = \dfrac{f(t_1)-f(t_0)}{t_1-t_0}, \\ c_2 = \dfrac{f(t_0)}{(t_2-t_0)(t_1-t_0)} - \dfrac{f(t_1)}{(t_2-t_1)(t_1-t_0)} + \dfrac{f(t_2)}{(t_2-t_0)(t_1-t_1)}, \end{cases} \tag{3.13}$$

es decir, c_0 depende de $f(t_0)$, c_1 depende de $f(t_0)$ y $f(t_1)$, y así sucesivamente. De este modo, c_k depende de los valores de $f(t)$ en $t_0, t_1, \ldots, t_k$. La notación

$$c_k = f[t_0, t_1, \ldots, t_k]$$

se adoptó hace muchos años para significar esta dependencia, además, se denominan *diferencias divididas* de $f(t)$.

Definición 3.1 (Diferencias Divididas)**.** Las diferencias divididas de una función $f(x)$ se definen como:

$$\begin{aligned} f[t_i] &= f(t_i), \\ f[t_{i-1}, t_i] &= \frac{f[t_i]-f[t_{i-1}]}{t_i - t_{i-1}}, \\ f[t_{i-2}, t_{i-1}, t_i] &= \frac{f[t_{i-1}, t_i]-f[t_{i-2}, t_{i-1}]}{t_i - t_{i-2}}, \\ f[t_{i-3}, t_{i-2}, t_{i-1}, t_i] &= \frac{f[t_{i-2}, t_{i-1}, t_i]-f[t_{i-3}, t_{i-2}, t_{i-1}]}{x_i - x_{i-3}}. \end{aligned}$$

Las diferencias divididas de orden superior se forman de acuerdo con la siguiente regla recursiva:

$$f[t_i, t_{i+1}, \ldots, t_{i+j}] = \frac{f[t_{i+1}, t_{i+2}, \ldots, t_{i+j}] - f[t_i, t_{i+1}, \ldots, t_{i+j-1}]}{t_{i+j} - t_i}.$$

La Tabla 4.1 se construyó empleando la definición de diferencias divididas para un ejemplo de 5 nodos. A las diferencias divididas

$$f[t_i],\ f[t_{i-1}, t_i],\ f[t_{i-2}, t_{i-1}, t_i],\ f[t_{i-3}, t_{i-2}, t_{i-1}, t_i], \quad \text{etc.},$$

se le denominan, respectivamente, diferencias de orden 0, 1, 2, 3, etc.

Tabla 3.1. Tabla de diferencias divididas para $y = f(t)$

t_i	$f[t_i]$	$f[t_{i-1}, t_i]$	$f[t_{i-2}, t_{i-1}, t_i]$	$f[t_{i-3}, t_{i-2}, t_{i-1}, t_i]$	$f[t_{i-4}, t_{i-3}, t_{i-2}, t_{i-1}, t_i]$
t_0	$f[t_0]$				
t_1	$f[t_1]$	$f[t_0, t_1]$			
t_2	$f[t_2]$	$f[t_1, t_2]$	$f[t_0, t_1, t_2]$		
t_3	$f[t_3]$	$f[t_2, t_3]$	$f[t_1, t_2, t_3]$	$f[t_0, t_1, t_2, t_3]$	
t_4	$f[t_4]$	$f[t_3, t_4]$	$f[t_2, t_3, t_4]$	$f[t_1, t_2, t_3, t_4]$	$f[t_0, t_1, t_2, t_3, t_4]$

Regresemos al cálculo de los coeficientes c_k del polinomio interpolante en la forma de Newton dado por la ecuación (3.7). Tales coeficientes se calculan utilizando las diferencias divididas:

$$c_k = f[t_0, t_1, \ldots, t_i], \quad k = 0, 1, \ldots, n. \tag{3.14}$$

Por ejemplo, para el caso en que hay tres nodos, t_0, t_1, t_2, los coeficientes c_k están dados por

$$\begin{aligned} c_0 &= f[t_0] = f(t_0), \\ c_1 &= f[t_0, t_1] = \frac{f(t_1) - f(t_0)}{t_1 - t_0}, \end{aligned}$$

$$\begin{aligned} c_2 = f[t_0, t_1, t_2] &= \frac{f[t_1, t_2] - f[t_0, t_1]}{(t_2 - t_0)} = \frac{\dfrac{f(t_2) - f(t_1)}{(t_2 - t_1)} - \dfrac{f(t_1) - f(t_0)}{(t_1 - t_0)}}{(t_2 - t_0)} \\ &= \frac{(t_1 - t_0)(f(t_2) - f(t_1)) - (t_2 - t_1)(f(t_1) - f(t_0))}{(t_2 - t_0)(t_2 - t_1)(t_1 - t_0)} \\ &= \frac{(t_2 - t_1) f(t_0) - (t_2 - t_0) f(t_1) + (t_1 - t_0) f(t_2)}{(t_2 - t_0)(t_2 - t_1)(t_1 - t_0)} \\ &= \frac{f(t_0)}{(t_2 - t_0)(t_1 - t_0)} - \frac{f(t_1)}{(t_2 - t_1)(t_1 - t_0)} + \frac{f(t_2)}{(t_2 - t_0)(t_1 - t_1)}, \end{aligned}$$

en otras palabras, los valores hallados de c_0, c_1 y c_2 usando (3.14), son iguales a los dados en (3.13).

Ejemplo 3.4. Construyamos la tabla de diferencias divididas asociada al conjunto de datos:

t_i	-2	-1	0	1
y_i	2	2	1	4

Los nodos son:

$$t_0 = -2, \quad t_1 = -1, \quad t_2 = 0, \quad t_3 = 1.$$

Las diferencias divididas de orden 0 son:

$$f[t_0] = 2, \quad f[t_1] = 2, \quad f[t_2] = 1, \quad f[t_3] = 4.$$

Las diferencias divididas de orden 1 son:

$$\begin{aligned} f[t_0, t_1] &= \frac{f[t_1] - f[t_0]}{t_1 - t_0} = \frac{2 - 2}{-1 - (-2)} = 0, \\ f[t_1, t_2] &= \frac{f[t_2] - f[t_1]}{t_2 - t_1} = \frac{1 - 2}{0 - (-1)} = -1, \\ f[t_2, t_3] &= \frac{f[t_3] - f[t_2]}{t_3 - t_2} = \frac{4 - 1}{1 - 0} = 3. \end{aligned}$$

Las diferencias divididas de orden 2 son:

$$\begin{aligned} f[t_0, t_1, t_2] &= \frac{f[t_1, t_2] - f[t_0, t_1]}{t_2 - t_0} = \frac{-1 - 0}{0 - (-2)} = -\frac{1}{2}, \\ f[t_1, t_2, t_3] &= \frac{f[t_2, t_3] - f[t_1, t_2]}{t_3 - t_1} = \frac{3 - (-1)}{1 - (-1)} = 2. \end{aligned}$$

La diferencia dividida de orden 3 es:

$$f[t_0,t_1,t_2,t_3] = \frac{f[t_1,t_2,t_3]-f[t_0,t_1,t_2]}{t_3-t_0} = \frac{2-(-1/2)}{1-(-2)} = \frac{5}{6}.$$

De esta forma, la tabla de diferencias dividas para el conjunto de datos dado se muestra en la Tabla 4.2.

Tabla 3.2. Tabla de diferencias divididas del Ejemplo 3.4

x_i	Diferencia dividida de orden 0	Diferencia dividida de orden 1	Diferencia dividida de orden 2	Diferencia dividida de orden 3
-2	2			
-1	2	0		
0	1	-1	$-1/2$	
1	4	3	2	$5/6$

Para realizar los cálculos con un computador, las diferencias divididas pueden almacenarse en una matriz triangular inferior $D \in \mathbb{R}^{n+1\times n+1}$, cuyos elementos están definidos como

$$d_{ij} = f[x_{i-j}, x_{i-j+1}, \ldots, x_i], \quad \text{para } j \le i.$$

Esta última relación se emplea para calcular recursivamente los elementos de D:

$$d_{i+1,1} = f[x_i], \quad \text{para } i = 0,1,\ldots,n,$$

$$d_{ij} = \frac{d_{i,j-1}-d_{i-1,j-1}}{x_{i-1}-x_{i-j}}, \quad \text{para } j = 2,3,\ldots,n+1,\ i = j, j+1,\ldots,n+1.$$

Note que el valor del coeficiente c_k dado en (3.14) es, por tanto, el elemento diagonal correspondiente $c_k = d_{k+1,k+1}$, para $k = 0,1,\ldots,n-1$. En el Programa 3.1 se ha implementado el cálculo de la matriz D de diferencias divididas.

Programa 3.1. Matriz de Diferencias Divididas

```
def difdiv(t,y):
    t = np.asarray(t,dtype=float)
    y = np.asarray(y,dtype=float)
    n,m = len(t),len(y)
    if n!=m:
        print('Error en el número de nodos')
        return
    D = np.zeros((n,n))
    D[:,0] = y
    for j in range(1,n):
        D[j:n,j] = (D[j:n,j-1]-D[j-1:n-1,j-1])/(t[j:n]-t[0:n-j])
    return D
```

Ejemplo 3.5. Construyamos el polinomio interpolante $p_3(t)$ en la forma de Newton de grado menor o igual a 3 asociado al conjunto de datos del Ejemplo 3.4, utilizando la tabla de diferencias divididas. La forma de Newton de $p_3(t)$ para los nodos dados es

$$p_3(t) = c_0 + c_1(t+2) + c_2(t+2)(x+1) + c_3(t+2)(t+1)t.$$

Ahora, en la Tabla 4.2 se aprecian las diferencias dividas del conjunto de datos dado. Luego, la matriz $D = (d_{ij})$ de diferencias divididas es

$$D = \begin{bmatrix} 2 & & & \\ 2 & 0 & & \\ 1 & -1 & -1/2 & \\ 4 & 3 & 2 & 5/6 \end{bmatrix},$$

además, utilizando la ecuación (3.14) los coeficientes c_k son:

$$c_0 = d_{11} = 2, \quad c_1 = d_{22} = 0, \quad c_2 = d_{33} = -\frac{1}{2}, \quad c_3 = d_{44} = \frac{5}{6}.$$

Por tanto, el polinomio interpolante en la forma de Newton es:

$$p_3(t) = 2 - \frac{1}{2}(t+2)(t+1) + \frac{5}{6}(t+2)(t+1)t.$$

Todos estos cálculos también podemos hacerlos usando la función `difdiv`, de la siguiente manera:

```
>>> import numpy as np
>>> t = [-2,-1,0,1]
>>> y = [2,2,1,4]
>>> D = difdiv(t,y)
>>> print(D)
[[ 2.          0.          0.          0.        ]
 [ 2.          0.          0.          0.        ]
 [ 1.         -1.         -0.5         0.        ]
 [ 4.          3.          2.          0.83333333]]
```

Ahora, los coeficientes c_k se obtienen como:

```
>>> c = np.diag(D)
>>> print(c)
[ 2.          0.         -0.5         0.83333333]
```

Observe que la forma de Newton del polinomio $p_3(t)$ del Ejemplo 3.3 es distinta a la forma de Newton hallada en el Ejemplo 3.5, pero estos polinomios son un único polinomio interpolante dado en su forma canónica $p_3(t) = 1 + \frac{1}{6}t + 2t^2 + \frac{5}{6}t^3$. Esta diferencia en la forma de Newton no es una incongruencia, sino que se origina por el ordenamiento de los nodos t_i. Para construir el polinomio $p_3(t)$ del Ejemplo 3.3, inicialmente se tenían los nodos $t_0 = -1$, $t_1 = 0$, $t_2 = 1$ y luego se agregó el nodo $t_3 = -2$; en cambio, para construir el polinomio $p_3(t)$ del Ejemplo 3.5, los nodos tomados fueron: $t_0 = -2$, $t_1 = -1$, $t_2 = 0$, $t_3 = 1$. Este cambio de orden de los nodos, modifica la forma de Newton del polinomio interpolante, pero sigue siendo un único polinomio interpolante.

Recordemos que la forma eficiente de evaluar un polinomio $p(t)$ es la regla de Ruffini o método de Horner de división. Gracias a las propiedades de la base triangular de Newton, en particular la dada en la ecuación (3.11), el método de Horner también se puede usar para evaluar el polinomio interpolante $p_n(t)$ en la forma de Newton. El Programa 3.2 muestra una adaptación del método Horner para evaluar en un punto z el polinomio interpolante en la forma de Newton.

Programa 3.2. Método de Horner para la forma de Newton

```
def nhorner(c,t,z):
    t = np.asarray(t,dtype=float)
    c = np.asarray(c,dtype=float)
    m = len(c)
    pval = c[m-1]
    for k in reversed(range(0,m-1)):
        pval = c[k] + (z-t[k])*pval
    return pval
```

3.1.4 Error de Interpolación

Dos preguntas importantes son las siguientes: ¿Qué sucede cuando el número de nodos de interpolación y el grado del polinomio interpolante aumentan? ¿El polinomio converge a la función? Un asunto relacionado a estas preguntas es el error de la aproximación; en particular, cuán grande debe ser el grado del polinomio de interpolación para lograr una precisión deseada. El siguiente teorema da una respuesta parcial a estas preguntas.

Teorema 3.2. *Sea $f(t)$ una función $n+1$ veces continuamente diferenciable en $[a,b]$. Sea $p_n(t)$ el polinomio de grado menor o igual a n que interpola a $f(t)$ en $n+1$ puntos distintos $t_0,t_1,\ldots,t_n$ en el intervalo $[a,b]$. Entonces, para cada $t\in[a,b]$ existe $\beta_t\in(a,b)$ tal que:*

$$f(t)=p_n(t)+\frac{1}{(n+1)!}f^{(n+1)}(\beta_t)\prod_{i=0}^{n}(t-t_i). \tag{3.15}$$

De la ecuación (3.15) uno puede obtener conclusiones útiles. Si $f(t)$ es $n+1$ veces continuamente diferenciable, se puede deducir que

$$\max_{a\le t\le b}|f(t)-p_n(t)|\le\frac{|b-a|^{n+1}}{(n+1)!}\max_{a\le t\le b}|f^{(n+1)}(t)|. \tag{3.16}$$

Si podemos encontrar una cota para la $(n+1)$-ésima derivada de $f(t)$, entonces (3.16) da una manera de garantizar teóricamente la precisión de la aproximación. Tales desigualdades son llamadas *cotas de error*, porque ellas nos permiten poner límites matemáticamente rigurosos sobre el error. Tome en cuenta, sin embargo, que (3.16) solamente es una cota que garantiza que el error es más pequeño que el lado derecho; con la computación actual el verdadero error pudiera ser de hecho mucho más pequeño.

Ejemplo 3.6. Sea $f(t)=\ln t$. Estimemos el error máximo de interpolación en el intervalo $[1,2]$, si aproximamos a $f(t)$ con el polinomio interpolante $p_3(t)$ asociado a la tabla de datos

t_i	1	4/3	5/3	2
$\ln(t_i)$	0	0.287682	0.510826	0.693147

Usando la ecuación (3.16), se tiene que una cota del error máximo de interpolación es:

$$\max_{1\le t\le 2}|\ln t-p_3(t)|\le\frac{1}{4!}\max_{1\le t\le 2}|f^{(4)}(t)|.$$

Ahora, la cuarta derivada de f es

$$f^{(4)}(t) = -\frac{6}{t^4},$$

luego,

$$\max_{1\leq t\leq 2} |f^{(4)}(t)| \leq 6.$$

Por lo tanto, una cota del error máximo de interpolación en el intervalo $[1,2]$ es:

$$\max_{1\leq t\leq 2} |\ln t - p_3(t)| \leq \frac{6}{4!} = 0.25$$

Esto significa que el máximo error en el intervalo $[1,2]$ es del orden 2.5×10^{-1}, si utilizamos el polinomio

$$p_3(t) = 0.1067175t^3 - 0.717291t^2 + 2.0979975t - 1.487424$$

que interpola la tabla de datos iniciales. En la siguiente tabla comparamos los valores exactos de $f(t)$ con los valores de $p_3(t)$ en los puntos $t = 1.25, 1.5, 1.75$.

t	$\ln t$	$p_3(t)$	$\lvert\ln t - p_3(t)\rvert$
1.25	0.223144	0.222738	0.000405
1.5	0.405465	0.405839	0.000374
1.75	0.559616	0.559307	0.000309

Observe que el error cometido en cada uno de los puntos $t = 1.25, 1.5, 1.75$ es menor que la cota del error máximo.

Pasemos ahora a estudiar la influencia que tiene la longitud del intervalo en el error de interpolación. Consideremos la misma función $f(t) = \ln t$, pero ahora estimemos el error máximo de interpolación en el intervalo $[1,4]$, si aproximamos a $f(t)$ con el polinomio interpolante $p_3(t)$ asociado a la siguiente tabla de datos.

t_i	1	2	3	4
$\ln(t_i)$	0	0.693147	1.098612	1.386294

En este caso, se tiene que

$$\max_{1\leq t\leq 4} |\ln t - p_3(t)| \leq \frac{3^4}{4!} \max_{1\leq t\leq 4} |f^{(4)}(t)|.$$

y

$$\max_{1\leq t\leq 4} |f^{(4)}(t)| \leq 6.$$

Luego, una cota del error máximo de interpolación en el intervalo $[1,4]$ es:

$$\max_{1\leq t\leq 4} |\ln t - p_3(t)| \leq \frac{3^4 6}{4!} = 20.25,$$

es decir, el máximo error en el intervalo $[1,4]$ es del orden 2.025×10^1, cuando se utiliza el el polinomio

$$p_3(t) = 0.0283165t^3 - 0.31374t^2 + 1.4361515t - 1.150728$$

que interpola la nueva tabla. Se observa un aumento significativo en la cota del error máximo con respecto al obtenido en el intervalo $[1,2]$. Esto también se aprecia cuando en la siguiente tabla comparamos los valores exactos de $f(t)$ con los valores de $p_3(t)$ en los puntos $t = 1.25, 1.5, 1.75$.

t	$\ln t$	$p_3(t)$	$\lvert \ln t - p_3(t) \rvert$
1.25	0.223144	0.209548	0.013595
1.5	0.405465	0.393152	0.012313
1.75	0.559616	0.553467	0.006149

En el ejemplo anterior verificamos que la aproximación de interpolación y su error dependen de la escogencia de los nodos de interpolación. El error es cero en los nodos de interpolación y por lo general más grande en puntos distantes de cualquiera de estos nodos. La escogencia habitual es tomar nodos *igualmente espaciados*, cuya distancia entre nodos consecutivos es igual a $h > 0$, denominado *tamaño de paso*. Los $(n+1)$ nodos t_i en el intervalo $[a,b]$ están definidos por:

$$t_i = a + ih, \quad i = 0, 1, \ldots, n,$$

donde $h = (b-a)/n$. En este caso, uno puede verificar que

$$\left| \prod_{i=0}^{n} (t - t_i) \right| \leq n! \, \frac{h^{n+1}}{4}, \quad \text{para } t \in [a,b]$$

y, de esta forma, el error máximo de interpolación es

$$\max_{a \leq t \leq b} |f(t) - p_n(t)| \leq \frac{h^{n+1}}{4(n+1)} \max_{a \leq t \leq b} |f^{(n+1)}(t)|. \tag{3.17}$$

Desafortunadamente, no podemos deducir de (3.17) que el error de interpolación tiende a cero cuando $n \to \infty$, a pesar de que el término $h^{n+1}/[4(n+1)]$ tiende a 0. En efecto, como se muestra en el Ejemplo 3.7, denominado *contraejemplo de Runge*, existen funciones $f(t)$ para las cuales el límite puede ser infinito, esto es

$$\lim_{n \to \infty} \max_{a \leq t \leq b} |f(t) - p_n(t)| = \infty.$$

Este sorprendente resultado indica que al aumentar el grado n del polinomio de interpolación no se obtiene necesariamente una mejor reconstrucción de $f(t)$.

Ejemplo 3.7 (Contraejemplo de Runge)**.** Supongamos que deseamos aproximar la siguiente función

$$f(t) = \frac{1}{1+t^2}, \qquad -5 \leq t \leq 5,$$

usando el polinomio interpolante $p_n(t)$ asociado a los $(n+1)$ puntos (t_i, y_i), con t_i nodos igualmente espaciados en el intervalo $[-5,5]$ y $y_i = f(t_i)$. Se puede comprobar la existencia de algunos puntos t en el intervalo de interpolación tal que

$$\lim_{n \to \infty} |f(t) - p_n(t)| \neq 0.$$

En particular, $p_n(t)$ diverge para $|t| > 3.63...$. Este fenómeno se evidencia particularmente en las vecindades de los puntos extremos del intervalo de interpolación, como se aprecia en la Figura 3.4, y se debe a la escogencia de nodos igualmente espaciados.

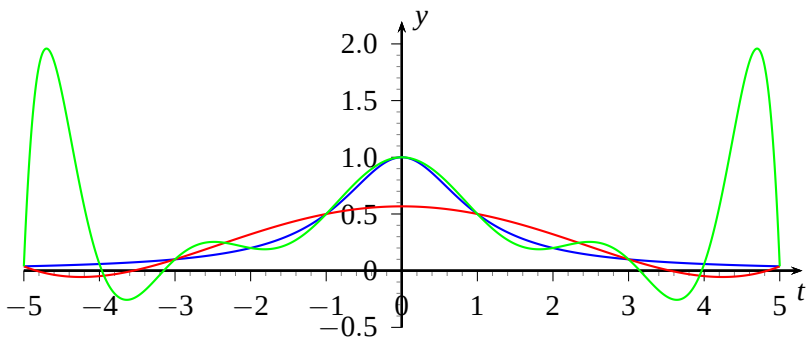


Figura 3.4. Polinomios interpolantes sobre nodos igualmente espaciados para la función $f(t) = 1/(1+t^2)$ (línea azul): polinomios interpolantes $p_5(t)$ (línea roja) y $p_{10}(t)$ (línea verde).

El fenómeno de Runge se puede evitar si se utiliza una distribución adecuada de los nodos. En particular, en un intervalo arbitrario $[a,b]$, uno puede considerar los llamados *nodos de Chebyshev*:

$$t_i = \frac{a+b}{2} - \frac{b-a}{2}\cos\left(i\frac{\pi}{n}\right), \quad i = 0,1,\ldots,n.$$

En efecto, para esta distribución especial de nodos es posible probar que, si $f(t)$ es una función continuamente diferenciable en $[a,b]$, entonces $p_n(t)$ converge a $f(t)$ cuando $n \to \infty$ para todo $t \in [a,b]$. Este resultado se establece en el siguiente teorema.

Teorema 3.3. *Sean $f(t)$ una función dos veces continuamente diferenciable en $[a,b]$ y $t_0,t_1,\ldots,t_n$ el conjunto de nodos de Chebyshev. Entonces, los polinomios interpolantes en esos nodos $p_n(t)$ cumplen*

$$\lim_{n\to\infty} \max_{a\le t\le b} |f(t) - p_n(t)| = 0.$$

Ejemplo 3.8. Consideremos de nuevo la función $f(t)$ del contraejemplo de Runge dado en el Ejemplo 3.7 y calculemos los polinomios interpolantes $p_5(t)$ y $p_{10}(t)$, usando los nodos de Chebyshev. En la Figura 3.5 se aprecian las gráficas de $p_5(t)$ y $p_{10}(t)$.

Un ejercicio interesante para el lector es hallar los polinomios $p_5(t)$ y $p_{10}(t)$, que interpola la función de Runge en los nodos de Chebyshev, y luego comparar los siguientes valores $f(4.5)$, $p_5(4.5)$ y $p_{10}(4.5)$.

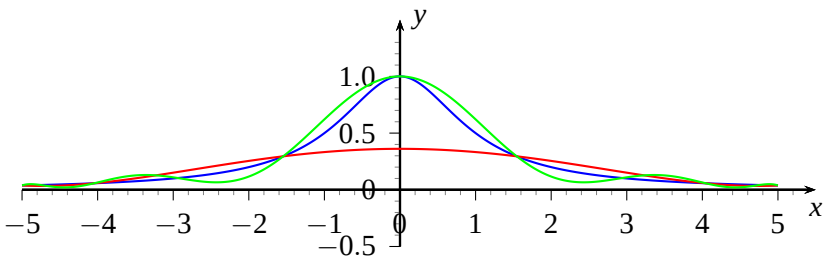


Figura 3.5. Polinomios interpolantes sobre nodos de Chebyshev para la función $f(t) = 1/(1+t^2)$ (línea azul): polinomios interpolantes $p_5(t)$ (línea roja) y $p_{10}(t)$ (línea verde).

A continuación describimos las funciones `polyfit` y `polyval` de la librería `numpy` que permiten, respectivamente, construir el polinomio interpolante y evaluarlo en puntos cualesquiera. En Python , uno puede guardar los puntos (t_i, y_i) en arreglos `t` e `y`, y luego el comando `polyfit(t,y,n)` de la librería `numpy` provee los coeficientes a_k del polinomio interpolante $p_n(t)$ en forma canónica dada por:

$$p_n(t) = a_1 t^n + a_2 t^{n-1} + \cdots + a_n t + a_{n+1}.$$

Ahora, con la instrucción `polyval(a,z)` de la librería `numpy` se calcula el valor que alcanza $p_n(t)$ en z. Veamos un ejemplo.

Ejemplo 3.9. Sea $f(t) = \ln t$. Construyamos el polinomio interpolante $p_3(t)$ asociado al conjunto de datos $(t_i, \ln t_i)$, donde t_i son nodos igualmente espaciados en el intervalo $[1,5]$. Recuerde, primero se debe cargar la librería `numpy`.

```
>>> import numpy as np
```

Debemos generar los 4 nodos t_i igualmente espaciados en el intervalo $[1,5]$. Para ello usamos la instrucción:

```
>>> t = np.linspace(1,5,4)
>>> print(t)
[ 1.          2.33333333  3.66666667  5.        ]
```

Ahora, con la siguiente instrucción construimos el arreglo `y`:

```
>>> y = np.log(t)
>>> print(y)
[ 0.          0.84729786  1.29928298  1.60943791]
```

Con la siguiente instrucción generamos los coeficientes del polinomio interpolante $p_3(t)$:

```
>>> a = np.polyfit(t,y,3)
>>> print(a)
[ 0.01782299 -0.23594265  1.26550262 -1.04738297]
```

Así, la forma canónica del polinomio interpolante es

$$p_3(t) = 0.01782299t^3 - 0.23594265t^2 + 1.26550262t - 1.04738297$$

Por ejemplo, para calcular el valor que alcanza $p_3(t)$ en $t = 2.5$, colocamos:

```
>>> np.polyval(a,2.5)
0.9202162937132583
```

Con la función `polyval` también podemos construir la gráfica $p_3(t)$. En la Figura 3.6 se muestra la gráfica de $p_3(t)$ en el intervalo$[1,5]$, la cual fue generada con las siguientes instrucciones:

```
>>> import matplotlib.pyplot as plt
>>> tt = np.linspace(-1,8,100)
>>> plt.plot(tt,np.log(tt),'r',tt,np.polyval(a,tt),'b',t,y,'oy')
>>> plt.grid()
>>> plt.legend(['$f(t)=\ln(t)$','$p_{3}(t)$','Puntos'])
```

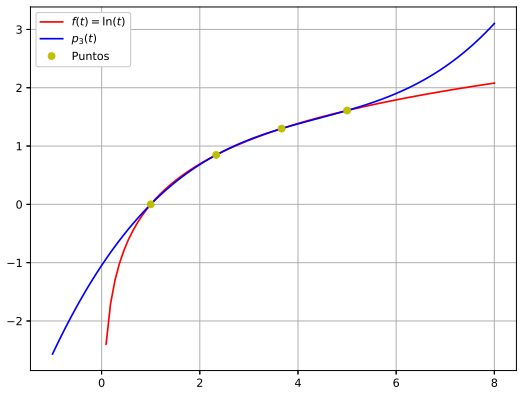


Figura 3.6. El polinomio interpolante de grado 3 introducido en el Ejemplo 3.9

Ejemplo 3.10. Consideremos nuevamente la función $f(t)$ del contraejemplo de Runge dado en el Ejemplo 3.7 y calculemos los polinomios interpolantes $p_5(t)$, $p_{10}(x)$ y $p_{14}(t)$ asociados a los nodos de Chebyshev en el intervalo $[-5,5]$, usando `polyfit`. En las siguientes instrucciones se generan los polinomios interpolantes $p_5(t)$, $p_{10}(t)$ y $p_{14}(t)$

```
>>> def f(t): return 1/(1+t**2)
>>> a,b = -5,5
>>> n = 5
>>> tc = -np.cos(np.pi*np.array(range(0,n+1))/n)
>>> t = (a+b)*0.5+(b-a)*tc*0.5
>>> c5 = np.polyfit(t,f(t),n)
>>> n = 10
>>> tc = -np.cos(np.pi*np.array(range(0,n+1))/n)
>>> t = (a+b)*0.5+(b-a)*tc*0.5
>>> c10 = np.polyfit(t,f(t),n)
>>> n = 14
>>> tc = -np.cos(np.pi*np.array(range(0,n+1))/n)
>>> t = (a+b)*0.5+(b-a)*tc*0.5
>>> c14 = np.polyfit(t,f(t),n)
```

Seguidamente se construyen las gráficas de los polinomios interpolantes $p_5(t)$, $p_{10}(t)$ y $p_{14}(t)$, las cuales se muestran en la Figura 3.7.

```
>>> tt = np.linspace(-5,5,500)
>>> plt.plot(tt,f(tt),'r',tt,np.polyval(c5,tt),tt,np.polyval(c10,tt),
 tt,np.polyval(c14,tt))
>>> plt.grid()
>>> plt.legend(['$f(t)=1/(1+t^{2})$','$p_{5}(t)$','$p_{10}(t)$','$p_{14}(t)$'])
```

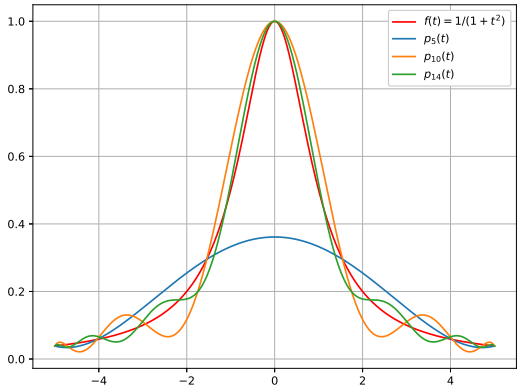


Figura 3.7. Polinomios interpolantes $p_5(t)$, $p_{10}(t)$ y $p_{14}(t)$ introducidos en el Ejemplo 3.10

3.1.5 Interpolación de Hermite

En muchas aplicaciones se hace necesario aproximar, por ejemplo, la primera o segunda derivada de una función o crear la gráfica de una curva suave que pasa por puntos de datos discretos. Para ello se requiere que el polinomio de interpolación satisfaga condiciones sobre las k-ésimas derivadas de la función, esto es, se debe encontrar un polinomio que interpole a $f(t)$, y también, a esas derivadas de manera que el polinomio imite lo mejor posible la función a aproximar. Este tipo de interpolación se le conoce con el nombre de *interpolación de Hermite*.

Asumamos que la función $f(t)$ posee derivada de cualquier orden. Dado un conjunto de nodos t_i con $i = 0,1,\ldots,n$, la interpolación de Hermite consiste en encontrar un polinomio $p(t)$ que satisfaga las k_i condiciones de interpolación:

$$p^{(j)}(t_i) = f^{(j)}(t_i), \tag{3.18}$$

para $j = 0,1,\ldots,k_i - 1$ y $i = 0,1,\ldots,n$. El polinomio $p(t)$ (si existe) que satisface las condiciones (3.18) se denomina *polinomio de Hermite*. El siguiente teorema presenta las condiciones de existencia y unicidad del polinomio de Hermite, además, indica el grado del mismo.

Teorema 3.4. *Sea $f(t)$ una función m-veces diferenciable en $\mathbb{R}$. Entonces, existe un único polinomio $p(x) \in \Pi_m$ que satisface las condiciones de interpolación* (3.18)*, donde m es un entero positivo tal que*

$$m+1 = \sum_{j=0}^{n} k_j. \tag{3.19}$$

El entero m que aparece en la ecuación (3.19) indica que el grado del polinomio de Hermite es a lo sumo m y el número $m+1$ es total de condiciones de interpolación. Ilustremos lo visto con un ejemplo.

Ejemplo 3.11. Sea $f(t)$ una función tal que los valores $f(0)=1$, $f'(0)=2$, $f(1)=3$, $f(2)=4$ y $f'(2)=1$ son conocidos. Determinemos el polinomio de Hermite que satisfaga las siguientes condiciones de interpolación:

$$\begin{array}{lll} p(0)=f(0)=1, & p(1)=f(1)=3, & p(2)=f(2)=4, \\ p'(0)=f'(0)=2, & p'(1)=f'(1)=2, & p'(2)=f'(2)=1. \end{array}$$

Como hay 6 condiciones, se puede suponer que el polinomio de Hermite $p(x)$ es a lo sumo de grado $m=5$; así, su forma canónica es

$$p(t)=a_0+a_1t+a_2t^2+a_3t^3+a_4t^4+a_5t^5 \quad \Rightarrow \quad p'(t)=a_1t+2a_2t+3a_3t^2+4a_4t^3.$$

De las condiciones de interpolación se sigue:

$$\begin{cases} a_0=1 \\ a_0+a_1+a_2+a_3+a_4+a_5=3 \\ a_0+2a_1+4a_2+8a_3+16a_4+32a_5=4 \\ a_1=2 \\ a_1+2a_2+3a_3+4a_4+5a_5=2 \\ 2a_1+4a_2+12a_3+32a_4+80a_5=1 \end{cases}$$

cuyas incógnitas son los coeficientes de $p(t)$. La solución de este sistema de ecuaciones lineales es el vector $\left(1,2,-\frac{1}{4},\frac{1}{2},-\frac{1}{4},0\right)$, por lo que el polinomio de Hermite buscado es:

$$p(t)=1+2t-\frac{1}{4}t^2+\frac{1}{2}t^3-\frac{1}{4}t^4,$$

cuya gráfica se muestra en la Figura 3.8.

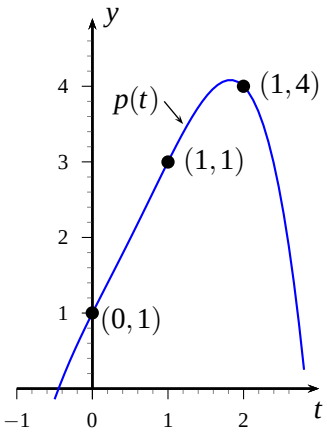


Figura 3.8. Polinomio de Hermite del Ejemplo 3.8

En la interpolación de Hermite pudiera suceder que la matriz resultante de la imposición de las condiciones de interpolación sea singular. En este caso el problema de interpolación de Hermite pudiera tener infinitas o ninguna solución, lo que complica su estudio. En el Ejemplo 3.8 se utilizó la forma canónica del polinomio de Hermite $p(t)$ para obtener un sistema de ecuaciones lineales, cuyas incógnitas eran precisamente los coeficientes $p(t)$. Pudiéramos haber hecho lo mismo, pero ahora utilizando la forma de Newton de $p(t)$ y se hubiese obtenido $p(t)$ expresado en la forma de Newton. En todo caso, siempre se debe resolver un sistema de ecuaciones lineales que está mal condicionado. Para evitar la resolución del sistema de ecuaciones lineales, se utiliza las *diferencias divididas con repeticiones* para hallar los coeficientes del polinomio de Hermite expresado en la forma de Newton. Para un estudio detallado de las diferencias dividas con repetición se recomiendan los libros [6], [14].

3.2 Interpolación Polinomial a Trozos

Una elección adecuada de los nodos de interpolación pueden mitigar algunas de las dificultades asociadas con la interpolación mediante un polinomio de grado grande. Sin embargo, es probable que un solo polinomio que interpole un gran número de puntos de datos produzca un comportamiento oscilante, lo cual sería un mal comportamiento en muchas aplicaciones. La interpolación polinómica a trozos proporciona una alternativa para solventar las dificultades prácticas y teóricas asociadas con la interpolación polinómica con un gran número de nodos. La ventaja de la interpolación polinomial a trozos es que una gran cantidad de puntos de datos, pueden ser interpolados con adecuados polinomios de grado bajo (1, 2 o 3).

En la interpolación polinomial a trozos de un conjunto dado de puntos (t_i, y_i), $i = 0, 1, \ldots, n$, con $t_1 < t_2 < \cdots < t_n$, se usa un polinomio diferente en cada subintervalo $I_i = [t_i, t_{i+1}]$. Por esta razón, las abscisas t_i, en las que los cambios interpolantes de un polinomio a otro, se denominan *nudos*, *puntos de interrupción* o *puntos de control*. El ejemplo más simple es la interpolación lineal por partes, en la que los puntos de datos sucesivos están conectados por líneas rectas. Formalmente, un polinomio interpolante a trozos se define como sigue ([6]).

Definición 3.2. Sea un conjunto dado de puntos (t_i, y_i), $i = 0, 1, \ldots, n$, con $t_1 < t_2 < \cdots < t_n$. Sean los n subintervalos definidos por $I_i = [t_i, t_{i+1}]$, para $i = 0, 1, \ldots, n$. Un *polinomio a trozos*, es una función $S(t)$ definida por:

$$S(t) = \begin{cases} S_0(t), & t \in I_0, \\ S_1(t), & t \in I_1, \\ \vdots & \\ S_{n-1}(t), & t \in I_{n-1}, \end{cases}$$

donde las funciones $S_j(t)$ son todas polinomios del mismo grado; además, $S(t)$ satisface

$$S(t_i) = y_i, \quad \text{para } i = 0, 1, \ldots, n.$$

Si el grado de cada $S_j(t)$ es igual a k, se dice que $S(t)$ es un polinomio interpolante a trozos de grado k.

Uno de los polinomios interpolantes a trozos más simples es el polinomio interpolante lineal a trozos, esto es, cuando todas las funciones $S_j(t)$ son un polinomio de grado 1 en cada subintervalo. Un polinomio lineal a trozos puede escribirse como:

$$S(t) = \begin{cases} b_0 + a_0 t, & t \in I_0, \\ b_1 + a_1 t, & t \in I_1, \\ \vdots & \\ b_{n-1} + a_{n-1} t, & t \in I_{n-1}, \end{cases} \tag{3.20}$$

además, de las condiciones $S(t_i) = y_i$ se sigue que

$$S_j(t_j) = y_j \qquad \text{y} \qquad S_{j+1}(t_{j+1}) = y_{j+1}.$$

Es decir, las funciones $S_j(t)$ están definidas, en cada subintervalo I_j, por el segmento de recta que une los puntos $(t_j, S_j(t_j))$ y $(t_{j+1}, S_{j+1}(t_{j+1}))$. Por tanto, utilizando la ecuación de una recta que pasa por dos puntos, obtenemos:

$$S_j(t) = y_j + \frac{y_{j+1} - y_j}{t_{j+1} - t_j}\,(t - t_j), \qquad \text{para } t_j \le t \le t_{j+1}.$$

de lo cual se infiere que

$$a_j = \frac{y_{j+1} - y_j}{t_{j+1} - t_j} \qquad \text{y} \qquad b_j = a_j t_j + y_j,$$

para $j = 0, 1, \ldots, n-1$. Es claro que el polinomio interpolante a trozos (3.20) es único y es comúnmente llamado *spline lineal*.

Ejemplo 3.12. Construyamos la forma general de un spline lineal $S^n(t)$ que interpole a una función $f(t)$ en los nodos $t_i = a + ih$, con $h = (b-a)/n$, $a < b$ e $i = 0, 1, \ldots, n$. Se tiene que $y_j = f(t_j)$ y $t_{j+1} = t_j + h$; luego,

$$a_j = \frac{y_{j+1} - y_j}{t_{j+1} - t_j} = \frac{f(t_j + h) - f(t_j)}{h} \qquad \text{y} \qquad b_j = \frac{f(t_j + h) - f(t_j)}{h}\, t_j + f(t_j),$$

para $j = 0, 1, \ldots, n-1$, de donde se deduce que

$$S_j(t) = b_j + a_j t = f(t_j) + \left(\frac{f(t_j + h) - f(t_j)}{h} \right) (t + t_j).$$

Así, el spline lineal buscado es

$$S^n(t) = \begin{cases} f(t_0) + \left(\frac{f(t_0+h) - f(t_0)}{h} \right) (t + t_0), & t \in I_0, \\ f(t_1) + \left(\frac{f(t_1+h) - f(t_1)}{h} \right) (t + t_1), & t \in I_1, \\ \vdots & \\ f(t_{n-1}) + \left(\frac{f(t_{n-1}+h) - f(t_{n-1})}{h} \right) (t + t_{n-1}), & t \in I_{n-1}, \end{cases}$$

Note que cuando n es suficientemente grande, h es sumamente pequeño; luego

$$f'(t_j) \approx \frac{f(t_j + h) - f(t_j)}{h}$$

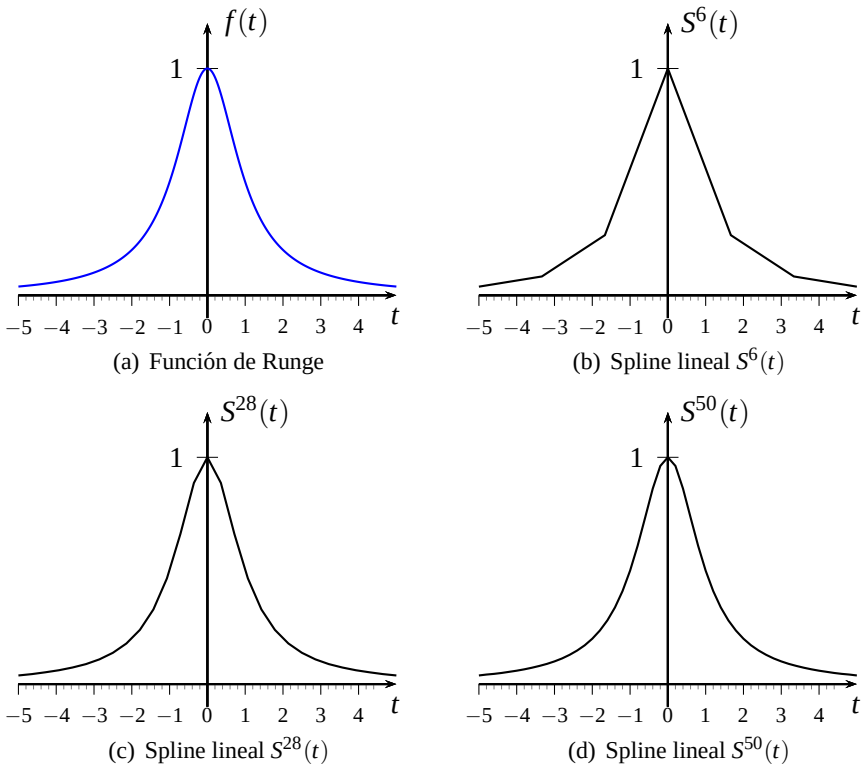
y, de esta forma,

$$S_j(t) \approx f(t_j) + f'(t_j)(t+t_j) = f(t_j) + f'(t_j)(t-t_j) - 2(a+jh)f'(t_j),$$

en otras palabras, cuando n es es lo suficientemente, $S_j(t)$ se comporta, para cada $t \in I_j$, como el desarrollo de Taylor de grado 1 de $f(t)$, centrado en t_j. En consecuencia, cuando n es lo suficientemente grande, $S^n(t) \approx f(t)$. Veamos que esto se cumple a través de un ejemplo numérico. Consideremos la función de Runge,

$$f(t) = \frac{1}{1+t^2}$$

y los nodos $t_i = -5 + ih$, con $h = 10/n$ e $i = 0, 1, \ldots, n$. En la Figura 3.9 se muestra las gráficas de la función de Runge y las funciones $S^6(t)$, $S^{28}(t)$ y $S^{50}(t)$, que son respectivamente los spline lineales que interpolan a la función de Runge en los nodos t_j, con $n = 6, 28, 50$.



(a) Función de Runge (b) Spline lineal $S^6(t)$

(c) Spline lineal $S^{28}(t)$ (d) Spline lineal $S^{50}(t)$

Figura 3.9. Sline lineales que interpolan a la función de Runge.

Note que efectivamente el spline $S^n(t)$ aproxima mejor a $f(t)$, a medida que n aumenta.

A pesar de que los spline lineales son tan sencillos, presentan buenas características: (i) son funciones continuas; y (ii) los spline lineales $S^n(t)$ que interpolan una función $f(t)$, convergen a $f(t)$ siempre que los nodos t_i estén igualmente espaciados. Pero, en general, los splines lineales no son funciones suaves. En efecto, un spline lineal puede tener pendientes diferentes en cada subintervalo I_k, y en consecuencia, en los nudos (puntos donde se unen los subintervalos) el spline no es derivable. Por tanto, los splines lineales son funciones continuas, cuya derivada existe solo en los subintervalos (t_j, t_{j+1}) y esto pudiera no ser conveniente en algunas aplicaciones.

3.2.1 Splines de orden k

Comúnmente, además de exigirle al polinomio interpolante a trozos que sea continua en el intervalo de interpolación, también se exige que si la función $S(t)$ es de grado k, entonces $S(t)$ debe tener $k-1$ derivadas continuas. En consecuencia, a medida que se aumente el grado del polinomio interpolante a trozos, aumenta la suavidad de $S(t)$. Los polinomios interpolantes a trozos que verifican esta condición se *denominan splines de grado k* o simplemente *spline*. La definición formal de un spline de grado k es la siguiente.

Definición 3.3. Sea un conjunto dado de puntos (t_i, y_i), $i = 0, 1, \ldots, n$, con $t_1 < t_2 < \cdots < t_n$. Sea k un entero positivo. Un spline de grado k, es un polinomio interpolante a trozos $S(t)$ de grado k, que satisface: la derivada j-ésima de $S(t)$ existe y es continua en el intervalo $[t_0, t_n]$, para $j = 0, 1, \ldots, k-1$.

Los spline de mayor uso son los *spline cúbicos* o spline de grado 3. Seguidamente se da la definición de spline cúbico.

Definición 3.4. Sea un conjunto dado de puntos (t_i, y_i), $i = 0, 1, \ldots, n$, con $t_1 < t_2 < \cdots < t_n$. Un *un spline cúbico*, es un polinomio a trozos $S(t)$ de grado 3 definido de la siguiente manera:

$$S(t) = \begin{cases} S_0(t), & t \in [t_0, t_1], \\ S_1(t), & t \in [t_1, t_2], \\ \vdots & \\ S_{n-1}(t), & t \in [t_{n-1}, t_n], \end{cases}$$

donde las funciones $S_j(t)$ son polinomios cúbicos en el intervalo $[t_j, t_{j+1}]$, que satisfacen las siguientes condiciones:

(i) *Condición de interpolación*:

$$S_j(t_j) = y_j, \quad \text{para } j = 0, 1, \ldots, n.$$

(ii) *Continuidad del spline en el intervalo* (t_0, t_n):

$$S_j(t_{j+1}) = S_{j+1}(t_{j+1}), \quad \text{para } j = 0, 1, \ldots, n-2.$$

(iii) *Continuidad de la primera derivada del spline en el intervalo* (t_0, t_n):

$$S'_j(t_{j+1}) = S'_{j+1}(t_{j+1}), \quad \text{para } j = 0, 1, \ldots, n-2.$$

(iv) *Continuidad de la segunda derivada del spline en el intervalo* (t_0, t_n):

$$S''_j(t_{j+1}) = S''_{j+1}(t_{j+1}), \quad \text{para } j = 0, 1, \ldots, n-2.$$

Utilicemos las condiciones (i)-(iv) para construir el spline cúbico $S(t)$. Sean los números z_j definidos como el valor que alcanza $S''(t)$ en cada nodo t_j, esto es,

$$z_j = S''(t_j), \quad j = 0,1,\ldots,n.$$

Como $S_j(t)$ es un polinomio de grado 3, entonces $S_j''(t)$ es un polinomio de grado ≤ 1, además, por la condición (iv), $S_j''(t)$ es la recta que pasa por los puntos (t_j, z_j) y (t_{j+1}, z_{j+1}), esto es,

$$S_j''(t) = z_j + \left(\frac{z_{j+1}-z_j}{t_{j+1}-t_j}\right)(t-t_j) = \frac{z_j}{h_j}(t_{j+1}-t) + \frac{z_{j+1}}{h_j}(t-t_j),$$

donde $h_j = t_{j+1}-t_j$, para $j = 0,1,\ldots,n-1$. Integrando en ambos lados de la última ecuación, obtenemos:

$$\begin{aligned} S_j'(t) &= \frac{z_j}{h_j}\int (t_{j+1}-t)\,dt + \frac{z_{j+1}}{h_j}\int (t-t_j)\,dt \\ &= -\frac{z_j}{h_j}\frac{(t_{j+1}-t)^2}{2} + \frac{z_{j+1}}{h_j}\frac{(t-t_j)^2}{2} + A_j. \end{aligned}$$

Integrando nuevamente

$$S_j(t) = \frac{z_j}{6h_j}(t_{j+1}-t)^3 + \frac{z_{j+1}}{6h_j}(t-t_j)^3 + A_j t + B_j,$$

donde A_j y B_j son constantes de integración. Tomando

$$C_j = \frac{B + A_j t_{j+1}}{h_j} \qquad \text{y} \qquad D_j = \frac{B_j + A_j t_j}{h_j},$$

la función $y = A_j t + B_j$ se puede expresar como

$$y = C_j(t-t_j) + D_j(t_{j+1}-t).$$

De esta forma,

$$S_j(t) = \frac{z_j}{6h_j}(t_{j+1}-t)^3 + \frac{z_{j+1}}{6h_j}(t-t_j)^3 + C_j(t-t_j) + D_j(t_{j+1}-t). \tag{3.21}$$

Utilizando convenientemente las condiciones de interpolación $S_j(t_j) = y_j$ y $S_j(t_{j+1}) = y_{j+1}$ en la ecuación (3.21), tenemos

$$y_{j+1} = \frac{z_{j+1}h_j^2}{6} + C_j h_j \qquad \text{y} \qquad y_j = \frac{z_j h_j^2}{6} + D_j h_j,$$

de donde se determinan las constantes C_j y D_j, esto es,

$$C_j = \frac{y_{j+1}}{h_j} - \frac{z_{j+1}h_j}{6} \qquad \text{y} \qquad D_j = \frac{y_j}{h_j} - \frac{z_j h_j}{6}.$$

Sustituyendo las expresiones halladas de C_j y D_j en (3.21), obtenemos:

$$\begin{aligned} S_j(t) = &\frac{z_j}{6h_j}(t_{j+1}-t)^3 + \frac{z_{j+1}}{6h_j}(t-t_j)^3 + \left(\frac{y_{j+1}}{h_j} - \frac{z_{j+1}h_j}{6}\right)(t-t_j) \\ &+ \left(\frac{y_j}{h_j} - \frac{z_j h_j}{6}\right)(t_{j+1}-t). \end{aligned} \tag{3.22}$$

Para determinar los valores de $z_1, z_2, \ldots, z_{n-1}$, utilizamos las condiciones de continuidad impuestas sobre $S'(x)$. En los nodos t_j se debe cumplir que $S'_{j-1}(t_j) = S'_j(t_j)$. Si derivamos la ecuación (3.22), obtenemos $S'_j(t)$. Evaluando en $t = t_j$ y luego simplificando, obtenemos:

$$S'_j(t_j) = -\frac{h_j}{3}z_j - \frac{h_j}{6}z_{j+1} - \frac{y_j}{h_j} + \frac{y_{j+1}}{h_j}. \tag{3.23}$$

De la misma manera, si empleamos la ecuación (3.22) para obtener $S'_{j-1}(t_j)$, llegamos a:

$$S'_{j-1}(t_j) = -\frac{h_{j-1}}{6}z_{j-1} + \frac{h_{j-1}}{3}z_j - \frac{y_{j-1}}{h_{j-1}} + \frac{y_j}{h_{j-1}}. \tag{3.24}$$

Por (3.23), (3.24) y la condición $S'_{j-1}(t_j) = S'_j(t_j)$, podemos escribir:

$$h_{j-1}z_{j-1} + 2(h_j + h_{j-1})z_j + h_j z_{j+1} = \frac{6}{h_j}(y_{j+1} - y_j) - \frac{6}{h_{j-1}}(y_j - y_{j-1}), \tag{3.25}$$

para $j = 1, 2, \ldots, n-1$. De esta forma, tenemos $n-1$ ecuaciones lineales y $n+1$ incógnitas $z_0, z_1, \ldots, z_n$. Se puede elegir z_0 y z_n de manera arbitraria y resolver el sistema de ecuaciones resultante para obtener $z_1, z_2, \ldots, z_{n-1}$. Las alternativas más empleadas para escoger z_0 y z_n son las siguientes:

- *Condiciones naturales o de frontera libre*: $S(t)$ debe satisfacer las siguientes condiciones de frontera de segundo orden:
$$S''(t_0) = S''(t_n) = 0,$$
de donde se obtiene $z_0 = z_n = 0$.

- *Condiciones de frontera fija*: $S(t)$ debe satisfacer las siguientes condiciones de frontera sobre la primera derivada:
$$S'(t_0) = f'(t_0), \qquad S'(t_n) = f'(t_n),$$
donde $f(t)$ es la función que se está interpolando.

Si se elige $z_0 = z_n = 0$, es decir, las condiciones naturales, la función spline resultante se llama *spline cúbico natural*. En este caso, el sistema de ecuaciones lineales (3.26) es simétrico, tridiagonal y diagonalmente dominante,

$$\begin{bmatrix} u_1 & h_2 & & & & & \\ h_1 & u_2 & h_2 & & & & \\ & h_2 & u_3 & h_3 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & h_{n-3} & u_{n-2} & h_{n-2} \\ & & & & h_{n-2} & u_{n-1} \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ \cdots \\ z_{n-2} \\ z_{n-1} \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \cdots \\ v_{n-2} \\ v_{n-1} \end{bmatrix}, \tag{3.26}$$

donde

$$\begin{aligned} h_j &= t_{j+1} - t_j, \\ u_j &= 2(h_j + h_{j-1}), \\ v_j &= \frac{6}{h_j}(y_{j+1} - y_j) - \frac{6}{h_{j-1}}(y_j - y_{j-1}). \end{aligned}$$

Este sistema es no singular, lo que implica la unicidad del spline cúbico natural; además, presenta ventajas para su resolución numérica con algunos de los métodos estudiados en el Capítulo 2.

Una vez obtenidos los números z_j mediante la resolución del sistema lineal (3.26), los polinomios cúbicos $S_j(t)$ se calculan por (3.22) o, equivalentemente, pueden ser evaluados eficientemente para $t \in [t_j, t_{j+1}]$, mediante la siguiente fórmula anidada:

$$S_j(t) = y_j + (t-t_j)\left[G_j + (t-t_j)(F_j + E_j(t-t_j))\right], \quad \text{para } j = 0,1,\ldots,n-1,$$

donde:

$$E_j = \frac{1}{6h_j}(z_{j+1}-z_j), \qquad F_j = \frac{z_j}{2}, \qquad G_j = -\frac{h_j}{6}z_{j+1} - \frac{h_j}{3}z_j + \frac{1}{h_j}(y_{j+1}-y_j),$$

para $j = 0,1,\ldots,n-1$.

La librería `Scipy` cuenta con una implementación muy eficiente de un spline cúbico, a través de la clase

```
scipy.interpolate.CubicSpline(t,y,axis=0,bc_type='not-a-knot',extrapolate=None)
```

que genera un spline cúbico $S(t)$. Con la opción `bc_type='natural'`, se genera el spline cúbico natural. Veamos un ejemplo.

Ejemplo 3.13. Construyamos el spline cúbico natural que interpola la tabla de datos

t_i	-3	-2.5	-1	0	1	2	3	3.5
y_i	1	-1	4	2	-5	9	7	-2

Procedemos de la siguiente manera.

```
>>> from scipy.interpolate import CubicSpline
>>> ti = np.array([-3,-2.5,-1,0,1,2,3,3.5])
>>> yi = np.array([1,-1 ,4,2,-5,9,7,-2])
>>> S = CubicSpline(ti , yi , bc_type='natural')
```

Hemos obtenido un objeto `S`, que representa un elemento de una clase y, para nuestro efecto, representa el spline cúbico natural $S(t)$, en otras palabras, con el objeto `S`, se puede calcular, por ejemplo, el valor de $S(1/2)$,

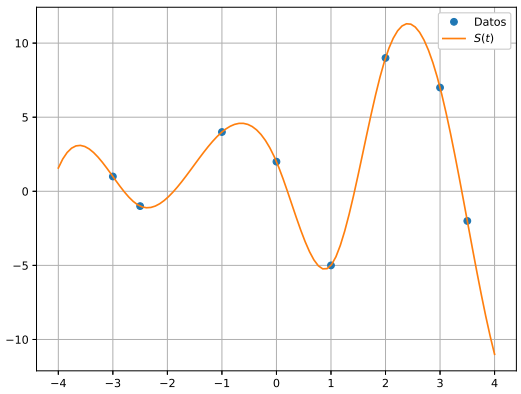
```
>>> S(1/2)
array(-3.15478348)
```

También podemos generar la gráfica de $S(t)$, por ejemplo, en el intervalo $[-4,4]$.

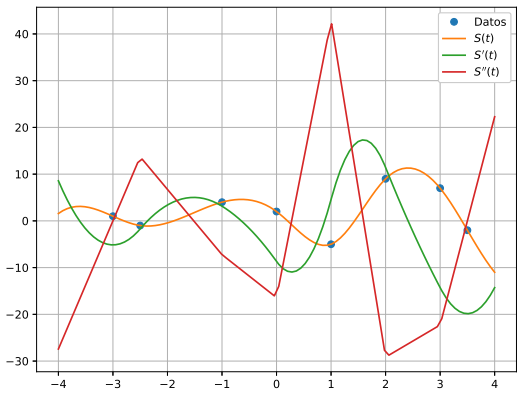
```
>>> t = np.linspace(-4,4,100)
>>> plt.plot(ti , yi , 'o', label='Datos')
>>> plt.plot(t, S(t), label="$S(t)$")
>>> plt.grid()
>>> plt.legend(loc='best')
```

En la Figura 3.10(a) se muestra la gráfica de $S(t)$. Además, podemos obtener las derivadas $S'(t)$, $S''(t)$ y $S'''(t)$, simplemente colocando `S(t,1)`, `S(t,2)` y `S(t,2)`, respectivamente. En la Figura 3.10(b) se muestran las gráficas de $S(t)$, $S'(t)$, $S''(t)$ y $S'''(t)$.

```
>>> plt.plot(ti , yi , 'o', label='Datos')
>>> plt.plot(t, S(t), label="$S(t)$")
>>> plt.plot(t, S(t,1), label="$S´(t)$")
>>> plt.plot(t, S(t,2), label="$S´´(t)$")
>>> plt.grid()
>>> plt.legend(loc='best')
```



(a) Gráfica de $S(t)$



(b) Gráficas de $S(t)$, $S'(t)$, $S''(t)$ y $S'''(t)$

Figura 3.10. Gráficas del spline cúbico $S(t)$ y de sus derivadas $S'(t)$, $S''(t)$ y $S'''(t)$, en el intervalo $[-4,4]$.

Finalizamos esta sección con un importante resultado de los spline cúbicos naturales. El spline cúbico natural posee la siguiente propiedad: entre todas las funciones con dos veces continuamente derivables en el intervalo $[a,b]$ que interpolan a $f(t)$ en un conjunto dado de nodos, el spline cúbico natural es el que minimiza la integral de la derivada segunda al cuadrado.

Teorema 3.5. *Sea $f(t)$ una función dos veces continuamente diferenciable en el intervalo $[a,b]$. Sean los nodos t_i tales que $a = t_0 < t_1 < \cdots < t_{n-1} < t_n = b$. Si $S(t)$ es el spline cúbico natural que interpola a $f(t)$ en esos nodos, entonces*

$$\int_a^b [S''(t)]^2, dt \leq \int_a^b [f''(t)]^2, dt.$$

El resultado anterior se conoce como la propiedad de *norma mínima* y tiene el significado del principio de energía mínima en mecánica.

3.3 Interpolación Trigonométrica y la FFT

Recordemos que dada una función $f(t)$ y $m+1$ puntos distintos, $t_0, t_1, \ldots, t_m$, existe un único polinomio $p_m(t)$ de grado $\leq m$ que interpola a $f(t)$, en otras palabras, $p_m(t_i) = f(t_i)$, para $i = 0, 1, \ldots, m$. Ahora bien, cuando se trata de representar funciones periódicas el espacio de los polinomios algebraicos no es el apropiado. En ese caso, el espacio más apropiado es el de las funciones trigonométricas. Antes de escoger las funciones trigonométricas básicas, recordemos el concepto de periodo de una función.

Definición 3.5. Una función $f(t)$ es *periódica* con periodo $T > 0$, si $f(t+T) = f(t)$ para todo $t \in \mathbb{R}$.

Por conveniencia, a lo largo de la sección nos centraremos en funciones de periodo 2π, ya que toda función $g(t)$ periódica con periodo T se puede expresar como un escalamiento de cierta función $f(t)$ periódica con periodo 2π. En efecto, dada $g(t)$ periódica con periodo T, la función $f(t)$ definida por $f(t) = g(\frac{Tt}{\pi})$ es periódica con periodo 2π, además, $g(t) = f(\frac{2\pi t}{T})$. De esta forma, las funciones 1, $\cos(x)$, $\cos(2x)$,..., y $\operatorname{sen}(x)$, $\operatorname{sen}(2x)$,..., son adecuadas para aproximar a una función $f(t)$ con periodo 2π.

Específicamente, estamos interesados en hallar un *polinomio trigonométrico* $P_M(t)$, que interpole a una función $f : [0, 2\pi] \to \mathbb{C}$ periódica con periodo 2π, en N nodos igualmente espaciados $t_n = 2\pi n/N$, para $n = 0, 1, \ldots, N-1$, es decir, se deben satisfacer las condiciones de interpolación:

$$P_M(t_n) = f(t_n), \qquad \text{para } n = 0, 1, \ldots, N-1.$$

El polinomio interpolante $P_M(t)$ se obtiene como una combinación lineal de funciones senos y cosenos. Cuando N es impar, el polinomio trigonométrico $P_M(t)$ está dado por:

$$P_M(t) = \frac{a_0}{2} + \sum_{k=1}^{M} (a_k \cos(kt) + b_k \operatorname{sen}(kt)), \tag{3.27}$$

donde $M = (N-1)/2$ y a_k, para $k = 0, 1, \ldots, M$, y b_k, para $k = 1, 2, \ldots, M$, son coeficientes complejos. En cambio, si N es par, el polinomio trigonométrico se escribe como:

$$P_M(t) = \frac{a_0}{2} + \sum_{k=1}^{M} (a_k \cos(kt) + b_k \operatorname{sen}(kt)) + a_{M+1} \cos((M+1)t). \tag{3.28}$$

donde $M = (N-2)/2$. Recordando la fórmula de Euler,

$$e^{jkt} = \cos(kt) + j \operatorname{sen}(kt),$$

con $j = \sqrt{-1}$ la unidad imaginaria, el polinomio trigonométrico (3.27), cuando N es impar, puede escribirse como

Se deja como ejercicio demostrar las fórmulas de la ecuación (3.29)

$$P_M(t) = \sum_{k=-M}^{M} c_k e^{jkt},$$

donde los coeficientes c_k, para $k = 0, 1, \ldots, M$, están relacionados con los coeficientes a_k y b_k a través de las fórmulas:

$$a_k = c_k + c_{-k}, \qquad b_k = j(c_k - c_{-k}). \tag{3.29}$$

Cuando N es par, el polinomio trigonométrico (3.28) puede escribirse como

$$P_M(t) = \sum_{k=-(M+1)}^{M+1} c_k e^{jkt},$$

donde $c_{M+1} = c_{-(M+1)} = a_{M+1}/2$.

En aras de una generalización, introducimos un parámetro μ que toma el valor de 0 si N es impar, y de 1, si N es par. Entonces, el polinomio trigonométrico $P_M(t)$ se puede escribir de una manera más general como

$$P_M(t) = \sum_{k=-(M+\mu)}^{M+\mu} c_k e^{jkt}. \tag{3.30}$$

Debido a su analogía con la serie de Fourier, $P_M(t)$ también se denomina *serie discreta de Fourier* de $f(t)$. Ahora, considerando las condiciones de interpolación en los nodos $t_n = nh$, con $h = 2\pi/N$, encontramos

$$P_M(t_n) = \sum_{k=-(M+\mu)}^{M+\mu} c_k e^{jknh} = f(t_n), \qquad \text{para } n = 0, 1, \ldots, N-1.$$

Pasemos a deducir una expresión cerrada para el cálculo de los coeficientes c_k. Multipliquemos en ambos lados de la ecuación anterior por e^{-jmnh}, donde m es un entero entre 0 y $N-1$, y luego sumamos con respecto a n,

$$\sum_{n=0}^{N-1} \sum_{k=-(M+\mu)}^{M+\mu} c_k e^{jknh} e^{-jmnh} = \sum_{n=0}^{N-1} f(x_n) e^{-jmnh}. \tag{3.31}$$

Demostremos la identidad

$$\sum_{n=0}^{N-1} e^{jnh(k-m)} = N \delta_{km}, \tag{3.32}$$

donde

$$\delta_{km} = \begin{cases} 0, & k \neq m, \\ 1, & k = m. \end{cases}$$

La identidad (3.32) es obvia cuando $k = m$. Supongamos que $k \neq m$. Se tiene que para todo $z \in \mathbb{C}$ tal que $z \neq 1$ se cumple

$$1 + z + z^2 + \cdots + z^{N-1} = \frac{1 - z^N}{1 - z}.$$

De esta forma, tomando $z = e^{jnh(k-m)} \neq 1$ en la ecuación anterior, encontramos que el lado izquierdo de la ecuación (3.32) adquiere la forma

$$\sum_{n=0}^{N-1} e^{jnh(k-m)} = \frac{1 - \left(e^{jh(k-m)}\right)^N}{1 - e^{jh(k-m)}}$$

la cual es igual a cero, ya que $h = 2\pi/N$ y

$$1 - \left(e^{jh(k-m)}\right)^N = 1 - \left(e^{\frac{j2\pi(k-m)}{N}}\right)^N = 1 - e^{2(k-m)\pi j} = 0.$$

Por lo tanto, la identidad (3.32) es cierta. Ahora, usando la identidad (3.32) podemos escribir:

$$\begin{aligned} \sum_{n=0}^{N-1} \sum_{k=-(M+\mu)}^{M+\mu} c_k e^{jknh} e^{-jmnh} &= \sum_{k=-(M+\mu)}^{M+\mu} c_k \sum_{n=0}^{N-1} e^{jnh(k-m)} \\ &= N \sum_{k=-(M+\mu)}^{M+\mu} c_k \delta_{km} \\ &= N c_m. \end{aligned} \tag{3.33}$$

Combinando las ecuaciones (3.31) y (3.33), se llega a la siguiente expresión cerrada de los coeficientes c_k,

$$c_k = \frac{1}{N} \sum_{n=0}^{N-1} f(t_n) \left(e^{-jkh}\right)^n, \qquad k = -(M+\mu), \ldots, (M+\mu). \tag{3.34}$$

Si $f(t)$ es una función real, entonces considerando que $\left(e^{jkh}\right)^n = \overline{(e^{-jkh})^n}$, por (3.34) tenemos, para $k = -(M+\mu), \ldots, (M+\mu)$,

$$c_{-k} = \frac{1}{N}\sum_{n=0}^{N-1} f(t_n)\left(e^{jkh}\right)^n = \overline{\frac{1}{N}\sum_{n=0}^{N-1} f(t_n)\,(e^{-jkh})^n} = \overline{c_k}.$$

Sustituyendo conveniente el valor de c_{-k} obtenido en las fórmulas de (3.29), podemos escribir:

$$a_k = c_k + c_{-k} = c_k + \overline{c_k} = 2\operatorname{Re} c_k, \qquad b_k = j(c_k - c_{-k}) = j(c_k - \overline{c_k}) = 2\operatorname{Im} c_k,$$

es decir, los coeficientes a_k y b_k son números reales. En consecuencia, si $f(t)$ es una función real, entonces el polinomio trigonométrico $P_M(t)$ dado en (3.30) es real.

De este modo, si $f(t)$ es real solo necesitamos calcular los coeficientes c_k para $k \geq 0$. Por lo tanto, cuando f es una función real los coeficientes c_k se calculan por la siguiente fórmula

$$c_k = \frac{1}{N} F_k, \qquad k = 0, 1, \ldots, N-1,$$

donde

$$F_k = \sum_{n=0}^{N-1} f(t_n) W_N^{kn}, \qquad k = 0, 1, \ldots, N-1, \tag{3.35}$$

se denomina *Transformada de Fourier Discreta* o DFT (del inglés, Discrete Fourier Transform) de f en N puntos o de orden N, con $W_N = e^{-j2\pi/N}$. Además, considerando las ecuaciones de interpolación $P_M(t_n) = f(t_n)$, obtenemos la siguiente expresión

$$f(t_n) = \frac{1}{N}\sum_{k=0}^{N-1} F_k W_N^{-kn}, \qquad n = 0, 1, \ldots, N-1. \tag{3.36}$$

denominada *Transformada Inversa de Fourier Discreta* o IDFT (del inglés, Inverse Discrete Fourier Transform).

Ejemplo 3.14. Construyamos el polinomio trigonométrico que interpola la tabla de datos

t_n	0	$2\pi/5$	$4\pi/5$	$6\pi/5$	$8\pi/5$
$f(t_n)$	6.8	3.2	-4.1	-3.9	3.3

Como $N = 5$ es par, entonces $M = (N-1)/2 = 2$ y el polinomio trigonométrico tiene la forma

$$P_2(t) = \frac{a_0}{2} + a_1\cos(t) + b_1\operatorname{sen}(t) + a_2\cos(2t) + b_2\operatorname{sen}(2t),$$

donde $a_0 = \frac{2}{5}\operatorname{Re} F_0$, $a_k = \frac{2}{5}\operatorname{Re} F_k$ y $b_k = \frac{2}{5}\operatorname{Im} F_k$, para $k = 1, 2$. Calculemos la DFT (3.35). Se tiene

$$\begin{cases} F_0 = f(t_0) + f(t_1) + f(t_2) + f(t_3) + f(t_4) \\ F_1 = f(t_0) + f(t_1)e^{-j2\pi/5} + f(t_2)e^{-j4\pi/5} + f(t_3)e^{-j6\pi/5} + f(t_4)e^{-j8\pi/5} \\ F_2 = f(t_0) + f(t_1)e^{-j4\pi/5} + f(t_2)e^{-j8\pi/5} + f(t_3)e^{-j12\pi/5} + f(t_4)e^{-j16\pi/5} \\ F_3 = f(t_0) + f(t_1)e^{-j6\pi/5} + f(t_2)e^{-j12\pi/5} + f(t_3)e^{-j18\pi/5} + f(t_4)e^{-j24\pi/5} \\ F_4 = f(t_0) + f(t_1)e^{-j8\pi/5} + f(t_2)e^{-j16\pi/5} + f(t_3)e^{-j24\pi/5} + f(t_4)e^{-j32\pi/5} \end{cases}$$

Tomando los valores $f(t_n)$ de la tabla, las ecuaciones anteriores adquieren forma:

$$\begin{cases} F_0 = 6.8 + 3.2 - 4.1 - 3.9 + 3.3 \\ F_1 = 6.8 + 3.2e^{-j2\pi/5} - 4.1e^{-j4\pi/5} - 3.9e^{-j6\pi/5} + 3.3e^{-j8\pi/5} \\ F_2 = 6.8 + 3.2e^{-j4\pi/5} - 4.1e^{-j8\pi/5} - 3.9e^{-j12\pi/5} + 3.3e^{-j16\pi/5} \\ F_3 = 6.8 + 3.2e^{-j6\pi/5} - 4.1e^{-j12\pi/5} - 3.9e^{-j18\pi/5} + 3.3e^{-j24\pi/5} \\ F_4 = 6.8 + 3.2e^{-j8\pi/5} - 4.1e^{-j16\pi/5} - 3.9e^{-j24\pi/5} + 3.3e^{-j32\pi/5} \end{cases}$$

De aquí se obtiene:

$$F_0 = 5.3,$$

$$F_1 = 15.280746418436735 + 0.21266270208800986\, j,$$

$$F_2 = -0.9307464184367371 - 0.1314327780297848\, j,$$

$$F_3 = -0.9307464184367369 + 0.13143277802978326\, j$$

$$F_4 = 15.280746418436737 - 0.2126627020880072\, j.$$

Por tanto, el polinomio trigonométrico que interpola la tabla de datos dada es:

$$\begin{aligned} P_2(t) =& 1.06 + 6.112298567374694\cos(t) + 0.08506508083520395\,\mathrm{sen}(t) \\ & - 0.37229856737469486\cos(2t) - 0.05257311121191392\,\mathrm{sen}(2t). \end{aligned}$$

La gráfica del polinomio $P_2(t)$ se muestra en la Figura 3.11.

Cuando $f(t)$ es real, el polinomio trigonométrico $P_M(t)$ se puede expresar, equivalentemente, por:

$$P_M(t) = \begin{cases} \dfrac{1}{N}F_0 + \dfrac{2}{N}\mathrm{Re}\left[\displaystyle\sum_{k=1}^{M} F_k e^{jkt}\right], & \text{si } N \text{ es impar,} \\ \dfrac{1}{N}F_0 + \dfrac{2}{N}\mathrm{Re}\left[\displaystyle\sum_{k=1}^{M} F_k e^{jkt} + \frac{1}{2}F_{M+1}e^{j(M+1)t}\right], & \text{si } N \text{ es par,} \end{cases} \tag{3.37}$$

donde F_k es la DFT dada por (3.35) y

$$M = \begin{cases} (N-1)/2, & \text{si } N \text{ es impar,} \\ (N-2)/2, & \text{si } N \text{ es par.} \end{cases}$$

La expresión (3.37) del polinomio trigonométrico permite que su implementación en Python sea muy sencilla, además, Python trabaja con aritmética compleja. Más adelante mostraremos una implementación del polinomio trigonométrico dado por (3.37).

Se aprecia de (3.35) que el cálculo de cada F_k resulta de evaluar un polinomio de grado $N-1$ en el punto W_N^k, lo cual se puede calcular con un costo aproximado de N multiplicaciones y N adiciones. En vista de que deben calcularse N números complejos F_k (observe que $F_0 = N\, f(t_0)$), el costo total de la construcción de la DFT es de N^2 operaciones punto-flotante. En la siguiente sección describiremos un algoritmo eficiente para calcular la DFT o la IDFT.

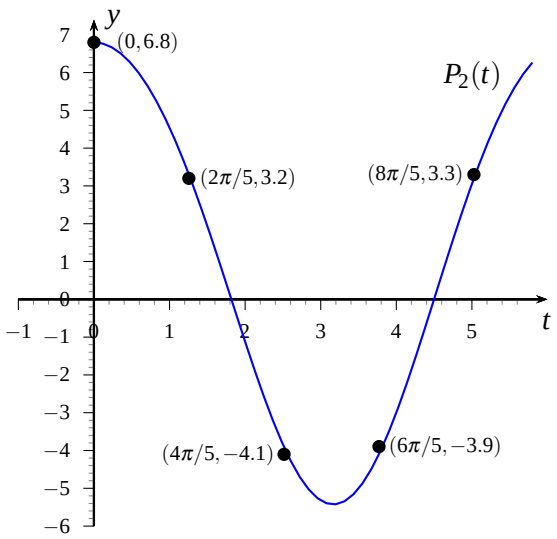


Figura 3.11. Polinomio trigonométrico $P_2(t)$ del Ejemplo 3.14

3.3.1 Transformada Rápida de Fourier

Existe un algoritmo muy eficiente para determinar la DFT (3.35), llamado *transformada rápida de Fourier*, el cual denotaremos FFT (del inglés, Fast Fourier Transform) por ser la abreviatura frecuentemente usada. La FFT hace que el costo del cómputo descienda de N^2 a una cantidad más razonable, $N\log_2 N$ operaciones punto-flotante. En la siguiente tabla mostramos lo que esto significa para valores grandes de N, que comúnmente aparecen en el procesamiento de señales.

N	N^2	$N\log_2 N$
1024	1048576	10240
2048	4194304	22528
4096	16777216	49152
8192	67108864	106496
16384	268435456	229376
32768	1073741824	491520

Describiremos el algoritmo de Cooley–Tukey [5]; en particular, mostraremos la implementación recursiva de la FFT cuando N es una potencia de 2. Para ver la derivación de este método es necesario considerar la siguiente notación. Tomemos $\mathbf{f} = (f_0, f_1, \ldots, f_{N-1})$ cuyas componentes son $f_n = f(t_n)$. De esta forma, cuando $N = 2^M$ la DFT (3.35) adquiere la forma:

$$\begin{aligned} F_k &= \sum_{n=0}^{2^M-1} f_n W_N^{kn} \\ &= \sum_{n=0}^{2^{M-1}-1} f_{2n}\, e^{-2\pi j \frac{(2n)k}{2^M}} + \sum_{n=0}^{2^{M-1}-1} f_{2n+1}\, e^{-2\pi j \frac{(2n+1)k}{2^M}} \\ &= \underbrace{\sum_{n=0}^{2^{M-1}-1} f_{2n}\, e^{-2\pi j \frac{nk}{2^{M-1}}}}_{=F_k^{par}} + \underbrace{e^{-2\pi j \frac{k}{2^M}} \sum_{n=0}^{2^{M-1}-1} f_{2n+1}\, e^{-2\pi j \frac{nk}{2^{M-1}}}}_{=F_k^{impar}}. \end{aligned} \tag{3.38}$$

Este resultado significa que la DFT se puede obtener calculando dos DFT de 2^{M-1} puntos cada una. Una de ellas con todos los índices pares f_{2n} y, la otra, con todos los índices impares f_{2n+1}, como nuevos vectores de datos. Este resultado se usa para derivar la forma recursiva de la FFT. El algoritmo de la forma recursiva de la FFT es un típico algoritmo de divide y vencerás; además, considera las siguientes ecuaciones:

$$F^{par}_{k+\frac{N}{2}} = F^{par}_{k}, \qquad F^{impar}_{k+\frac{N}{2}} = -F^{impar}_{k}, \qquad k = 0,1,\ldots,2^{M-1}-1. \tag{3.39}$$

Verifiquemos que (3.39) es cierta. Se tiene que

$$\begin{aligned} F^{par}_{k+\frac{N}{2}} &= F^{par}_{k+2^{M-1}} \\ &= \sum_{n=0}^{2^{M-1}-1} f_{2n}\, e^{-2\pi j \frac{n(k+2^{M-1})}{2^{M-1}}} \\ &= \sum_{n=0}^{2^{M-1}-1} f_{2n}\, e^{-2\pi j \frac{nk}{2^{M-1}}}\, e^{-2n\pi j} \\ &= \sum_{n=0}^{2^{M-1}-1} f_{2n}\, e^{-2\pi j \frac{nk}{2^{M-1}}} = F^{par}_{k}, \end{aligned}$$

$$\begin{aligned} F^{impar}_{k+\frac{N}{2}} &= F^{impar}_{k+2^{M-1}} \\ &= e^{-2\pi j \frac{k+2^{M-1}}{2^{M}}} \sum_{n=0}^{2^{M-1}-1} f_{2n+1}\, e^{-2\pi j \frac{n(k+2^{M-1})}{2^{M-1}}} \\ &= e^{-2\pi j \frac{k}{2^{M}}}\, e^{-\pi j} \sum_{n=0}^{2^{M-1}-1} f_{2n+1}\, e^{-2\pi j \frac{nk}{2^{M-1}}}\, e^{-2n\pi j} \\ &= -e^{-2\pi j \frac{k}{2^{M}}} \sum_{n=0}^{2^{M-1}-1} f_{2n+1}\, e^{-2\pi j \frac{nk}{2^{M-1}}} = -F^{impar}_{k}. \end{aligned}$$

De esta forma, considerando las fórmulas (3.38) y (3.39) se construye la forma recursiva de la FFT, cuya implementación se aprecia en el Programa 3.3.

Programa 3.3. FFT recursiva

```
def fft_r(f):
    N = f.shape[0]
    if N==1:
        return f
    elif N % 2 > 0:
        raise ValueError('La medida de f debe ser una potencia de 2')
        return
    else:
        F_par = fft_r(f[::2])
        F_impar = fft_r(f[1::2])
        z = np.exp(-2j * np.pi * np.arange(N) / N)
        return np.array(list(F_par + z[:int(N/2)] * F_impar) +  \
                        list(F_par + z[int(N/2):] * F_impar))
```

Podemos apreciar que la implementación del algoritmo recursivo de la FFT es bastante corta y fácil, y el número de operaciones punto-flotante es $N\log_2 N$. Sin embargo, el costo de almacenamiento de esta implementación es muy alto. La función `fft_r` se llama dos veces en cada

recursión, para ello se requiere almacenar N veces un arreglo de longitud 2^N. Esto conduce a un costo de almacenamiento en memoria del orden de $N2^N$. Para almacenar un número complejo, se necesitan dos variables de doble precisión, esto es, $64\,\text{Bit} = 8\,\text{Byte}$. Por ejemplo, para un vector $\mathbf{f}$ de alrededor de un millón de componentes, digamos $N = 2^{20} = 1048576$, el algoritmo necesita al menos $20 \cdot 1048576 \cdot 8 = 160\,\text{Megabyte}$ de almacenamiento en memoria.

Existen otras implementaciones de la FFT más eficientes en almacenamiento, las cuales se obtienen con factorizaciones matriciales. Las transformadas DFT e IDFT se pueden expresar en forma matricial. Tomemos los vectores

$$\mathbf{F} = (F_0, F_1, \ldots, F_{N-1}) \qquad \text{y} \qquad \mathbf{f} = (f_0, f_1, \ldots, f_{N-1}).$$

De esta forma, la DFT e IDFT se pueden expresar equivalentemente como:

$$\mathbf{F} = \mathbf{W}\mathbf{f} \qquad \text{y} \qquad \mathbf{f} = \frac{1}{N}\mathbf{W}^*\mathbf{F},$$

donde $\mathbf{W} \in \mathbb{C}^{N\times N}$ es la matriz definida por:

$$\mathbf{W} = \begin{bmatrix} W_N^{0\cdot 0} & W_N^{0\cdot 1} & \cdots & W_N^{0\cdot(N-1)} \\ W_N^{1\cdot 0} & W_N^{1\cdot 1} & \cdots & W_N^{1\cdot(N-1)} \\ \vdots & \vdots & \ddots & \vdots \\ W_N^{(N-1)\cdot 0} & W_N^{(N-1)\cdot 1} & \cdots & W_N^{(N-1)\cdot(N-1)} \end{bmatrix} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & W_N & \cdots & W_N^{(N-1)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & W_N^{(N-1)} & \cdots & W_N^{(N-1)^2} \end{bmatrix}$$

denominada *matriz de Fourier*, la cual posee las siguientes propiedades (entre otras):

i) $\mathbf{W}$ es simétrica.

ii) $\mathbf{W}^{-1} = \frac{1}{N}\mathbf{W}^*$, donde $\mathbf{W}^*$ es la *matriz traspuesta conjugada* de $\mathbf{W}$ que se obtiene tomando el conjugado de cada elemento de $\mathbf{W}$ y luego hallando la traspuesta de la matriz resultante.

Factorizando la matriz de Fourier $\mathbf{W}$ se obtienen implementaciones eficientes en almacenamiento, que se denominan *iterativas*. Las implementaciones prácticas de la FFT pueden manejar tanto la versión recursiva como la iterativa; por ejemplo, la función `fft` del módulo `fft` de la librería `Numpy`, considera las dos implementaciones. Además, en `fft` se implementa la FFT para un número de puntos cualquiera (N no es necesariamente una potencia de 2).

Programa 3.4. Polinomio Trigonométrico $P_M(t)$

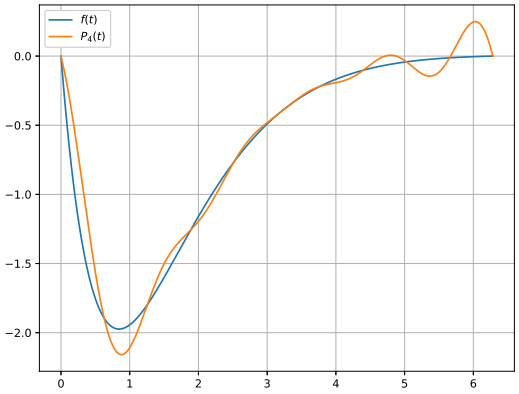
```
def PolTrig(N, F, t):
    impar = N % 2 > 0
    if impar:
        M = int((N-1)/2)
    else:
        M = int((N-2)/2)
    P = (0+0*1j)*t
    kc = np.arange(0, M+1, dtype=float)
    for i,k in enumerate(kc):
        P += (F[i] * np.exp(k * 1j * t))
    if not impar:
        P += (F[M+1]*np.exp((M + 1)*1j*t)/2)
    P = (2*P - F[0]).real/N
    return P
```

Los siguientes ejemplos muestran gráficamente la precisión de los polinomios trigonométricos al aproximar una función $f(t)$ en el intervalo $[0,2\pi]$. En algunos de ellos, el polinomio trigonométrico resulta ser una buena aproximación, pero en otros no. En estos ejemplos, utilizamos el Programa 3.4 en donde se implementa el polinomio trigonométrico (3.37) a través de la función `PolTrig`, cuyos parámetros son: `N`, el número de nodos; `F`, un arreglo que guarda los valores de la DFT (normalmente calculada usando la función `fft` de `Numpy`); y `t`, un escalor o un arreglo donde se desea evaluar el polinomio trigonométrico.

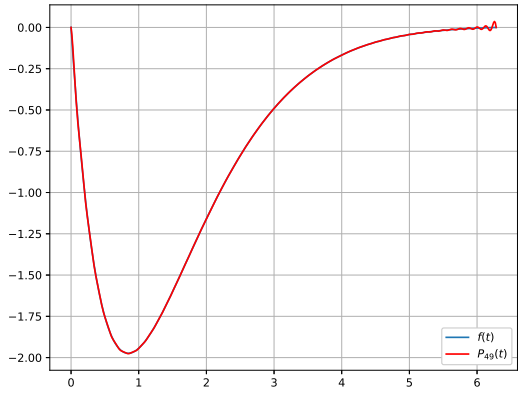
Ejemplo 3.15. Consideremos la función $f(t) = t(t-2\pi)e^{-t}$ para $t \in [0,2\pi]$. Usamos la función `PolTrig` para determinar el polinomio trigonométrico $P_M(t)$ que interpola a $f(t)$ en los nodos $t_n = 2n\pi/N$, para $n = 0,1,\ldots,N$, tomando dos casos: $N = 10$ y $N = 100$. Tenga presente que, para usar la función `PolTrig` del Programa 3.4, debe calcular previamente la DFT; por ejemplo, para el caso de $N = 10$, se procede de la siguiente manera.

```
>>> def f(t): return t*(t-2*np.pi)*np.exp(-t)
>>> N = 10
>>> tn = 2*np.pi*np.arange(N)/N
>>> fn = f(tn)
>>> F = np.fft.fft(fn)  #Cálculo de la DFT
```

Luego de calcular la DFT, se ejecuta la función `PolTrig`. Los resultados de la aproximación en los dos casos de estudio se muestran en la Figura 3.12.



(a) La función $f(t)$ (línea azul) y el polinomio trigonométrico $P_4(t)$ (línea naranja)



(b) La función $f(t)$ y el polinomio trigonométrico $P_{49}(t)$ (línea roja)

Figura 3.12. La función $f(t)$ y los polinomios trigonométricos $P_4(t)$ y $P_{49}(t)$, correspondientes a 10 y 100 puntos, respectivamente.

Note que el polinomio $P_4(t)$ se encuentra cerca de $f(t)$ en el centro del intervalo, mientras que comienza a oscilar en los extremos del intervalo. En cambio, $P_{49}(t)$ muestra una buena aproximación prácticamente en todo en el intervalo, salvo cerca del extremo derecho donde comienza levemente a oscilar.

Las oscilaciones del polinomio trigonométrico en los extremos del intervalo pueden aumentar drásticamente para ciertas funciones. Veamos el siguiente ejemplo.

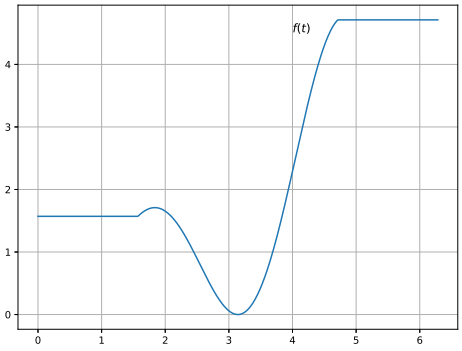
Ejemplo 3.16. Hallemos el polinomio trigonométrico $P_M(t)$ que interpola a

$$f(t) = \begin{cases} \pi/2, & 0 \le t < \pi/2, \\ t\cos^2(t-\pi/2), & \pi/2 \le t \le 3\pi/2, \\ 3\pi/2, & 3\pi/2 < t \le 2\pi, \end{cases}$$

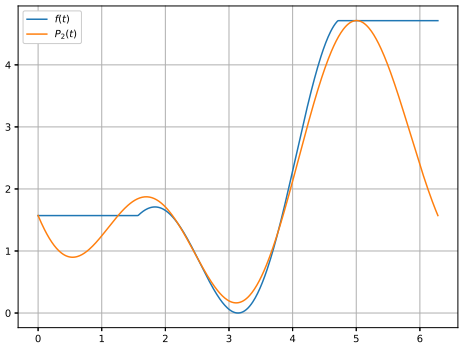
en los nodos $t_n = 2\pi n/N$, para $n = 0, 1, \ldots, N$, tomando tres casos: $N = 5$, $N = 20$ y $N = 50$. Con la siguiente instrucción se puede definir $f(t)$:

```
def f(t):
    return np.piecewise(t,[t<0,(t>=0)&(t<np.pi/2),(t>np.pi/2)&(t<=3*np.pi/2),
                           (t>3*np.pi/2)&(t<=2*np.pi),t>2*np.pi], [0,np.pi/2,
                           lambda t:t*np.cos(t-np.pi/2)**2,3*np.pi/2,0])
```

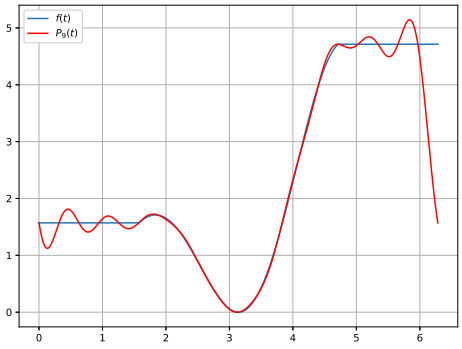
Usamos la función `PolTrig` para determinar los polinomios trigonométricos $P_2(t)$, $P_9(t)$ y $P_{24}(t)$, que interpolan a $f(t)$ en $N = 5, 20$ y 50 puntos igualmente espaciados, respectivamente. En la Figura 3.13 se observan los resultados obtenidos. Observe que a medida que el número de nodos aumenta, el polinomio trigonométrico aproxima mejor a $f(t)$, pero las oscilaciones en los extremos también aumentan.



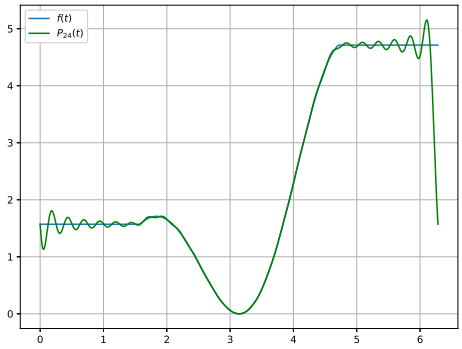
(a) La función $f(t)$ (línea azul)



(b) La función $f(t)$ y el polinomio trigonométrico $P_2(t)$ (línea naranja)



(c) La función $f(t)$ y el polinomio trigonométrico $P_9(t)$ (línea roja)



(d) La función $f(t)$ y el polinomio trigonométrico $P_{24}(t)$ (línea verde)

Figura 3.13. La función $f(t)$ y los polinomios trigonométricos $P_2(t)$, $P_9(t)$ y $P_{24}(t)$, correspondientes a 5, 20 y 50 puntos, respectivamente.

En algunos casos, la precisión de la interpolación trigonométrica puede reducirse drásti-camente, como se muestra en el siguiente ejemplo.

Ejemplo 3.17. Utilicemos la función `PolTrig` para aproximar la función $f(t) = f_1(t) + f_2(t)$, donde $f_l(t) = \text{sen}(t)$ y $f_2(t) = \text{sen}(5t)$, usando ocho nodos igualmente espaciados en el intervalo $[0, 2\pi]$. El resultado se muestra en la Figura 3.14. Note que existe una gran imprecisión del polinomio trigonométrico $P_3(t)$, incluso en algunos intervalos muestra una inversión de fase con respecto a la función $f(t)$.

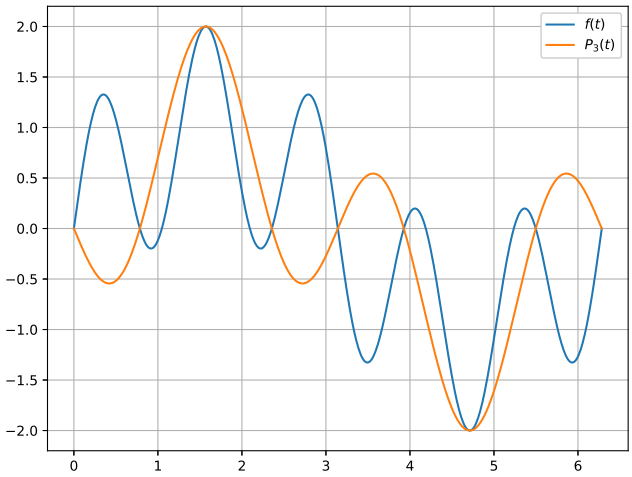


Figura 3.14. Los efectos del solapamiento: comparación entre la función $f(t) = \text{sen}(t) + \text{sen}(5t)$ (línea azul) y su interpolante trigonométrico $P_3(t)$ (línea naranja).

La falta de precisión del polinomio trigonométrico del Ejemplo 3.17, se puede explicar de la siguiente manera. En los nodos considerados, la función $f_2(t)$ coincide con $f_3(t) = -\text{sen}(3t)$ que tiene una frecuencia más baja (ver la Figura 3.15). Por tanto, $F(t) = f_1(t) + f_3(t)$ es la función que es aproximada y no $f(t)$; de hecho, el polinomio $P_3(t)$ coincide con $F(t)$. El fenómeno descrito se conoce como *solapamiento* y puede ocurrir cuando $f(t)$ es la suma de varios componentes que tienen diferentes frecuencias. Cuando el número de nodos no es suficiente para resolver las frecuencias más altas, estas últimas pueden interferir con las frecuencias bajas, originando polinomios trigonométricos inexactos. Para obtener una mejor aproximación de las funciones con frecuencias más altas, uno tiene que aumentar el número de nodos de interpolación.

3.3.2 Interpolación Trigonométrica en un Intervalo $[a, b)$

Sea un conjunto de datos (x_n, y_n), $n = 0, 1, \ldots, N$, igualmente espaciados, esto es, los nodos $x_n = n(b-a)/N$ están igualmente espaciados en el intervalo $[a, b)$. Se puede construir un polinomio trigonométrico que interpola el conjunto de datos dados en el intervalo $[a, b]$. Primero se construye el polinomio trigonométrico $P(t)$ que interpola los puntos (t_n, y_n), donde $t_n = 2\pi n/N$ son nodos igualmente espaciados en el intervalo $[0, 2\pi]$, para $n = 0, 1, \ldots, N$. Luego, se define el polinomio trigonométrico

$$Q(t) = P\left(\frac{2\pi}{(b-a)}t\right), \quad t \in [a, b).$$

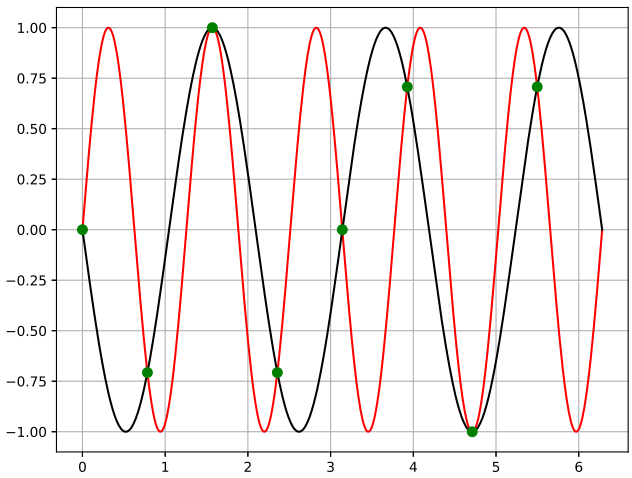


Figura 3.15. Fenómeno de solapamiento: las funciones $\text{sen}(5x)$ (línea roja) y $-\text{sen}(3x)$ (línea negra) toman los mismos valores en los nodos de interpolación. Esta circunstancia explica la grave pérdida de precisión que se muestra en la Figura 3.14.

Es sencillo verificar que $Q(x_n) = y_n$, para $n = 0, 1, \ldots, N-1$. Veamos un ejemplo.

Ejemplo 3.18. Consideremos la siguiente tabla de datos

x_i	y_i	x_i	y_i
0.0	2.95	0.8	-1.57
0.1	2.01	0.9	-1.12
0.2	0.33	1.0	-0.58
0.3	0.71	1.1	-0.69
0.4	0.11	1.2	-0.21
0.5	0.92	1.3	-0.54
0.6	-0.16	1.4	-0.63
0.7	0.68	1.5	-2.09

Se tienen $N = 16$ nodos $x_i \in [0, 1.6)$, para $i = 0, 1, \ldots, 15$. Utilicemos la función `PolTrig` para construir el polinomio $Q(t)$. Se procede de la siguiente manera.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from numpy.fft import fft
>>> N = 16
>>> a = 0;    b = 1.6                # Intervalo [a,b)
>>> xn = np.arange(0,1.6,0.1)        # Nodos igualmente espaciados en [0,1.6)
>>> yn = np.array([2.95,2.01,0.33,0.71,0.11,0.92,-0.16,0.68,-1.57,-1.12,-0.58,
                   -0.69,-0.21,-0.54,-0.63,-2.09])
>>> F = fft(yn)                                  # Cálculo de la DFT
>>> x = np.linspace(a,b-0.1,100)
>>> Qt = PolTrig(N, F, 2*np.pi*x/(b-a))  # Construcción de Q(t)
>>> plt.plot(xn,yn,'o', label='Puntos')
>>> plt.plot(x,Qt, label='$Q(t)$')
>>> plt.grid()
>>> plt.legend()
```

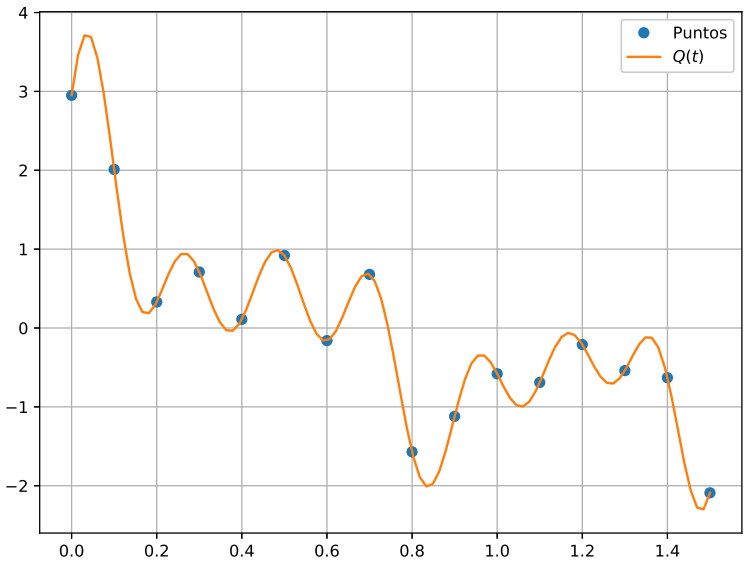


Figura 3.16. Polinomio trigonométrico $Q(t)$ que interpola la tabla de datos del Ejemplo 3.18.

En la Figura 3.16 se muestra la gráfica del polinomio trigonométrico $Q(t)$ que interpola el conjunto de datos (x_n, y_n), $n = 0, 1, \ldots, N$, igualmente espaciados.

3.4 Ajuste de Curvas

En las secciones previas utilizamos polinomios o funciones trigonométricas para aproximar una función a través del conocimiento de los valores que la función alcanza en un número finito de puntos t_i. Exigíamos que la función interpolante y la función coincidieran en los nodos t_i. En la mayoría de las aplicaciones de ingeniería, esta última exigencia no es conveniente, ya que los datos provienen de mediciones experimentales que generalmente tienen cierto error. Además, como el número de datos experimentales normalmente es elevado, el grado del polinomio interpolante asociado a esos datos sería muy grande, lo cual produciría un gran error en las aproximaciones debido a las oscilaciones del polinomio.

En esta sección introduciremos el enfoque de *cuadrados mínimos* para aproximar un conjunto de puntos (t_i, y_i) con una función $f(t)$ denominada *función de ajuste*; en otras palabras, nuestro objetivo es determinar una fórmula $y = f(t)$ que relacione los coordenadas t_i e y_i, de manera que $f(t_i)$ sea lo más parecida a y_i en algún sentido que debe ser precisado. Inicialmente, describiremos el *ajuste polinomial* que consiste en utilizar un polinomio como función de ajuste. Luego utilizaremos el ajuste polinomial para ajustar funciones no lineales, mediante el proceso de *linealización*.

3.4.1 Ajuste Polinomial de Datos

Asumamos que el conjunto de n puntos (t_i, y_i), $i = 1, 2, \ldots, n$, está disponible. Deseamos hallar una función de ajuste $f(t)$, cuyo modelo matemático contiene $m+1$ parámetros reales $x_0, x_1, \ldots, x_m$, de tal manera que $f(t_i)$ aproxime lo "mejor posible" a y_i. Tal aproximación se puede garantizar encontrando valores de los parámetros $x_0, x_1, \ldots, x_m$ de manera que la diferencia entre y_i

y $f(t_i)$ sea la más pequeña posible, es decir, el valor de $|f(t_i)-y_i|$ debe ser relativamente pequeño para todo $i=1,2,\ldots,n$ (ver Figura 3.17).

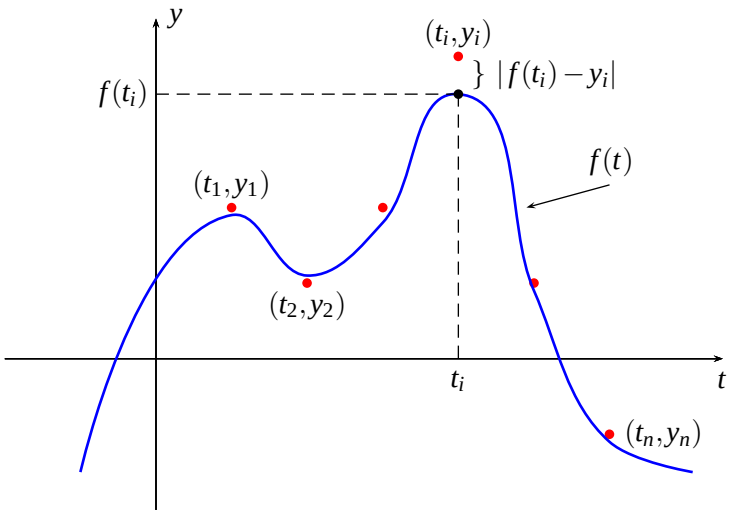


Figura 3.17. Ajuste por cuadrados mínimos

En particular, hallaremos valores de los parámetros $x_0,x_1,\ldots,x_m$ utilizando el enfoque de *cuadrados mínimos*, que consiste en encontrar valores de los parámetros tal que minimicen la suma de las diferencias al cuadrado de y_i y $f(t_i)$. Matemáticamente, dado el vector de parámetros $\mathbf{x}=(x_0,x_1,\ldots,x_m)$, definamos una función $g:\mathbb{R}^{m+1}\to\mathbb{R}$ como:

$$g(\mathbf{x})=\frac{1}{2}\sum_{i=1}^{n}(f(\mathbf{x};t_i)-y_i)^2, \tag{3.40}$$

donde $f(\mathbf{x};t)$ denota la función $f(t)$ evaluada en t, empleando los parámetros $x_0,\ldots,x_m$.

El enfoque de cuadrados mínimos se resume en hallar un mínimo de $g(\mathbf{x})$, es decir, resolver el problema de minimización:

$$\min_{\mathbf{x}\in\mathbb{R}^{m+1}} g(\mathbf{x})\equiv\frac{1}{2}\sum_{i=1}^{n}(f(\mathbf{x};t_i)-y_i)^2. \tag{3.41}$$

Es bien sabido que toda solución del problema (3.41) es un punto estacionario de g; por lo que el gradiente de $g(\mathbf{x})$ se anula en toda solución del problema de minimización, esto es, si $\mathbf{x}\in\mathbb{R}^{m+1}$ es una solución de (3.41), entonces se satisface:

$$\nabla g(\mathbf{x})=0,$$

en otras palabras,

$$\begin{cases}\dfrac{\partial g}{\partial x_0}(\mathbf{x})=0\\ \dfrac{\partial g}{\partial x_1}(\mathbf{x})=0\\ \vdots\\ \dfrac{\partial g}{\partial x_m}(\mathbf{x})=0\end{cases} \tag{3.42}$$

De esta forma, para hallar una solución del problema (3.41) basta con resolver el sistema de ecuaciones (3.42), cuyas incógnitas son los parámetros $x_0, x_1, \ldots, x_m$. Los términos $r_i = f(t_i) - y_i$ en (3.40) se denominan *residuales*; ellos representan la discrepancia entre los datos y la función de ajuste en t_i. Las ecuaciones de (3.42) son generalmente no lineales en x_j; en este caso, se dice que el problema (3.41) es un problema de *cuadrados mínimos no lineales*. De lo contrario, cuando el sistema (3.42) es lineal, se dice que (3.41) es un problema de *cuadrados mínimos lineales*.

Polinomio de Cuadrados Mínimos

En esta sección nos centraremos en hallar la solución del sistema (3.42) para los casos en que la función de ajuste tiene una forma particular, de tal manera que el sistema (3.42) sea lineal. Si la función de ajuste $f(t)$ es un polinomio, entonces el sistema (3.41) es lineal. En efecto, cuando $f(t)$ es un polinomio de grado m,

$$f(t) = x_0 + x_1 t + \cdots + x_m t^m,$$

la función $g(\mathbf{x})$ dada en (3.40) adquiere la forma:

$$g(\mathbf{x}) = \frac{1}{2}\sum_{i=1}^{n}(x_0 + x_1 t_i + \cdots + x_m t_i^m - y_i)^2.$$

Calculando las derivadas parciales $\partial g/\partial x_j$ se tiene:

$$\begin{aligned}
\frac{\partial g}{\partial x_0}(\mathbf{x}) &= \sum_{i=1}^{n}(x_0 + x_1 t_i + \cdots + x_m t_i^m - y_i),\\
\frac{\partial g}{\partial x_1}(\mathbf{x}) &= \sum_{i=1}^{n} t_i(x_0 + x_1 t_i + \cdots + x_m t_i^m - y_i),\\
&\vdots\\
\frac{\partial g}{\partial x_m}(\mathbf{x}) &= \sum_{i=1}^{n} t_i^m(x_0 + x_1 t_i + \cdots + x_m t_i^m - y_i).
\end{aligned}$$

Sustituyendo las expresiones de estas derivadas parciales en el sistema (3.42), obtenemos las bien conocidas *ecuaciones normales*:

$$\left\{\begin{array}{rcl}
nx_0 + x_1\sum_{i=1}^{n} t_i + \cdots + x_m\sum_{i=1}^{n} t_i^m &=& \sum_{i=1}^{n} y_i\\
x_0\sum_{i=1}^{n} t_i + x_1\sum_{i=1}^{n} t_i^2 + \cdots + x_m\sum_{i=1}^{n} t_i^{m+1} &=& \sum_{i=1}^{n} y_i t_i\\
&\vdots&\\
x_0\sum_{i=1}^{n} t_i^m + x_1\sum_{i=1}^{n} t_i^{m+1} + \cdots + x_m\sum_{i=1}^{n} t_i^{2m} &=& \sum_{i=1}^{n} y_i t_i^m
\end{array}\right. \tag{3.43}$$

Estas ecuaciones conforman un sistema lineal de $m+1$ ecuaciones, cuyas $m+1$ incógnitas son los coeficientes $x_0, x_1, \ldots, x_m$ del polinomio de ajuste $f(t)$, denominado *polinomio de cuadrados mínimos*. Cuando $m = n$, el polinomio de cuadrados mínimos coincide con el polinomio interpolante.

Al expresar las ecuaciones normales (3.43) en forma matricial, se observa un patrón interesante en la matriz de los coeficientes:

$$\begin{bmatrix} n & \sum_{i=1}^{n} t_i & \sum_{i=1}^{n} t_i^2 & \cdots & \sum_{i=1}^{n} t_i^m \\ \sum_{i=1}^{n} t_i & \sum_{i=1}^{n} t_i^2 & \sum_{i=1}^{n} t_i^3 & \cdots & \sum_{i=1}^{n} t_i^{m+1} \\ \sum_{i=1}^{n} t_i^2 & \sum_{i=1}^{n} t_i^3 & \sum_{i=1}^{n} t_i^4 & \cdots & \sum_{i=1}^{n} t_i^{m+2} \\ \vdots & \vdots & \vdots & & \vdots \\ \sum_{i=1}^{n} t_i^m & \sum_{i=1}^{n} t_i^{m+1} & \sum_{i=1}^{n} t_i^{m+2} & \cdots & \sum_{i=1}^{n} t_i^{2m} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} = \begin{pmatrix} \sum_{i=1}^{n} y_i \\ \sum_{i=1}^{n} y_i t_i \\ \sum_{i=1}^{n} y_i t_i^2 \\ \vdots \\ \sum_{i=1}^{n} y_i t_i^m \end{pmatrix}. \tag{3.44}$$

La matriz de la ecuación (3.44) se denomina *matriz normal*. Tal matriz se puede obtener a partir de una matriz de Vandermonde $A \in \mathbb{R}^{(m+1)\times n}$, usualmente llamada *matriz de diseño*,

$$A = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ t_1 & t_2 & t_3 & \cdots & t_n \\ t_1^2 & t_2^2 & t_3^2 & \cdots & t_n^2 \\ \vdots & \vdots & \vdots & & \vdots \\ t_1^m & t_2^m & t_3^m & \cdots & t_n^m \end{pmatrix}.$$

Es fácil probar que

$$AA^T = \begin{pmatrix} n & \sum_{i=1}^{n} t_i & \sum_{i=1}^{n} t_i^2 & \cdots & \sum_{i=1}^{n} t_i^m \\ \sum_{i=1}^{n} t_i & \sum_{i=1}^{n} t_i^2 & \sum_{i=1}^{n} t_i^3 & \cdots & \sum_{i=1}^{n} t_i^{m+1} \\ \sum_{i=1}^{n} t_i^2 & \sum_{i=1}^{n} t_i^3 & \sum_{i=1}^{n} t_i^4 & \cdots & \sum_{i=1}^{n} t_i^{m+2} \\ \vdots & \vdots & \vdots & & \vdots \\ \sum_{i=1}^{n} t_i^m & \sum_{i=1}^{n} t_i^{m+1} & \sum_{i=1}^{n} t_i^{m+2} & \cdots & \sum_{i=1}^{n} t_i^{2m} \end{pmatrix} \quad \text{y} \quad A\mathbf{y} = \begin{pmatrix} \sum_{i=1}^{n} y_i \\ \sum_{i=1}^{n} y_i t_i \\ \sum_{i=1}^{n} y_i t_i^2 \\ \vdots \\ \sum_{i=1}^{n} y_i t_i^m \end{pmatrix},$$

donde $\mathbf{y} = (y_1, y_2, \ldots, y_n)$ es un vector generalmente llamado *vector de observaciones*. Esto significa que las ecuaciones normales adquieren la forma matricial

$$AA^T\mathbf{x} = A\mathbf{y}, \tag{3.45}$$

además, el problema de minimización (3.41) se puede expresar de la siguiente manera:

$$\min_{\mathbf{x}\in\mathbb{R}^{m+1}} \frac{1}{2}\|A\mathbf{x} - \mathbf{y}\|^2, \tag{3.46}$$

que caracteriza a los problemas de cuadrados mínimos lineales. Existe una amplia literatura para la resolución de este tipo de problemas. En el Capitulo 8 trataremos este tema.

Para resolver el sistema de ecuaciones normales (3.45), se utiliza generalmente la eliminación gaussiana. No obstante, como AA^T es una matriz simétrica semi-definida positiva, el sistema (3.45) también se puede resolver utilizando la factorización de Cholesky. Otra alternativa es usar un método iterativo para resolver (3.45), pero se debe tener presente que AA^T se vuelve progresivamente mal-condicionada con el aumento de m.

Es natural hacer la siguiente pregunta: ¿qué grado del polinomio de cuadrados mínimos se debe usar para ajustar un conjunto de datos? Para responder esta pregunta es necesario calcular el error cometido al ajustar un conjunto de datos. Tal error se define, para una función de ajuste cualquiera, como

$$e_m = \sqrt{\sum_{i=1}^{n} r_i^2} = \sqrt{\sum_{i=1}^{n} (f(\hat{\mathbf{x}}; t_i) - y_i)^2}, \tag{3.47}$$

donde $\hat{\mathbf{x}} = (\hat{x}_0, \hat{x}_1, \ldots, \hat{x}_m)$ es una solución del sistema (3.42).

Cuando la función de ajuste es un polinomio de grado m, el error de ajuste por cuadrados mínimos es:

$$e_m = \sqrt{\sum_{i=1}^{n} (\hat{x}_0 + \hat{x}_1 t_i + \cdots + \hat{x}_m t_i^m - y_i)^2} = \|A\hat{\mathbf{x}} - \mathbf{y}\|. \tag{3.48}$$

De esta forma, el grado del polinomio de cuadrados mínimos se incrementa en tanto haya un decremento significativo del error (3.48).

Recta de Cuadrados Mínimos

En la sección anterior generamos el polinomio de cuadrados mínimos de grado m para ajustar los n puntos (t_i, y_i). Pasemos ahora a considerar el caso cuando el polinomio de cuadrados mínimos es de grado 1, que se usa en muchas aplicaciones de ajuste de curvas. En este caso, las ecuaciones normales (3.43) adquieren la forma:

$$\begin{cases} n x_0 + x_1 \sum_{i=1}^{n} t_i &= \sum_{i=1}^{n} y_i \\ x_0 \sum_{i=1}^{n} t_i + x_1 \sum_{i=1}^{n} t_i^2 &= \sum_{i=1}^{n} y_i t_i \end{cases} \tag{3.49}$$

La solución de este sistema es

$$x_0 = \frac{\sum_{i=1}^{n} y_i \sum_{i=1}^{n} t_i^2 - \sum_{i=1}^{n} t_i \sum_{i=1}^{n} t_i y_i}{n \sum_{i=1}^{n} t_i^2 - \left(\sum_{i=1}^{n} t_i \right)^2}, \tag{3.50}$$

$$x_1 = \frac{n \sum_{i=1}^{n} t_i y_i - \sum_{i=1}^{n} t_i \sum_{i=1}^{n} y_i}{n \sum_{i=1}^{n} t_i^2 - \left(\sum_{i=1}^{n} t_i \right)^2}. \tag{3.51}$$

El polinomio correspondiente

$$f(t) = x_0 + x_1 t$$

es conocido como *recta de cuadrados mínimos* o *línea de regresión*.

3.4.2 Ajuste con Funciones No Lineales

El sistema (3.42) es generalmente no lineal, por ello es necesario usar técnicas de optimización para resolver el problema de cuadrados mínimos no lineales (3.41). En muchos casos, los datos

provenientes de pruebas experimentales no muestran un comportamiento que sea imitable por un polinomio. En tales casos, es preferible ajustar los datos con una función que no sea un polinomio, pero la misma tiene una forma particular. Habitualmente se considera la siguiente lista (no completa) de funciones de ajuste:

- $f(t) = x_0\, e^{x_1 t}$
- $f(t) = x_0\, t x_1$
- $f(t) = x_0 \ln t + x_1$
- $f(t) = \dfrac{x_0}{t} + x_1$
- $f(t) = \dfrac{x_0}{t + x_1}$
- $f(t) = \dfrac{x_0 t}{1 + x_1 t}$

Describiremos un procedimiento, basado en el ajuste polinomial, que permite ajustar una de las funciones no lineales de la lista anterior a un conjunto de datos. Consideremos como ejemplo a la función de ajuste

$$f(t) = x_0\, e^{x_1 t}.$$

Normalmente, el ajuste de esta función nos llevaría a resolver un problema de cuadrados mínimos no lineales, pero podemos proceder de otra manera. Si nosotros tomamos $\ln f(t)$ como función de ajuste en lugar de $f(t)$, el problema se transforma en ajustar una recta de cuadrados mínimos: ajustar la función

$$F(t) = \ln f(t) = \ln x_0 + x_1 t = b_0 + b_1 t$$

a los puntos $(t_i, \ln y_i)$, $i = 1, 2, \ldots, n$, donde $b_0 = \ln x_0$ y $b_1 = x_1$. Ahora, los valores de los parámetros de la función de ajuste original se obtienen (por supuesto, después de hallar los valores de b_0 y b_1) como:

$$x_0 = e^{b_0}, \qquad x_1 = b_1.$$

El proceso descrito se denomina *linealización* de la función de ajuste. En la Tabla 4.3 se aprecian las linealizaciones de las otras funciones de ajuste dadas arriba.

Seguidamente daremos algunos ejemplos de ajustes de curvas.

Ejemplo 3.19. Comencemos ajustando una recta de cuadrados mínimos a la siguiente tabla de datos.

t_i	0.0	1.2	2.0	2.5	3.0
y_i	2.9	3.7	4.1	4.4	5.0

Ajustemos los datos con la recta de cuadrados mínimos

$$f(x) = x_0 + x_1 t,$$

donde los valores de los parámetros x_0 y x_1 están dados respectivamente por las ecuaciones (3.50) y (3.51). Se tiene que

$$\sum_{i=1}^{5} t_i = 8.7, \qquad \sum_{i=1}^{5} t_i^2 = 20.69,$$

$$\sum_{i=1}^{5} y_i = 20.1, \quad \sum_{i=1}^{5} t_i y_i = 30.64$$

Tabla 3.3. Funciones de ajuste linealizadas

$f(t)$	$F(t)$	Datos a ser ajustados por $F(t)$	Restricciones de los datos
$x_0\,e^{x_1 t}$	$\ln f(t) = \ln x_0 + x_1 t$	$(t_i, \ln y_i)$	$y_i > 0$
$x_0\,t^{x_1}$	$\ln f(t) = \ln x_0 + x_1 \ln t$	$(\ln t_i, \ln y_i)$	$t_i > 0, y_i > 0$
$x_0 \ln t + x_1$	$f(t) = x_0 \ln t + x_1$	$(\ln t_i, y_i)$	$t_i > 0$
$\frac{x_0}{t} + x_1$	$f(t) = x_0(1/t) + x_1$	$(1/t_i, y_i)$	$t_i \neq 0$
$\frac{x_0}{t + x_1}$	$1/f(t) = (1/x_0)t + (x_1/x_0)$	$(t_i, 1/y_i)$	$y_i \neq 0$
$\frac{x_0 t}{1 + x_1 t}$	$1/f(t) = (1/x_0)(1/t) + (x_1/x_0)$	$(1/t_i, 1/y_i)$	$t_i \neq 0, y_i \neq 0$

Ahora, sustituyendo convenientemente estos últimos valores en las ecuaciones (3.50) y (3.51), obtenemos:

$$\begin{aligned} x_0 &= \frac{(20.1)(20.69) - (8.7)(30.64)}{5(20.69) - (8.7)^2} = \frac{79.701}{27.76} = 2.8711, \\ x_1 &= \frac{5(30.64) - (8.7)(20.1)}{5(20.69) - (8.7)^2} = \frac{18.33}{27.76} = 0.6603 \end{aligned}$$

De esta forma, la recta de cuadrados mínimos es $f(t) = 2.8711 + 0.6603t$, cuya gráfica se muestra en la Figura 3.18 junto con los puntos (t_i, y_i).

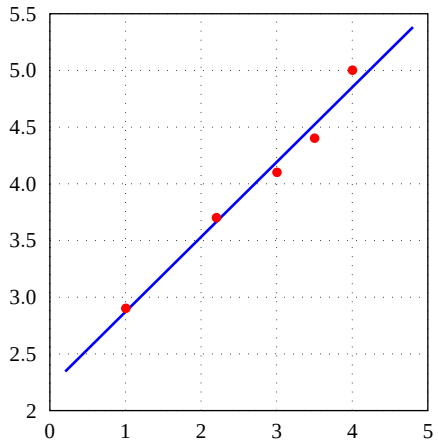


Figura 3.18. Recta de cuadrados mínimos del Ejemplo 3.19

Ahora, calculemos los residuos para hallar el error de cuadrados mínimos:

y_i	2.9	3.7	4.1	4.4	5.0
$f(t_i)$	2.8711	3.6635	4.1917	4.5219	4.8520
$f(t_i) - y_i$	-0.0289	-0.0365	0.0917	0.1219	-0.1480

Por lo tanto, el error de cuadrados mínimos es

$$e_m = \sum_{i=1}^{5} (f(t_i) - y_i)^2 = 0.2176$$

Ejemplo 3.20. Ajustemos la función $f(t) = x_0 e^{x_1 t}$ a la tabla de datos

t_i	1.2	2.8	4.3	5.4	6.8	7.9
y_i	7.5	16.1	38.9	67.0	146.6	266.2

Para el ajuste de los datos con la función $f(t) = x_0 e^{x_1 t}$ usamos su linealización, es decir, ajustamos la función

$$F(t) = \ln f(t) = \ln(x_0) + x_1 t = b_0 + b_1 t$$

a la tabla de datos

t_i	1.2	2.8	4.3	5.4	6.8	7.9
$\ln y_i$	2.0149	2.7788	3.6610	4.2047	4.9877	5.5842

Se tiene que

$$\sum_{i=1}^{6} t_i = 28.4, \qquad \sum_{i=1}^{6} t_i^2 = 165.58,$$

$$\sum_{i=1}^{6} \ln y_i = 23.2314, \quad \sum_{i=1}^{6} t_i \ln y_i = 126.6782$$

Como $F(t) = b_0 + b_1 t$ es la recta de cuadrados mínimos, entonces usando las ecuaciones (3.50) y (3.51), los valores de los parámetros b_0 y b_1 son (saltando algunos detalles aritméticos):

$$b_0 = 1.3321, \quad b_1 = 0.5366$$

Por lo tanto, $x_0 = e^{b_0} = 3.7889$, $x_1 = 0.5366$ y la función de ajuste es $f(t) = 3.7889\, e^{0.5366 t}$. La gráfica de $f(t)$ y los puntos (t_i, y_i) se muestran en la Figura 3.19.

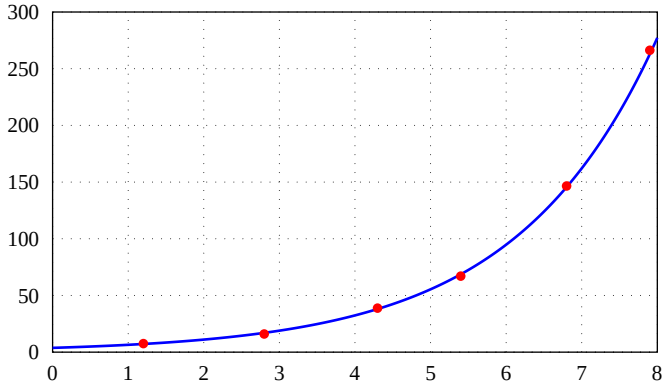


Figura 3.19. Función de ajuste y puntos del Ejemplo 3.20

Seguidamente calculamos el error de cuadrados mínimos:

y_i	7.5000	16.1000	38.9000	67.0000	146.6000	266.2000
$f(t_i)$	7.2138	17.0228	38.0711	68.6977	145.6135	262.7533
$f(t_i)-y_i$	-0.2862	0.9228	-0.8289	1.6977	-0.9865	-3.4467

Por lo tanto, el error de cuadrados mínimos es

$$e_m = \sum_{i=1}^{5} (f(t_i) - y_i)^2 = 4.166$$

Ejemplo 3.21. Mostraremos una función implementada en Python que hemos denominado `polcm`, que permite calcular los coeficientes del polinomio de cuadrados mínimos de grado m que ajusta n puntos (t_i, y_i). Los argumentos de entrada de `polcm` son: un arreglo `t` con elementos t_i, un arreglo `y` con elementos y_i y el grado `m` del polinomio. Los argumentos de salida de `polcm` son: un arreglo `x=(x[0],x[1],...,x[m])` de coeficientes del polinomio

$$p(t) = x[0]t^m + x[1]t^{m-1} + \cdots + x[m-1]t + x[m]$$

y el error de cuadrados mínimos `em`. La función `polcm` la emplearemos para ajustar la tabla de datos

t_i	-5	-4	-3	-2	-1	0	1	2	3	4	5
y_i	0.0385	0.0588	0.1	0.2	0.5	1.0	0.5	0.2	0.1	0.0588	0.0385

con las siguientes funciones:

a) $f_1(t) = a_0 + a_1 t$

b) $f_2(t) = a_0 + a_1 t + a_2 t^2$

c) $f_3(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3$

d) $f_4(t) = \dfrac{1}{a_0 + a_1 t + a_2 t^2}$

El Programa 3.5 muestra la función `polcm`.

Programa 3.5. Función `polcm`

```
def polcm(t,y,m):
    t = np.asarray(t,dtype=float)
    y = np.asarray(y,dtype=float)
    n = len(t)
    A = np.zeros((m+1,n))
    A[0,:] = np.ones(n)
    for k in  range(1,m+1):
        A[k,:] = t**k
    x = np.linalg.solve(A.dot(A.T),A.dot(y))
    x = x[range(m,-1,-1)]
    em = np.sqrt(np.sum((np.polyval(x, t)- y)**2))
    return x,em
```

Seguidamente empleamos la función `policm` para ajustar los datos. Como las funciones $f_1(t)$, $f_2(t)$ y $f_3(t)$ son polinomios, utilizamos directamente la función `policm`. Para el ajuste de los datos con la función $f_4(t)$ usamos su linealización, es decir, ajustamos la función

$$F_4(t) = \frac{1}{f_4(t)} = x_0 + x_1 t + x_2 t^2.$$

a los datos $(t_i, 1/y_i)$, $i = 1,2,\ldots,11$. Con las siguientes instrucciones se generan los coeficientes de los polinomios $f_1(t)$, $f_2(t)$, $f_3(t)$ y $F_4(t)$, y los errores de cuadrados mínimos para el ajuste con $f_1(t)$, $f_2(t)$ y $f_3(t)$.

```
>>> t = np.array([-5,-4,-3,-2,-1,0,1,2,3,4,5])
>>> y=np.array([0.0385,0.0588,0.1,0.2,0.5,1,0.5,0.2,0.1,0.0588,0.0385])
>>> x1,em1 = polcm(t,y,1)
>>> print('x1=',x1,'\nem1={0:.3e}'.format(em1))
x1= [0.          0.25405455]
em1=9.486e-01
>>> x2,em2 = polcm(t,y,2)
>>> print('x2=',x2,',','\nem2={0:.3e}'.format(em2))
x2= [-0.02300629   0.           0.48411748] ,
em2=6.677e-01
>>> x3,em3 = polcm(t,y,3)
>>> print('x3=',x3,'\nem3={0:.3e}'.format(em3))
x3= [ 6.37997238e-20 -2.30062937e-02 -8.83311669e-19  4.84117483e-01]
em3=6.677e-01
>>> x4,em4 = polcm(t,1/y,2)
>>> print('x4=',x4)
x4= [0.99918696  0.          1.00464471]
```

Además con la siguiente instrucción calculamos el error al ajustar $f_4(t)$ a los datos dados:

```
>>> em4 = np.sqrt(sum((1/np.polyval(x4,t)-y)**2))
>>> print('em4={0:.3e}'.format(em4))
em4=4.818e-03
```

De esta forma, las funciones de ajuste y sus respectivos errores de ajuste son:

$$\begin{aligned}
f_1(t) &= 0.254, & e_{m_1} &= 9.486\times 10^{-1},\\
f_2(t) &= 0.484 - 2.301\times 10^{-2}t^2, & e_{m_2} &= 6.677\times 10^{-1},\\
f_3(t) &= 0.484 - 8.833\times 10^{-19}t - 2.301\times 10^{-2}t^2 - 6.37\times 10^{-20}t^3, & e_{m_3} &= 6.677\times 10^{-1},\\
f_4(t) &= \frac{1}{1.0046 + 0.99919t^2}, & e_{m_4} &= 4.818\times 10^{-3}
\end{aligned}$$

Las gráficas de $f_1(t)$, $f_2(t)$, $f_3(t)$ y $f_4(t)$, y los puntos (t_i, y_i) se muestran en la Figura 3.20. Todas las gráficas se generaron con las siguientes instrucciones.

```
import matplotlib.pyplot as plt
plt.figure(1)
tt = np.linspace(-6,6,500)
plt.plot(tt,np.polyval(x1,tt),label='$f_{1}(t)$')
plt.plot(tt,np.polyval(x2,tt),label='$f_{2}(t)$')
plt.plot(tt,np.polyval(x3,tt),label='$f_{3}(t)$')
plt.plot(tt,1/np.polyval(x4,tt),label='$f_{4}(t)$')
plt.plot(t,y,'o',label='Puntos')
plt.grid()
plt.legend(loc='best')
plt.show()
```

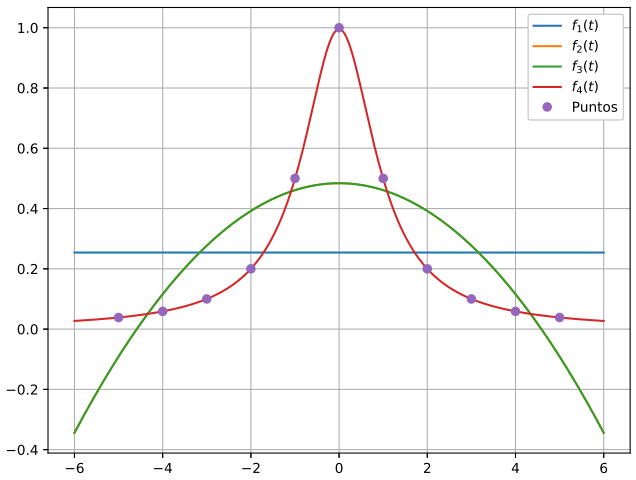


Figura 3.20. Funciones de ajuste del Ejemplo 3.21

Un análisis visual de las gráficas conjuntamente con el valor de los errores de ajuste, permite asegurar que la función $f_4(t)$ es la que mejor ajusta los datos.

La función `polyfit(t,y,m)`, además de proveer los coeficientes del polinomio interpolante $p_n(t)$ cuando $m = n$, ésta también provee los coeficientes a_k del polinomio de cuadrados mínimos de grado m en su forma canónica,

$$p(t) = a_1 t^n + a_2 t^{n-1} + \cdots + a_n t + a_{m+1}.$$

Veamos un ejemplo.

Ejemplo 3.22. Empleemos la función `polyfit(t,y,m)` para ajustar la tabla de datos del Ejemplo 3.10,

t_i	-5	-4	-3	-2	-1	0	1	2	3	4	5
y_i	0.0385	0.0588	0.1	0.2	0.5	1.0	0.5	0.2	0.1	0.0588	0.0385

con los siguientes polinomios:

a) $f_1(t) = a_1 t + a_2$

b) $f_2(t) = a_1 t^2 + a_2 t + a_3$

c) $f_3(t) = a_1 t^3 + a_2 t^2 + a_3 t + a_4$

Utilicemos `polyfit(t,y,m)` convenientemente para cada m:

```
>>> t = np.array([-5,-4,-3,-2,-1,0,1,2,3,4,5])
>>> y = np.array([0.0385,0.0588,0.1,0.2,0.5,1,0.5,0.2,0.1,0.0588,0.0385])
>>> a1 = np.polyfit(t,y,1)
>>> print('a1=',a1)
a1= [-6.61597575e-19  2.54054545e-01]
>>> a2 = np.polyfit(t,y,2)
>>> print('a2=',a2)
```

```
a2= [-0.02300629   0.          0.48411748]
>>> a3 = np.polyfit(t,y,3)
>>> print('a3=',a3)
a3= [ 1.01065491e-18 -2.30062937e-02 -2.11711224e-17  4.84117483e-01]
```

De esta forma, los polinomios de cuadrados mínimos hallados son:

$$f_1(t) = 0.254 - 6.616 \times 10^{-19} t$$
$$f_2(t) = 0.484 - 2.301 \times 10^{-2} t^2$$
$$f_3(t) = 0.484 - 2.117 \times 10^{-17} t - 2.301 \times 10^{-2} t^2 + 1.011 \times 10^{-18} t^3$$

3.5 Aplicaciones

A continuación describimos dos aplicaciones donde se emplean la interpolación y el ajuste curvas. En la primera aplicación se usa la interpolación a trozos para aproximar datos obtenidos en mediciones experimentales. La segunda aplicación es del área del procesamiento de señales; particularmente, se describe el empleo de la FFT para filtrar una señal digital.

3.5.1 Ajuste de Datos de Medición

Usaremos las técnicas de interpolación polinomial a trozos y el ajuste de curvas para aproximar la intensidad del campo eléctrico E y la densidad del flujo magnético B de una subestación de $500\,kV$ (ver Figura 3.21). Los puntos de medición (x_i, y_j) se distribuyen uniformemente en el dominio de 125m $\times$ 50m, donde $x_i = ih$ para $i = 0, 1, \ldots, 25$ e $y_j = jh$ para $j = 0, 1, \ldots, 10$, donde $h = 5$m. En total, tenemos $26 \times 11 = 286$ puntos de medición, cuya distribución se muestra en el diagrama esquemático de la Figura 3.21. Los datos de medición de E y B se muestran en las Tablas 3.4 y 3.5.

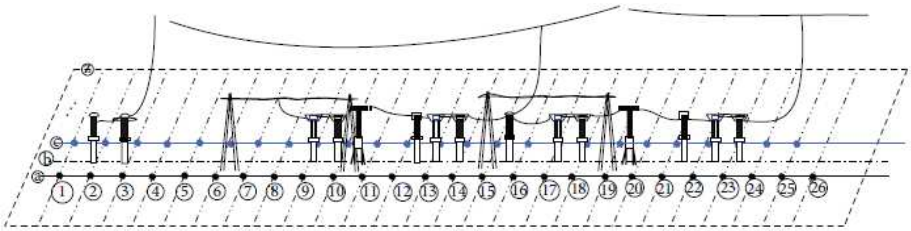


Figura 3.21. Distribución de los puntos de medición en una subestación de $500\,kV$. Fuente: [33, p. 5].

Tabla 3.4. Datos de medición de la intensidad del campo eléctrico $E(\mathrm{kV/m})$.

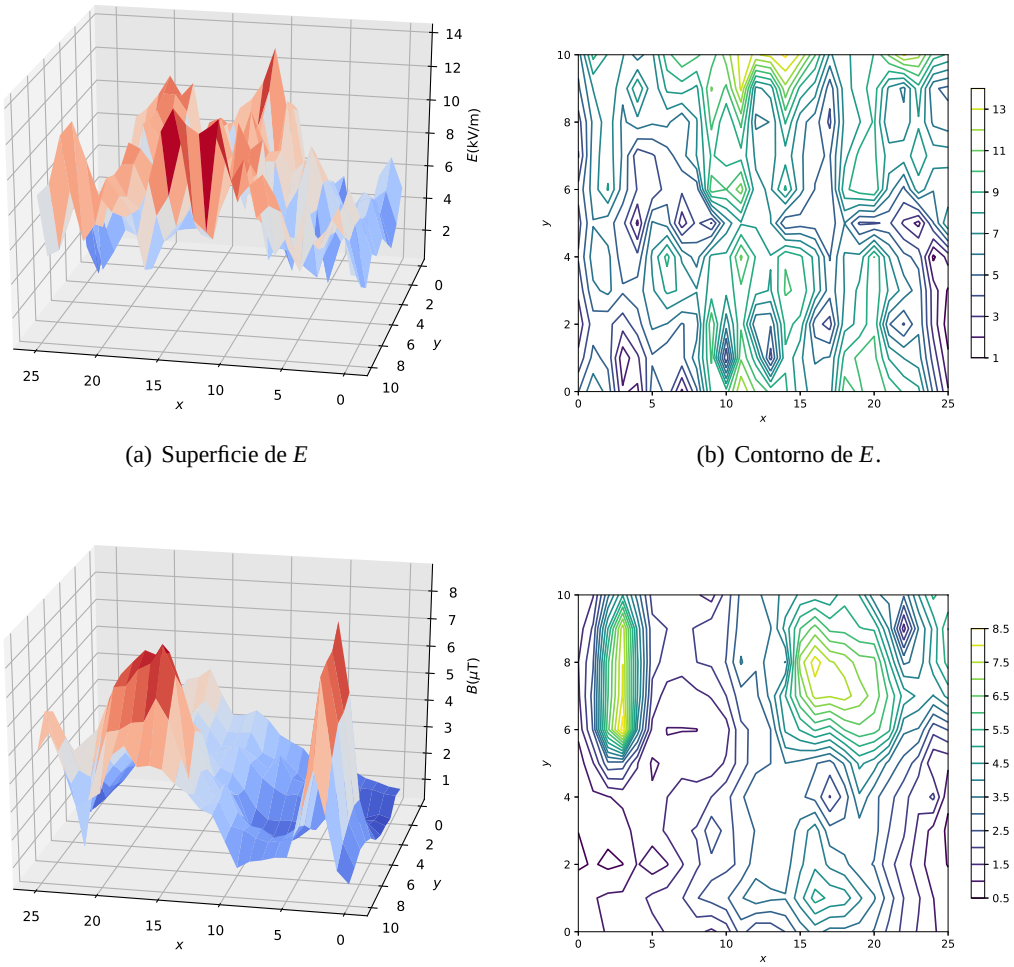
	$y=0$	$y=1$	$y=2$	$y=3$	$y=4$	$y=5$	$y=6$	$y=7$	$y=8$	$y=9$	$y=10$
$x=0$	4.949	3.277	2.347	3.23	4.112	3.615	4.316	2.903	3.569	4.801	5.473
$x=1$	6.797	5.301	4.485	5.788	6.485	5.945	6.467	6.548	6.227	7.038	9.313
$x=2$	5.612	4.686	4.24	6.243	6.281	5.435	8.081	7.618	7.579	7.984	7.909
$x=3$	2.152	1.307	3.431	5.155	4.385	4.334	5.504	5.864	5.78	6.211	7.121
$x=4$	1.751	2.802	4.571	5.501	4.268	1.798	3.929	3.343	6.646	8.955	5.446
$x=5$	5.462	4.854	6.211	6.829	7.205	6.144	3.186	3.889	6.135	6.167	6.787
$x=6$	4.791	4.184	5.969	8.217	9.332	5.625	4.492	4.57	6.242	8.215	9.666
$x=7$	1.804	4.701	4.918	7.875	6.825	2.389	4.218	5.613	5.093	8.519	10.84
$x=8$	4.829	6.525	5.528	7.087	5.527	4.284	4.377	7.031	6.907	7.782	10.2
$x=9$	9.263	10.15	10.81	8.395	8.305	2.93	10.63	9.105	10.26	11.07	9.338
$x=10$	9.487	3.081	6.437	8.27	8.213	6.467	9.992	8.564	8.012	9.816	11.5
$x=11$	12.93	10.87	9.811	9.748	11.15	7.88	11.49	8.632	10.26	13.2	13.38
$x=12$	10.87	8.478	5.087	8.386	8.919	7.857	7.27	6.813	5.774	6.735	12.87
$x=13$	9.77	3.571	5.102	7.923	8.048	7.049	6.839	6.73	6.028	7.398	13.34
$x=14$	9.21	8.824	9.278	10.13	9.922	4.093	9.181	7.557	9.081	11.72	14.2
$x=15$	7.484	7.209	8.727	9.804	9.198	4.547	6.29	6.197	6.387	8.708	12.87
$x=16$	6.486	5.893	4.438	7.916	7.709	4.464	4.973	5.132	5.46	7.527	10.07
$x=17$	6.453	6.077	3.573	5.424	4.939	4.408	4.336	4.926	3.78	4.337	6.833
$x=18$	10.55	9.301	7.643	7.029	7.144	5.988	8.715	7.662	7.267	7.548	7.905
$x=19$	9.761	10.3	8.221	8.272	7.264	3.85	9.635	9.294	7.918	8.52	8.154
$x=20$	10.91	10.32	9.822	9.683	10.01	3.887	9.962	9.198	9.145	9.494	9.047
$x=21$	9.75	7.933	5.084	7.867	7.022	4.177	6.486	7.573	6.068	6.796	11.13
$x=22$	8.859	6.973	3.898	6.596	5.284	2.828	4.747	7.551	6.353	5.748	13.15
$x=23$	8.263	6.456	5.796	5.545	4.694	1.66	7.632	8.527	7.874	8.625	12.3
$x=24$	4.775	3.587	2.566	2.526	0.781	4.971	6.054	7.571	5.791	5.141	9.547
$x=25$	2.092	1.184	0.45	0.863	2.395	7.947	6.002	7.537	5.431	4.751	8.835

Tabla 3.5. Datos de medición de la densidad del flujo magnético $B(\mu\mathrm{T})$.

	$y=0$	$y=1$	$y=2$	$y=3$	$y=4$	$y=5$	$y=6$	$y=7$	$y=8$	$y=9$	$y=10$
$x=0$	0.829	0.923	0.413	0.505	0.52	0.715	1.152	1.399	1.492	1.219	1.008
$x=1$	0.89	0.962	0.548	0.684	0.901	1.463	2.546	2.69	2.454	2.086	1.457
$x=2$	1.034	1.103	0.396	0.628	0.97	2.862	6.336	5.973	5.905	4.995	3.356
$x=3$	1.149	1.122	0.492	0.734	1.138	2.763	8.782	8.061	7.996	7.806	4.956
$x=4$	1.375	1.348	1.033	1.229	1.399	1.516	3.847	4.5	4.576	4.322	3.268
$x=5$	1.365	1.611	0.743	1.219	1.064	0.932	1.344	1.84	2.158	2.151	1.799
$x=6$	1.539	1.746	0.967	1.599	1.493	1.083	0.984	1.254	1.844	2.0	1.76
$x=7$	1.632	1.961	1.459	1.931	1.842	1.219	0.987	1.217	1.704	1.907	1.635
$x=8$	1.631	2.477	1.904	2.035	1.759	1.177	0.991	1.378	1.998	2.032	1.555
$x=9$	1.392	2.682	2.332	2.864	2.003	1.19	1.159	1.531	2.072	2.297	1.294
$x=10$	1.71	2.925	2.377	2.313	2.352	1.577	1.272	1.998	2.523	2.389	2.227
$x=11$	2.746	3.454	2.589	2.705	2.888	2.262	2.4	2.799	3.551	3.257	3.129
$x=12$	2.056	3.789	3.287	3.029	3.467	2.798	2.608	2.881	3.044	3.107	3.314
$x=13$	2.127	3.593	3.258	3.086	3.303	2.997	2.995	3.642	3.903	3.466	3.828
$x=14$	2.514	3.533	3.284	3.071	3.153	3.494	4.577	4.997	3.475	4.692	4.131
$x=15$	3.19	4.295	3.909	3.718	3.071	3.361	5.036	6.466	7.24	5.992	4.592
$x=16$	3.693	5.221	4.375	4.268	2.855	3.039	5.573	7.664	8.264	6.707	4.711
$x=17$	3.333	4.89	3.882	3.182	1.964	2.794	5.254	7.546	7.562	5.866	4.629
$x=18$	3.349	4.779	3.626	3.182	2.694	3.568	6.023	7.699	7.339	5.809	5.347
$x=19$	3.289	4.086	3.167	3.24	3.93	4.612	6.361	7.266	6.739	5.526	4.992
$x=20$	2.96	3.559	2.971	3.187	3.144	4.245	6.199	6.804	6.671	5.431	3.621
$x=21$	2.372	2.747	2.138	2.503	2.68	3.321	5.616	5.917	5.081	3.95	3.338
$x=22$	1.853	1.998	1.771	2.057	2.189	2.826	3.6	4.814	3.414	1.261	3.084
$x=23$	1.443	1.555	1.182	1.38	1.621	2.236	3.462	4.397	3.644	3.519	4.484
$x=24$	1.025	1.006	0.74	0.806	2.119	1.026	2.169	3.976	4.804	4.624	5.628
$x=25$	0.903	0.779	0.475	0.405	0.588	1.1	2.532	4.665	5.693	6.016	4.971

Inicialmente mostramos en la Figura 3.22, las superficies y los contornos del campo eléctrico y el flujo magnético, obtenidos de los datos de medición. En el Programa 3.6 se muestra el Script utilizado para construir tales figuras. Aquí se requieren los archivos de texto `campoelectrico.txt`

y `flujomagnetico.txt`, que contienen los datos de las Tablas 3.4 y 3.5, respectivamente; además, están disponibles en: http://neutron.ing.ucv.ve/cruzw/index.html



(a) Superficie de E

(b) Contorno de E.

(c) Superficie de B.

(d) Contorno de B.

Figura 3.22. Superficies y contornos de los datos de medición para E y B.

Programa 3.6. Script para graficar E y B

```
import numpy as np
from mpl_toolkits.mplot3d import axes3d, Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm
from scipy import interpolate

def plot_data(xx,yy,zz,nfig,zetiq):
    fig = plt.figure(nfig)
    ax = Axes3D(fig)
    #Gráfica de la superficie
    surf = ax.plot_surface(xx, yy, zz, cmap=cm.coolwarm,linewidth=0,
                           antialiased=False)
    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
    ax.set_zlabel(zetiq)
    ax.view_init(24,102)
    #Gráfica de las curvas de nivel
    plt.figure(nfig+1)
    CS = plt.contour(xx, yy, zz, 16)
    CB = plt.colorbar(CS, shrink=0.8, extend='both')
    plt.xlabel('$x$')
    plt.ylabel('$y$')

campo = np.loadtxt('campoelectrico.txt', delimiter=" ")
flujo = np.loadtxt('flujomagnetico.txt', delimiter=" ")
x = np.linspace(0,25,26)
y = np.linspace(0,10,11)
xx, yy = np.meshgrid(x,y)

#Gráficas del campo eléctrico
plot_data(xx,yy,campo.T,1,'$E(\mathrm{kV/m})$')

#Gráficas del flujo magnético
plot_data(xx,yy,flujo.T,3,'$B(\mu\mathrm{T})$')
```

De estas figuras se observa lo siguiente: (i) La intensidad del campo eléctrico presenta fuertes variaciones en comparación con la densidad del flujo magnético. La razón de ello es que la intensidad del campo eléctrico se ve afectada de forma significativa por el equipo de la subestación, mientras que la afectación de la densidad del flujo magnético es menor ([33]); y (ii) las superficies tienen cambios pronunciados (con muchos picos) y los contornos correspondientes muestran poca suavidad. Esto representa un inconveniente, ya que la distribución real del campo electromagnético es en realidad continua y suave.

La función `interp2d(x, y, z)` del módulo `interpolate` de `Scipy`, emplea splines para aproximar una función $z = f(x,y)$ en una rejilla de puntos $(x_0,y_0),(x_1,y_2),\ldots(x_n,y_m)$. Utilicemos tal función para aproximar la intensidad del campo eléctrico $E(x,y)$ y la densidad del flujo magnético $B(x,y)$, en los puntos (x_i,y_j) del dominio 125m $\times$ 50m, donde $x_i = ih$ para $i = 0,1,\ldots,25$ e $y_j = jh$ para $j = 0,1,\ldots,10$, con $h = 5$m. Con las siguientes instrucciones generamos aproximaciones de $E(x,y)$ y $B(x,y)$, usando splines cúbicos.

```
>>> x = np.linspace(0,25,26)
>>> y = np.linspace(0,10,11)
>>> E = interpolate.interp2d(x, y, campo.T, kind='cubic')
>>> B = interpolate.interp2d(x, y, flujo.T, kind='cubic')
```

Los comandos

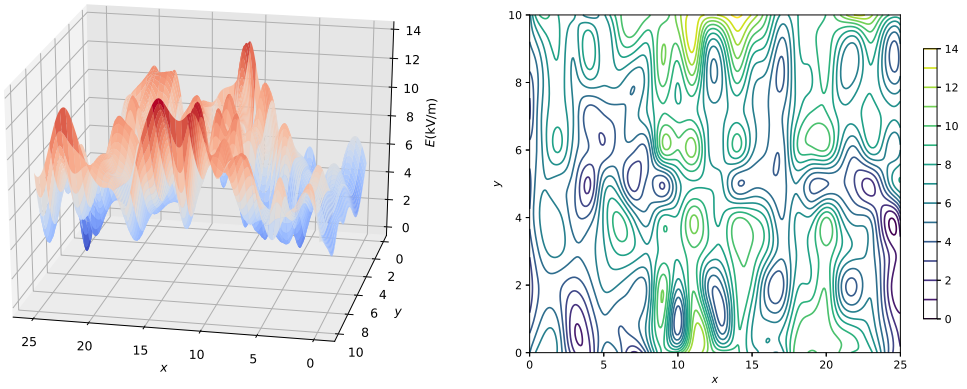
```
E = interpolate.interp2d(x, y, campo.T, kind='cubic')
```

y

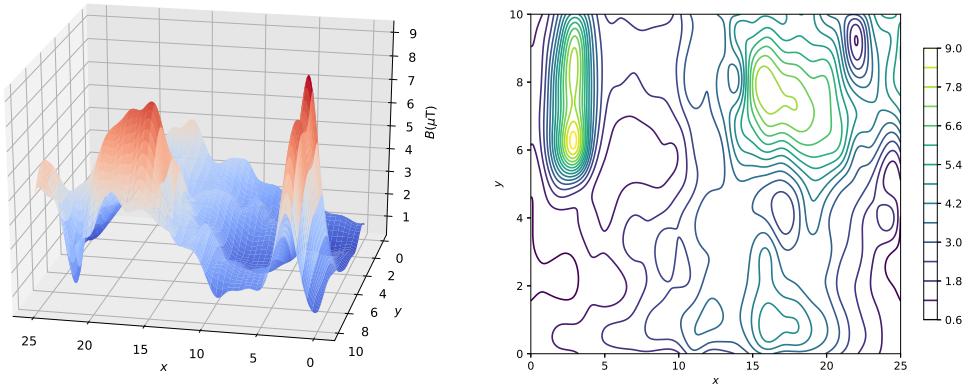
```
B = interpolate.interp2d(x, y, flujo.T, kind='cubic'),
```

arrojan como salida una función que aproxima a $E(x,y)$ y $B(x,y)$, respectivamente; por ejemplo, podemos calcular $E(2.5,15)$ y $B(2.5,15)$,

```
>>> E(2.5,15)
array([7.53139319])
>>> B(2.5,15)
array([4.45758641])
```



(a) Superficie de E (b) Contorno de E.



(c) Superficie de B. (d) Contorno de B.

Figura 3.23. Aproximaciones de las superficies y los contornos de E y B, usando splines cúbicos.

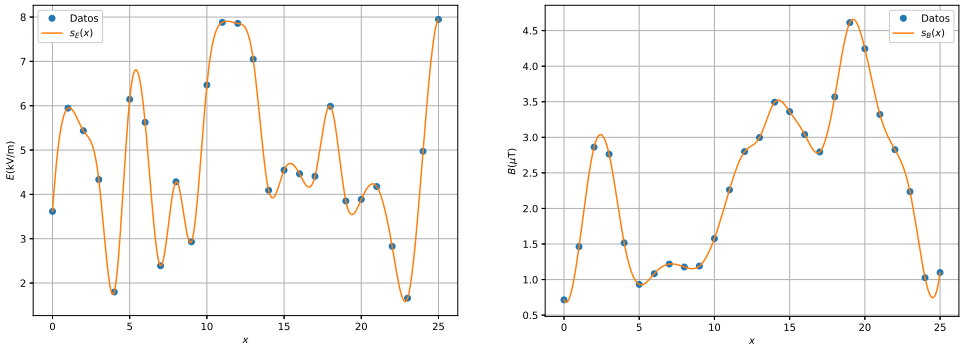
También podemos obtener las superficies y los contornos de $E(x,y)$ y $B(x,y)$, que se muestran en la Figura 3.23.

```
>>> x_v = np.linspace(0,25,400)
>>> y_v = np.linspace(0,10,400)
>>> xx_v, yy_v = np.meshgrid(x_v, y_v)
>>> zz_v = E(x_v, y_v)
```

```
>>> plot_data(xx_v,yy_v,zz_v,1,'$E(\mathrm{kV/m})$')
>>> zz_v = B(x_v, y_v)
>>> plot_data(xx_v,yy_v,zz_v,3,'$B(\mu\mathrm{T})$')
```

Aquí se utiliza la función `plot_data` del Programa 3.6. Note que las superficies y los contornos de la Figura 3.23 son suaves.

Las aproximaciones de $E(x,y)$ y $B(x,y)$ encontradas con `interpolate.interp2d` permiten ajustar la curva de la intensidad del campo eléctrico o de la intensidad del flujo magnético, para los datos de medición de una línea específica. Por ejemplo, si se selecciona la línea $y = 25$m, las funciones $s_E(x) = E(x,25)$ y $s_B(x) = B(x,25)$ son aproximaciones de la intensidad del campo eléctrico y de la densidad del flujo magnético, en la línea que corresponde a la sexta columna en las Tablas 3.4 y 3.5, respectivamente. En la Figura 3.24 se muestran las gráficas de $s_E(x)$ y $s_B(x)$. Note que ambas funciones son splines cúbicos.



(a) Gráfica de $s_E(x)$. (b) Gráfica de $s_B(x)$.

Figura 3.24. Aproximaciones de $E(x,y)$ y $B(x,y)$ en la línea $y = 25$m.

3.6 Ejercicios

1. Encuentre los polinomios de menor grado que interpolan a los siguientes conjuntos de datos:

a)

t_i	7	1	2
y_i	146	2	1

b)

t_i	3	7	1	2
y_i	10	146	2	1

c)

t_i	3	7	1	2
y_i	12	146	2	1

d)

t_i	1.5	2.7	3.1	-2.1	-6.6	11.0
y_i	0.0	0.0	0.0	1.0	0.0	0.0

2. Determine el polinomio interpolante en las formas de Lagrange y Newton para los siguientes datos:

t_i	2	0	3
y_i	11	7	28

3. Encuentre el polinomio interpolante en las formas de Lagrange y Newton para los siguientes datos:

t_i	-2	0	1
y_i	0	1	-1

Escriba ambos polinomios en la forma estándar con el fin de verificar que son idénticos como funciones.

4. Si interpolamos la función $f(t) = e^{t-1}$ con un polinomio $p_{12}(t)$ de grado 12, utilizando 13 nodos en $[-1,1]$, ¿cuál es una buena cota superior para $|f(t) - p_{12}(t)|$ en $[-1,1]$?

5. Encuentre un polinomio de grado 4 que ajuste los siguientes valores:

t_i	0	1	2
$f(t_i)$	2	-4	-44
$f'(t_i)$	-9	4	

6. Encuentre un polinomio de grado 5 que ajuste los valores dados en el Problema 5 y que, además, satisfaga $p(3) = 2$.

7. ¿Qué condición se debe imponer sobre los nodos t_0 y t_1 si se quiere que el problema de interpolación

$$p(t_i) = c_{i0} \qquad p''(t_i) = c_{i2} \qquad (i = 0,1)$$

se resuelva con un polinomio cúbico (para c_{ij} arbitrario)?

8. Un experimento ha producido los siguientes datos:

t_i	0.0	0.5	1.0	6.0	7.0	9.0
y_i	0.0	1.6	2.0	2.0	1.5	0.0

Deseamos interpolar los datos con una curva suave con la esperanza de obtener valores razonables de *y* para valores de *t* entre los puntos en los que se tomaron las mediciones.

a) Usando cualquier método de su preferencia, determine el polinomio de grado cinco que interpola los datos dados y haga su gráfica sobre el rango $0 \le t \le 9$.

b) De manera similar, determine un spline cúbico natural que interpola los datos dados y haga su gráfica en el mismo rango.

c) ¿Qué interpolante parece dar valores más razonables entre los puntos de datos dados? ¿Puede explicar por qué cada curva se comporta como lo hace?

d) ¿Podría un spline lineal ser una mejor opción para estos datos en particular? ¿Por qué?

9. La función *gamma* está definida por

$$\Gamma(t) = \int_0^\infty x^{t-1} e^{-x}\, dx, \qquad t > 0.$$

Para un argumento entero *n*, la función gamma tiene el valor

$$\Gamma(n) = (n-1)!,$$

así que interpolando la tabla de dados

t_i	1	2	3	4	5
y_i	1	1	2	6	24

debe producir una aproximación a la función gamma en un rango determinado.

a) Calcule el polinomio de grado cuatro que interpola estos cinco puntos de datos. Grafique el polinomio resultante, así como los valores correspondientes de la función `gamma` de la librería `scipy.special`, sobre el dominio $[1,5]$.

b) Use un spline cúbico natural para interpolar los mismos datos y nuevamente trace la curva resultante junto con la función `gamma`.

c) ¿Cuál de los dos interpolantes es más preciso en la mayor parte del dominio?

d) ¿Cuál de los dos interpolantes es más preciso entre 1 y 2?

10. Para las siguientes tablas de datos construya la gráfica del polinomio trigonométrico que las interpola, además, indique su grado.

a)

x_i	1.0	1.3	1.6	1.9	2.2	2.5
y_i	0.9	1	-1	0.8	0.9	1

b)

x_i	2.0	2.3	2.6	2.9	3.2	3.5
y_i	1.40	1.06	0.77	0.15	-0.62	0.31

c)

x_i	0.7	1.0	1.3	1.6	1.9	2.2	2.5	2.8
y_i	-0.20	-0.54	-0.12	0.38	1.02	0.92	0.56	0.19

d)

x_i	1.0	1.5	2.0	2.5	3.0	3.5	4.0	4.5
y_i	1.00	0.82	0.13	0.74	0.95	1.16	0.85	-0.25

e)

x_i	y_i	x_i	y_i
1.0	5.024	1.8	0.543
1.1	5.536	1.9	0.510
1.2	3.023	1.0	0.702
1.3	1.505	2.1	0.189
1.4	1.559	2.2	0.176
1.5	1.021	2.3	-0.096
1.6	0.965	2.4	-1.112
1.7	0.998	2.5	0.465

11. Para las siguientes tablas de datos ajuste una recta de cuadrados mínimos $f(t) = x_0 + x_1 t$ y reporte el error cometido.

a)

t_i	-2	-1	0	1	2
y_i	1.2	1.9	2.6	3.3	4.0

b)

t_i	-4	-2	0	2	4
y_i	0.44	3.34	6.24	9.14	12.04

c)

t_i	-4	-1	0	2	3
y_i	-3.0	-0.9	-0.2	1.2	1.9

d)

t_i	-8	-2	0	4	6
y_i	7.32	3.81	2.64	0.30	-0.87

12. Para las siguientes tablas de datos ajuste una parábola de cuadrados mínimos $f(t) = x_0 + x_1 t + x_2 t^2$ y reporte el error cometido.

a)

t_i	-2	-1	0	1	2
y_i	-5.8	1.1	3.8	3.3	1.5

b)

t_i	-2	-1	0	1	2
y_i	2.8	2.1	3.25	6.0	11.5

13. Para la siguiente tabla de datos,

t_i	1	2	3	4	5
y_i	0.6	1.9	4.3	7.6	12.6

determine la curva de cada familia que mejor se les ajusta en el sentido de los cuadrados mínimos.

a) $f(t) = x_0\, e^{x_1 t}$

b) $f(t) = x_0\, t^{x_1}$

14. Para la siguiente tabla de datos,

t_i	-1	0	1	2	3
y_i	6.62	3.94	2.17	1.35	0.89

determine la curva de cada familia que mejor se les ajusta en el sentido de los cuadrados mínimos.

a) $f(t) = a_0\, e^{a_1 t}$

b) $f(t) = 1/(a_1 t + a_0)$

c) $f(t) = (a_0 + a_1 t)^{-2}$

15. Para el conjunto de datos que se muestra al final del enunciado, determine la curva de cada familia que mejor se les ajusta en el sentido de los cuadrados mínimos.

a) $f(t) = \dfrac{x_1}{t} + x_0$

b) $f(t) = \dfrac{1}{x_1 t + x_0}$

c) $f(t) = x_1 \ln(t) + x_0$

d) $f(t) = (x_0 + x_1 t)^{-2}$

e) $f(t) = \dfrac{x_0}{t + x_1}$

f) $f(t) = \dfrac{x}{x_0 + x_1 t}$

g) $f(t) = x_0\, t^{x_1}$

h) $f(t) = x_0\, e^{-x_1 t}$

t_i	y_i	t_i	y_i
0.1000	0.3755	1.0744	11.3236
0.1487	0.5467	1.1231	9.8766
0.1974	0.7514	1.1718	8.5651
0.2462	1.0011	1.2205	7.4238
0.2949	1.3094	1.2692	6.4507
0.3436	1.6934	1.3179	5.6285
0.3923	2.1754	1.3667	4.9351
0.4410	2.7835	1.4154	4.3492
0.4897	3.5532	1.4641	3.8519
0.5385	4.5257	1.5128	3.4275
0.5872	5.7439	1.5615	3.0630
0.6359	7.2385	1.6103	2.7481
0.6846	9.0005	1.6590	2.4743
0.7333	10.9355	1.7077	2.2348
0.7821	12.8203	1.7564	2.0240
0.8308	14.3141	1.8051	1.8376
0.8795	15.0806	1.8538	1.6720
0.9282	14.9740	1.9026	1.5240
0.9769	14.1183	1.9513	1.3913
1.0256	12.8042	2.0000	1.2719

4 | Ecuaciones No Lineales

Como mencionamos en el Capítulo 2, en la naturaleza algunas relaciones son lineales, pero muchas otras son intrínsecamente no lineales. Por ejemplo, la fuerza sobre un objeto en movimiento debido a la resistencia del aire es proporcional al cuadrado de su velocidad; o la relación entre la masa y la energía descrita en la famosa ecuación de Albert Einstein, $E = mc^2$, que establece que la energía es igual a la masa por la velocidad de la luz al cuadrado. Otra ecuación no lineal que incluye más características físicas es la ecuación de estado de van der Waals,

$$\left(p + \frac{\alpha}{v^2}\right)(v - \beta) = RT,$$

donde p es la presión relativa, v es el volumen específico, T es la temperatura, R es una constante universal y α y β son constantes que dependen del gas en particular. Este es un ejemplo típico en que la relación se vuelve más no lineal a medida que se consideran más características físicas.

Haciendo analogía con los sistemas de ecuaciones lineales $A\mathbf{x} = \mathbf{b}$, vistos en el Capí-tulo 2, la forma más natural de denotar una ecuación no lineal sería $\mathbf{f}(\mathbf{x}) = \mathbf{y}$, pero es más habitual restar en ambos lados de la ecuación el vector constante $\mathbf{y}$ e incorporarlo a la función vectorial $\mathbf{f}(\mathbf{x})$, obteniendo como resultado que la ecuación a resolver es $\mathbf{f}(\mathbf{x}) = 0$.

En general, una ecuación no lineal

$$\mathbf{f}(\mathbf{x}) = 0$$

donde $\mathbf{f} : \mathbb{R}^m \to \mathbb{R}^n$, se denomina *sistema de ecuaciones no lineales* de m ecuaciones y n incógnitas. En este capítulo estudiaremos el caso en que el sistema tiene el mismo número de ecuaciones e incógnitas, es decir, $m = n$. Comenzaremos el capítulo describiendo los métodos numéricos de mayor uso para resolver una ecuación no lineal ($m = n = 1$) o, como usualmente se le conoce, *hallar las raíces o ceros de una función*. En la segunda parte del capítulo, describiremos los métodos numéricos clásicos para resolver sistemas de ecuaciones no lineales de dimensión $n \geq 2$.

A lo largo del capítulo presentaremos teoremas cuya demostración se pueden hallar en libros especializados de análisis numérico, en particular, recomendamos los siguientes libros [21], [16], [6], [14].

4.1 Raíces o Ceros de una Función Escalar

Un problema común que se encuentra en Ingeniería es el siguiente: dada una función $f : \mathbb{R} \to \mathbb{R}$, determinar los valores de x de manera que $f(x) = 0$. Las soluciones (valores de x) son conocidos como *raíces* de la ecuación $f(x) = 0$ o *ceros* de la función $f(x)$.

Las raíces de una ecuación pueden ser reales o complejas, por ejemplo, las soluciones de la ecuación polinómica

$$a_0 + a_1 x + \cdots + a_m x^m = 0$$

donde las raíces complejas son importantes (como en el análisis de vibraciones amortiguadas, por ejemplo). En esta sección, nos concentraremos en encontrar raíces reales.

En general, una ecuación puede tener una raíz, infinitas o pudiera no tener raíces. Por ejemplo, la ecuación

$$\operatorname{sen} x - x = 0$$

que es un caso particular de la *ecuación de Kepler*, tiene una sola raíz, a saber $x = 0$, mientras una ecuación que aparece en la teoría de la difracción de la luz,

$$\tan x - x = 0,$$

tiene un número infinito de raíces ($x = 0, \pm 4.493, \pm 7.725, \ldots$). En cambio, la ecuación $e^x = 0$ no tiene raíces.

La mayoría de los procedimientos para encontrar raíces son métodos iterativos que requieren un punto inicial, es decir, una aproximación de la raíz. Esta aproximación inicial puede ser crucial; un valor de partida malo puede hacer que el método no converja o puede que converja a una "mala" raíz (una raíz diferente a la buscada o que no se corresponda con las propiedades físicas del modelo). No hay una receta universal para obtener una aproximación inicial. Si la ecuación está asociada a un problema físico, entonces el contexto del problema (visión física) podría sugerir la ubicación aproximada de la raíz. De lo contrario, debe hacerse una búsqueda gráfica dibujando la función.

4.1.1 Método de Bisección

Sea $f(x)$ una función continua en el intervalo $[a,b]$. El primer método que estudiaremos para hallar una raíz de la ecuación $f(x) = 0$ es el método de *bisección*, que está basado en el Teorema de Bolzano (Teorema 1.9).

Si $f(x)$ es continua en $[a,b]$ y $f(a)f(b) < 0$, entonces el Teorema 1.9 nos garantiza que existe por lo menos una raíz $\alpha \in (a,b)$ de la ecuación $f(x) = 0$. De esta forma, la estrategia del método de bisección es reducir a la mitad el intervalo dado y seleccionar el subintervalo donde $f(x)$ presenta un cambio de signo (este subintervalo contiene a α). Este procedimiento se repite hasta que el último intervalo seleccionado es tan pequeño que con ello se alcanza la precisión deseada. La sucesión de puntos medios $\{x_k\}$ de los subintervalos I_k inevitablemente tiende a α, dado que la longitud de los subintervalos tiende a cero cuando k tiende a infinito.

Asumamos que $f(x)$ tiene una sola raíz en el intervalo $[a,b]$. El método de bisección inicia estableciendo los extremos del intervalo inicial y la aproximación de la raíz:

$$a_0 = a, \quad b_0 = b, \quad I_0 = [a,b], \quad x_0 = (a_0 + b_0)/2.$$

En el paso $k \geq 1$, selecciona el subintervalo $I_k = [a_k, b_k]$ del intervalo $I_{k-1} = [a_{k-1}, b_{k-1}]$ con el siguiente procedimiento: dado $x_{k-1} = (a_{k-1} + b_{k-1})/2$, si $f(x_{k-1}) = 0$ entonces $\alpha = x_{k-1}$ y el método termina; de lo contrario realiza lo siguiente:

- ◇ si $f(a_{k-1})f(x_{k-1}) < 0$, entonces toma $a_k = a_{k-1}$ y $b_k = x_{k-1}$;
- ◇ si $f(x_{k-1})f(b_{k-1}) < 0$, entonces toma $a_k = x_{k-1}$ y $b_k = b_{k-1}$.

Luego define $x_k = (a_k + b_k)/2$ e incrementa k en 1. Este proceso se repite hasta obtener la precisión deseada. Un ejemplo gráfico de los primeros iterados generados por el procedimiento anterior se muestra en la Figura 4.1.

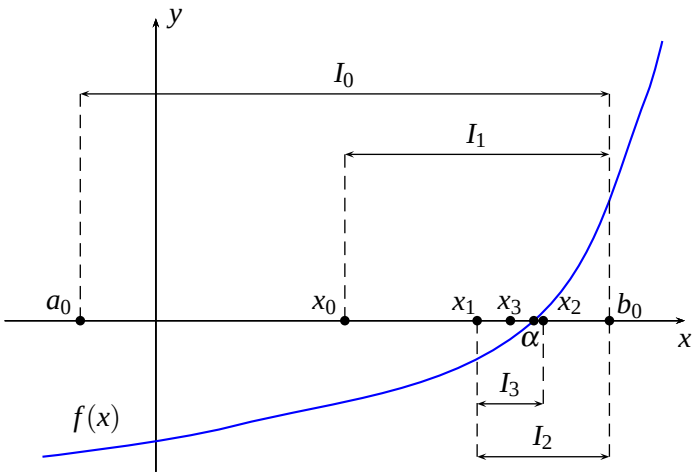


Figura 4.1. Primeras iteraciones del método de bisección

El siguiente teorema presenta algunas propiedades de los iterados del método de bisec-ción.

Teorema 4.1 (Convergencia del método de bisección)**.** *Sea $f(x)$ una función continua en $[a,b]$ tal que $f(a)f(b) < 0$. Asumamos que α es la única raíz de la ecuación $f(x) = 0$ en el intervalo (a,b). Sean $\{a_k\}$, $\{b_k\}$ y $\{x_k\}$ las sucesiones generadas por el método de bisección. Entonces,*

$$\lim_{k\to\infty} a_k = \lim_{k\to\infty} b_k = \lim_{k\to\infty} x_k = \alpha,$$

Además,

$$|x_k - \alpha| \leq \frac{1}{2^{k+1}}(b-a), \quad \text{para } k \geq 0. \tag{4.1}$$

Del Teorema 4.1 se pueden obtener conclusiones interesantes. Inicialmente, este teorema nos dice que el método de bisección converge a la raíz de la ecuación $f(x) = 0$ siempre y cuando $f(x)$ sea continua en $[a,b]$ y exista cambio de signo, además, la ecuación (4.1) permite establecer la precisión deseada. Efectivamente, supongamos que deseamos hallar una aproximación x_n de la raíz α tal que el error sea menor que una tolerancia fija $\varepsilon > 0$, esto es $|x_n - \alpha| < \varepsilon$. Por la desigualdad (4.1), si n es el entero positivo más pequeño tal que

$$\frac{1}{2^{n+1}}(b-a) < \varepsilon,$$

entonces se satisface que $|x_n - \alpha| \leq \varepsilon$. Por la desigualdad anterior, obtenemos que el menor número de iteraciones n que bisección debe hacer para que se cumpla $|x_n - \alpha| < \varepsilon$, está dado por:

$$n \geq \frac{\ln((b-a)/\varepsilon)}{\ln(2)} - 1. \tag{4.2}$$

El Teorema 4.1 también nos indica que bisección tiene una velocidad de convergencia de orden $p = 1$, es decir, bisección converge Q-linealmente.

En el Algoritmo 4.1 se muestra el seudocódigo del método de bisección, cuya implementación se aprecia en la Programa 4.1.

Algoritmo 4.1 (Método de bisección)**.**

Entrada: a, b, $f(x)$ continua en $[a,b]$ tal que $f(a)f(b) < 0$.

$a_0 = a$
$b_0 = b$
$x_0 = (a_0 + b_0)/2$
$k = 0$
while $|f(x_k)| > 0$ **do**
    if $f(a_k)f(x_k) < 0$ **then**
        $a_{k+1} = a_k$
        $b_{k+1} = x_k$
    else
        $a_{k+1} = x_k$
        $b_{k+1} = b_k$
    end if
    $x_{k+1} = (a_{k+1} + b_{k+1})/2$
    $k = k+1$
end while

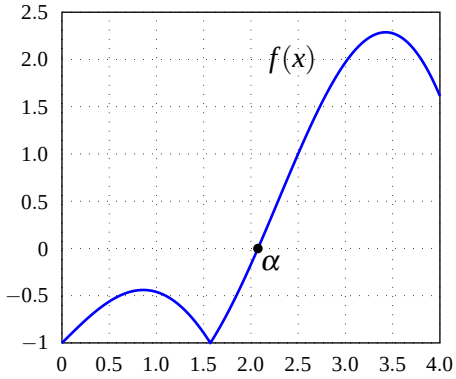
Programa 4.1. Método de Bisección

```
def biseccion(a,b,f,tol=1e-4):
    nmax = np.ceil((np.log((b-a)/tol)/np.log(2))-1)
    fa = f(a)
    fb = f(b)
    k = 0
    while(k < nmax):
        if fa==0:
            return a, k
        elif fb==0:
            return b, k
        x = (a+b)/2
        fx = f(x)
        if fa*fx < 0:
            b, fb = x, fx
        else:
            a, fa = x, fx
        k += 1
    return x, k
```

Ejemplo 4.1. Calculemos la raíz de la ecuación

$$f(x) = x|\cos x| - 1$$

que pertenece al intervalo $(0,4)$ usando el método de bisección, además, el error no debe exceder a $\varepsilon = 10^{-4}$. La gráfica de $f(x)$ muestra que ésta es una función que contiene varios puntos críticos en el intervalo $(0,4)$, y esto pudiera traer inconvenientes en la ejecución de bisección.



Por ello, es necesario buscar un subintervalo de $(0,4)$ que contenga a la raíz α de la ecua-ción $f(x)=0$. Observando nuevamente la gráfica de $f(x)$, notamos que la raíz α pertenece al subintervalo $[1.6,3]$, además, $f(1.6)=-0.95328$ y $f(3)=1.97$, es decir, hay cambio de signo. Ahora, por la ecuación (4.2) obtenemos que el número de iteraciones que bisección debe hacer para que se cumpla $|x_n-\alpha|\leq\varepsilon$ es:

$$n\geq\frac{\ln(1.4/10^{-4})}{\ln(2)}-1\approx 12.773139206719692,$$

lo que indica que $n=13$. Corrimos la función `biseccion`, tomando $a=1.6$, $b=3$ y `tol=1.0e-4`.

```
>>> def f(x): return x*np.abs(np.cos(x))-1
>>> x,it = biseccion(1.6,3,f)
>>> print('Solución:',x,'\nIteraciones:',it)
Solución: 2.0739013671875
Iteraciones: 13
```

Bisección halló la raíz $\alpha=2.073987$ en 13 iteraciones. En la Tabla 4.1 se muestran las iteraciones de la función `biseccion`.

Tabla 4.1. Determinación de un cero de la función $f(x)=x|\cos x|-1$ en el intervalo $[1.6,3]$ usando el método de bisección

k	a_k	b_k	x_k	$\lvert f(x_k)\rvert$
0	1.600000	3.000000	2.300000	5.32e-01
1	1.600000	2.300000	1.950000	2.78e-01
2	1.950000	2.300000	2.125000	1.18e-01
3	1.950000	2.125000	2.037500	8.32e-02
4	2.037500	2.125000	2.081250	1.68e-02
5	2.037500	2.081250	2.059375	3.34e-02
6	2.059375	2.081250	2.070312	8.32e-03
7	2.070312	2.081250	2.075781	4.25e-03
8	2.070312	2.075781	2.073047	2.04e-03
9	2.073047	2.075781	2.074414	1.11e-03
10	2.073047	2.074414	2.073730	4.65e-04
11	2.073730	2.074414	2.074072	3.21e-04
12	2.073730	2.074072	2.073901	7.23e-05
13	2.073901	2.074072	2.073987	1.24e-04

La ventaja más importante de bisección es que sólo requiere que la función sea continua en el intervalo $[a,b]$ y que exista cambio de signo. Otra ventaja importante es que el número de iteraciones para alcanzar una exactitud específica se conoce de antemano. Pero, no todo es bueno. La objeción más importante al método de bisección es su lentitud de convergencia. Como veremos en las siguientes secciones, otros métodos requieren menos iteraciones para alcanzar la misma exactitud.

A pesar de los argumentos en el sentido de que otros métodos encuentran raíces con menos iteraciones, bisección es una herramienta importante del cálculo numérico. Bi-sección suele utilizarse para encontrar un valor aproximado de la raíz, y luego este valor se refina por medio de métodos más eficientes. La razón de esto es que la mayoría de los otros métodos necesitan de una aproximación inicial de la raíz para comenzar su ejecución; al carecer de dicha aproximación pueden fallar por completo.

4.1.2 Método de Newton

Con el fin de diseñar algoritmos con mejores propiedades de convergencia que el método de bisección, es necesario incluir la información de los valores obtenidos por $f(x)$ y, posiblemente, también por su derivada $f'(x)$ (si f es diferenciable) o por una aproximación adecuada. Para ese propósito, asumamos que $\alpha \in [a,b]$ es una raíz de la ecuación $f(x) = 0$ y que $f(x)$ es una función diferenciable en el intervalo $[a,b]$. Consideremos el desarrollo de Taylor de $f(x)$ al rededor de una aproximación $x^+ \in (a,b)$ de α, truncado en el primer término:

$$f(x) = f(x^+) + f'(x^+)(x-x^+) + \frac{1}{2}f''(\xi)(x-x^+),$$

donde ξ es un número entre x^+ y α. De esta forma, evaluando la ecuación anterior en $x = \alpha$, obtenemos:

$$0 = f(\alpha) = f(x^+) + f'(x^+)(\alpha-x^+) + \frac{1}{2}f''(\xi)(\alpha-x^+),$$

de donde se deduce que

$$f(x^+) + f'(x^+)(\alpha-x^+) \approx 0.$$

Ahora, despejando α en esta última ecuación se tiene que

$$\alpha \approx x^+ - \frac{f(x^+)}{f'(x^+)}, \tag{4.3}$$

es decir, el número $x^+ - (f(x^+)/f'(x^+))$ es una aproximación de la raíz α siempre y cuando x^+ esté suficientemente próximo a α. Basándose en la ecuación (4.3), el método de Newton comienza con una aproximación x_0 de α a partir de la cual se define progresivamente una sucesión de aproximaciones dadas por la fórmula:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, \quad k \geq 0. \tag{4.4}$$

Lo que hace el método de Newton es construir la tangente a la gráfica de $f(x)$ en un punto cercano a α, y observar el punto de intersección de esta línea con el eje x, siendo éste punto de intersección la siguiente aproximación de α (observe la Figura 4.2). Este proceso se repite hasta alcanzar la precisión deseada.

A partir de la interpretación geométrica del método de Newton podemos imaginar con facilidad funciones y puntos iniciales para los que él no funciona. En la Figura 4.3 se muestra una de esas

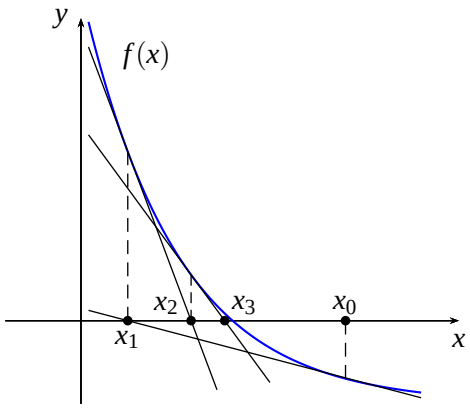


Figura 4.2. Primeras tres iteraciones del método de Newton

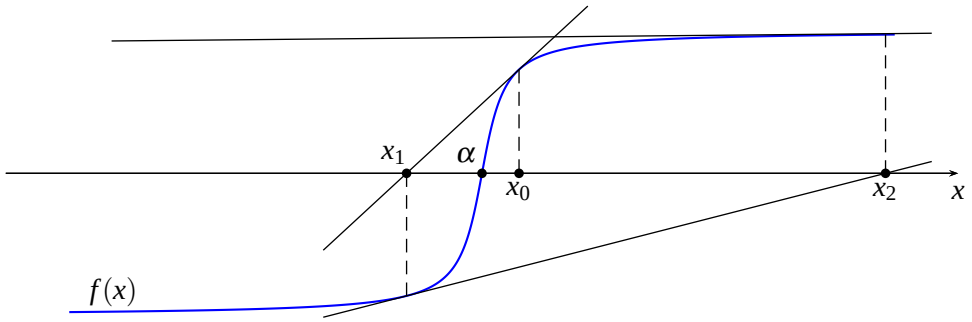


Figura 4.3. Un ejemplo de no convergencia del método de Newton

funciones. En este ejemplo la forma de la curva es tal que para ciertos valores iniciales la sucesión $\{x_k\}$ diverge. Lo que esto significa es que el iterado inicial x_0 debe estar lo suficientemente cerca de la raíz, además, la función debe tener una forma especial muy cerca de la raíz.

En el Algoritmo 4.2 se muestra el pseudocódigo del método de Newton y su implementación se aprecia en el Programa 4.2.

Algoritmo 4.2 (Método de Newton)**.**

Entrada: $f(x)$ diferenciable tal que $f'(\alpha) \neq 0$, iterado inicial x_0.

1: $k = 0$
2: **while** $|f(x_k)| > 0$ **do**
3: $\quad x_{k+1} = x_k - \dfrac{f(x_k)}{f'(x_k)}$
4: $\quad k = k + 1$
5: **end while**

Programa 4.2. Método de Newton

```
def newton(x0,f,df,maxiter=100,tol=1e-4):
    x = x0
    fx = f(x)
    k = 0
    while True:
        if(abs(fx)<=tol):
            print('Termino satisfactoriamente')
            return x,k
        if (k >= maxiter):
            print('Alcanzó el número máximo de iteraciones')
            return
        dfx = df(x)
        if(abs(dfx)<1.0e-15):
            print('Newton falla: df(x_k)=0')
            return
        x = x- fx/dfx
        fx = f(x)
        k += 1
```

Las propiedades de convergencia del método de Newton se presentan en el Teorema 4.2.

Teorema 4.2 (Convergencia del método de Newton)**.** *Sean $f(x)$ una función dos veces continuamente diferenciable y α un cero de $f(x)$ tal que $f'(\alpha) \neq 0$. Si el iterado inicial x_0 del método de Newton se escoge lo suficientemente cerca de α, entonces la sucesión $\{x_k\}$ generada por* (4.4) *converge a α y satisface:*

$$|x_{k+1} - \alpha| \leq \frac{|f''(\alpha)|}{2|f'(\alpha)|} |x_k - \alpha|^2, \quad \textit{para } k \geq 0. \tag{4.5}$$

La ecuación (4.5) nos dice que el error $|x_{k+1} - \alpha|$ es aproximadamente el error $|x_k - \alpha|^2$ multiplicado por una constante. En otras palabras, Newton posee velocidad de convergencia de orden $p = 2$, es decir, *convergencia Q-cuadrática*. En muchas aplicaciones la convergencia Q-cuadrática explica la aparente duplicación de la precisión que se obtiene con cada iteración del método de Newton.

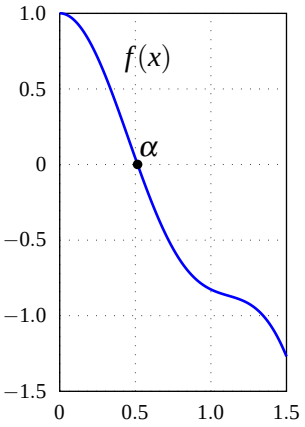
En este punto es bueno preguntarse ¿cuándo paramos las iteraciones de un método iterativo? Vimos que para bisección se conoce de antemano el número de iteraciones necesarias para alcanzar la precisión deseada; pero para otro método iterativo no es así. En general, para el método de Newton o cualquier otro método iterativo para hallar ceros de una función, se utilizan los siguientes criterios de parada:

- ◇ cuando se alcance un máximo número de iteraciones, digamos maxit;
- ◇ terminar las iteraciones cuando el valor absoluto de la diferencia de dos iterados consecutivos es menor que una tolerancia $0 <$ xtol < 1 prefijada, en otras palabras, parar si $|x_k - x_{k-1}| <$ xtol;
- ◇ parar la ejecución del método cuando el valor absoluto del residual en un iterado es menor que la tolerancia $0 <$ ftol < 1, es decir, parar si $|f(x_k)| <$ ftol.

Ejemplo 4.2. Consideremos la función

$$f(x) = \cos^2(2x) - x^2.$$

Usemos el método de Newton para encontrar la raíz $\alpha \approx 0.514933265$ de la ecuación $f(x) = 0$ en el intervalo $(0, 1.5)$, además, el error no debe exceder $\varepsilon = 5 \times 10^{-8}$. De la definición de $f(x)$ se infiere que es una función diferenciable en todo punto del intervalo $[0, 1.5]$, cuya derivada es $f'(x) = -4\cos(2x)\,\mathrm{sen}(2x) - 2x$. Además, observamos en la gráfica de $f(x)$ en el intervalo $[0, 1.5]$, que ésta no posee puntos críticos (puntos de máximos o mínimos donde se anule la derivada).



Seguidamente corrimos la función `newton`, tomando $x_0 = 0.2$ como una aproximación inicial de la raíz α.

```
>>> def f(x): return np.cos(2*x)**2-x**2
>>> def df(x): return -4*np.cos(2*x)*np.sin(2*x)-2*x
>>> x, it = newton(0.2,f,df,tol=5.0e-8)
Newton termino satisfactoriamente
>>> print('Solución:',x,'\nIteraciones:',it)
Solución: 0.5149332640638451
Iteraciones: 4
```

En la Tabla 4.2 se muestran las iteraciones de `newton`.

Tabla 4.2. Determinación del cero $\alpha \approx 0.514933265$ de la función $f(x) = \cos^2(2x) - x^2$ usando el método de Newton con iterado inicial $x_0 = 0.2$

k	x_k	$\lvert x_k - \alpha \rvert$
0	0.200000000	0.314933265
1	0.640588645	0.125655380
2	0.502204519	0.012728746
3	0.514889664	0.000043601
4	0.514933264	0.000000001

Note que Newton necesitó solo de 4 iteraciones para alcanzar la precisión deseada. También se nota que el número de cifras significativas de precisión se duplican a partir de la segunda iteración.

4.1.3 Método de la Secante

Recordemos que la iteración de Newton se define mediante la expresión

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, \quad k \geq 0.$$

Una de las desventajas del método de Newton es que utiliza la derivada de la función. Para superar esta dificultad se han propuesto una serie de métodos. Uno de ellos consiste en reemplazar $f'(x_k)$ en la iteración de Newton por un cociente de *diferencias* como el siguiente:

$$f'(x_k) \approx \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}. \tag{4.6}$$

La aproximación dada por (4.6) tiene su origen en la definición de la derivada de $f(x)$ en términos de un límite, a saber:

$$f'(a) = \lim_{x \to a} \frac{f(x) - f(a)}{x - a}.$$

Cuando se reemplaza $f'(x_k)$ por la expresión (4.6) en la forma iterativa de Newton, el método resultante se conoce como *método de la secante*. Su fórmula iterativa es

$$x_{k+1} = x_k - f(x_k)\left[\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}\right], \quad k \geq 1. \tag{4.7}$$

Ya que el cálculo de x_{k+1} requiere conocer x_k y x_{k-1}, se debe dar al principio los valores de dos puntos iniciales x_0 y x_1.

La interpretación gráfica del método de la secante es similar a la del método de Newton. La línea tangente a la curva en el punto $(x_k, f(x_k))$ se reemplaza por una línea secante a la curva en los puntos $(x_{k-1}, f(x_{k-1}))$ y $(x_k, f(x_k))$ (ver la Figura 4.4). El punto de intersección de esta recta secante con el eje x es la siguiente aproximación de α. Este proceso se repite hasta alcanzar la precisión deseada. De esta interpretación geométrica inferimos que existen funciones para las que el método de la secante no funciona (por ejemplo, funciones muy planas).

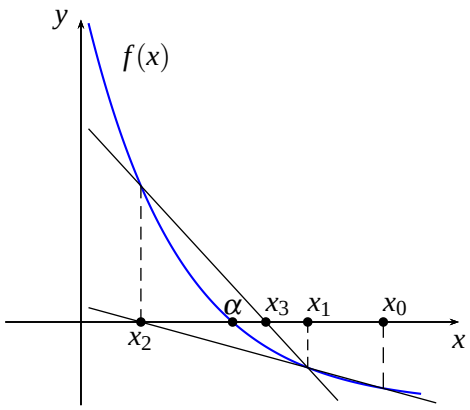


Figura 4.4. Primeras tres iteraciones del método de la secante

En el Algoritmo 4.3 se muestra el pseudocódigo del método de la secante y en el Programa 4.3 se muestra su implementación.

Algoritmo 4.3 (Método de la secante)**.**

Entrada: $f(x)$ y x_0, x_1 aproximaciones de la raíz α.

1: $k = 1$
2: **while** $|f(x_k)| > 0$ **do**
3: $\quad x_{k+1} = x_k - \left[\dfrac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}\right] f(x_k)$
4: $\quad k = k + 1$
5: **end while**

Programa 4.3. Método de la secante

```
def secante(x0,x1,f,maxiter=100,tol=1e-4):
    xk1, xk = x0, x1
    fxk1, fxk = f(xk1), f(xk)
    k = 1
    while True:
        if(abs(fxk)<=tol):
            print('Terminó satisfactoriamente')
            return xk,k
        if (k >= maxiter):
            print('Alcanzó el número máximo de iteraciones')
            return
        dfxk = (xk-xk1)/(fxk-fxk1)
        if(abs(dfxk)<1.0e-15):
            print('Secante falla')
            return
        xk1,fxk1 = xk,fxk
        xk = xk - dfxk*fxk
        fxk = f(xk)
        k += 1
```

En el Teorema 4.3 se presentan algunas de las propiedades de convergencia del método de la secante.

Teorema 4.3 (Convergencia del método de la secante)**.** *Sean $f(x)$ una función dos veces continuamente diferenciable y α un cero de $f(x)$ tal que $f'(\alpha) \neq 0$. Si los iterados iniciales x_0 y x_1 del método de la secante se escogen suficientemente cerca de α, entonces la sucesión $\{x_k\}$ generada por* (4.7) *converge a α y satisface:*

$$|x_{k+1} - \alpha| \leq \left(\frac{|f''(\alpha)|}{2|f'(\alpha)|}\right)^{0.62} |x_k - \alpha|^p, \quad \text{para } k \geq 1, \tag{4.8}$$

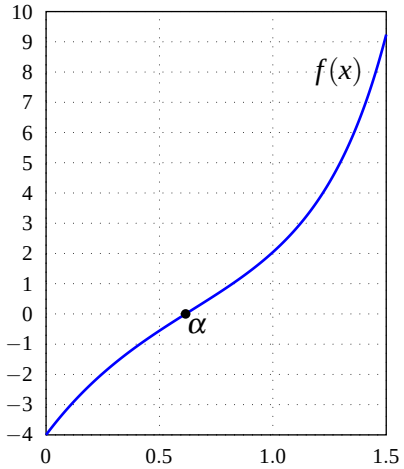
donde $p = (1+\sqrt{5})/2 \approx 1.62$.

El Teorema 4.3 establece que el método de la secante tiene velocidad de convergencia local de orden $p = (1+\sqrt{5})/2 \approx 1.62$, es decir, secante converge Q-superlinealmente.

Ejemplo 4.3. La función

$$f(x) = e^{x^2} - \frac{5}{e^{2x}}$$

tiene un cero positivo conocido $\alpha = \sqrt{1+\ln(5)} - 1 \approx 0.615375471$. Usemos el método de la secante para hallar una aproximación de α de tal manera que el error no exceda $\varepsilon = 10^{-8}$. La gráfica de $f(x)$ en el intervalo $[0, 1.5]$ muestra que α es la única raíz en ese intervalo y aquí $f(x)$ no posee puntos críticos.



Seguidamente corrimos la función `secante`, tomando $x_0 = 1$ y $x_1 = 0.5$ como aproximaciones iniciales de la raíz α.

```
>>> def f(x): return np.exp(x**2)-(5/np.exp(2*x))
>>> x,it = secante(1,0.5,f,tol=1.0e-8)
Terminó satisfactoriamente
>>> print('Solución:',x,'\nIteraciones:',it)
Solución: 0.6153754710372857
Iteraciones: 5
```

En la Tabla 4.3 se muestran las iteraciones de `secante`.

Tabla 4.3. Determinación del cero $\alpha = \sqrt{1+\ln(5)} - 1 \approx 0.615375471$ de la función $f(x) = e^{x^2} - 5e^{-2x}$ usando el método de la secante con iterados iniciales $x_0 = 1$, $x_1 = 0.5$

k	x_k	$\lvert x_k - \alpha \rvert$
0	1.000000000	0.38462452900
1	0.500000000	0.11537547100
2	0.606926582	0.00844888942
3	0.615201056	0.00017441535
4	0.615375349	0.00000012183
5	0.615375471	0.00000000004

Observe que el método de la secante necesitó solo de 5 iteraciones para alcanzar la precisión deseada.

4.1.4 Método de Müller

El método de la secante también puede interpretarse utilizando la interpolación lineal de iterados. En sí, secante construye el polinomio interpolante $p_1(x)$ que interpola los puntos $(x_{k-1}, f(x_{k-1}))$ y $(x_k, f(x_k))$, para aproximar a $f(x)$; luego obtiene una aproximación de la raíz de $f(x) = 0$ como el cero de la ecuación $p_1(x) = 0$. El método de *Müller* también puede interpretarse a través de la interpolación, pero en este caso en la interpolación cuadrática. Cuando contamos con tres iterados, podemos tomar una aproximación de la función $f(x)$ interpolando con un polinomio de grado dos y resolver una ecuación de segundo grado para obtener una aproximación de la raíz (ver Figura 4.5).

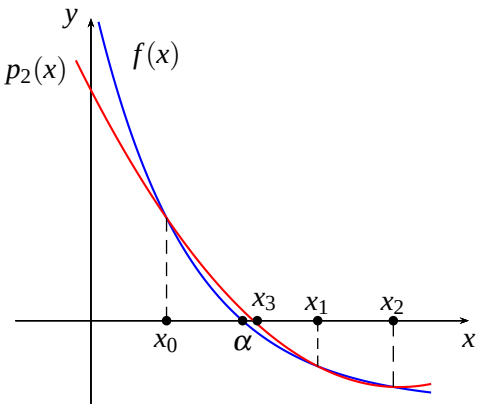


Figura 4.5. Aproximación de la raíz con interpolación cuadrática

Supongamos que tenemos tres iterados x_0, x_1 y x_2 con sus correspondientes valores en la función $f(x_0)$, $f(x_1)$ y $f(x_2)$. El polinomio interpolante de grado 2, en su forma de Newton, que interpola los puntos $(x_0, f(x_0))$, $(x_1, f(x_1))$ y $(x_2, f(x_2))$ está dado por

$$p_2(x) = f(x_2) + f[x_2,x_1](x-x_2) + f[x_2,x_1,x_0](x-x_2)(x-x_1).$$

Este polinomio podemos expresarlo equivalentemente como (se deja al lector demostrar esta equivalencia):

$$p_2(x) = a(x-x_2)^2 + b(x-x_2) + c,$$

donde

$$a = f[x_2,x_1,x_0], \qquad b = f[x_2,x_1] + (x_2-x_1)f[x_2,x_1,x_0], \qquad c = f(x_2).$$

Ahora bien, una solución de la ecuación $p_2(x) = 0$ se puede tomar como una aproximación del cero de la ecuación $f(x) = 0$. En otras palabras, las raíces del polinomio $p_2(x)$ pueden ser aproximaciones del cero de $f(x)$. Este polinomio tiene dos raíces

$$x = x_2 + \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Normalmente se toma la raíz más cercana a x_2 (ver Figura 4.5), la cual se puede calcular por

$$x_3 = x_2 + \frac{-b + \text{sgn}(b)\sqrt{b^2 - 4ac}}{2a}.$$

Esta última expresión puede considerar la anulación de dos términos muy cercanos. Por ello es preferible una expresión más estable (se deja como ejercicio para el lector demostrar la equivalencia de estas expresiones):

$$x_3 = x_2 - \frac{2c}{b + \text{sgn}(b)\sqrt{b^2 - 4ac}}.$$

El método de Müller consiste en realizar iterativamente el proceso anterior conservando los últimos tres iterados. La fórmula iterativa del método de Muller es:

$$x_{k+1} = x_k - \frac{2c}{b + \text{sgn}(b)\sqrt{b^2 - 4ac}}, \quad k \geq 2, \tag{4.9}$$

donde

$$a = f[x_k, x_{k-1}, x_{k-2}], \qquad b = f[x_k, x_{k-1}] + (x_k - x_{k-1}) f[x_k, x_{k-1}, x_{k-2}], \qquad c = f(x_k).$$

En el Algoritmo 4.4 mostramos el pseudocódigo del método de Müller y en el Programa 4.4 su implementación.

Algoritmo 4.4 (Método de Müller)**.**

Entrada: $f(x)$ y x_0, x_1, x_1 aproximaciones de la raíz α.

1: $k = 2$
2: **while** $|f(x_k)| > 0$ **do**
3: $\quad a = f[x_k, x_{k-1}, x_{k-2}]$
4: $\quad b = f[x_k, x_{k-1}] + (x_k - x_{k-1}) f[x_k, x_{k-1}, x_{k-2}]$
5: $\quad c = f(x_k)$
6: $\quad x_{k+1} = x_k - \dfrac{2c}{b + \text{sgn}(b)\sqrt{b^2 - 4ac}}$
7: $\quad k = k + 1$
8: **end while**

Programa 4.4. Método de Müller

```
def muller(x0,x1,x2,f,maxiter=100,tol=1e-4):
    xk_2, xk_1, xk = x0, x1, x2
    fxk2, fxk1 = f(xk_2), f(xk_1)
    k = 2
    while True:
        fxk = f(xk)
        if(abs(fxk)<=tol):
            return xk, k
        if (k >= maxiter):
            print('Alcanzó el número máximo de iteraciones')
            return xk, k
        df1 = (fxk1 - fxk2) / (xk_1 - xk_2)
        df2 = (fxk - fxk1) / (xk - xk_1)
        a = (df2-df1) / (xk-xk_2)
        b = df2 + (xk-xk_2)*a
        c = fxk
        xk_2, fxk2 = xk_1, fxk1
        xk_1, fxk1 = xk, fxk
        xk = xk - (2*c / (b + np.sign(b)*np.sqrt(b**2 - 4*a*c)))
        k += 1
```

El Teorema 4.4 presenta un resultado de convergencia del método de Müller.

Teorema 4.4 (Convergencia del método de Müller)**.** *Sean $f(x)$ una función dos veces continuamente diferenciable y α un cero de $f(x)$ tal que $f'(\alpha) \neq 0$. Si los iterados iniciales x_0, x_1 y x_2 del método de Müller se escogen suficientemente cerca de α, entonces la sucesión $\{x_k\}$ generada por* (4.9) *converge a α y existe una constante $C > 0$ tal que:*

$$|x_{k+1} - \alpha| \leq C\,|x_k - \alpha|^p, \quad \text{para } k \geq 2, \tag{4.10}$$

donde p es la raíz más grande del polinomio $x^3 - x^2 - x - 1$, es decir,

$$p = \frac{1}{3} + \frac{4}{9\left(\frac{\sqrt{33}}{9} + \frac{19}{27}\right)^{1/3}} + \left(\frac{\sqrt{33}}{9} + \frac{19}{27}\right)^{1/3} \approx 1.839.$$

El Teorema 4.4 establece que la velocidad de convergencia local del método de Müller es de orden $p \approx 1.839$, es decir, el método de Müller, igual que el método de la secante, posee convergencia Q-superlineal, pero la velocidad de Müller es ligeramente superior a la de la secante ($p \approx 1.62$).

Ejemplo 4.4. Consideremos la función $f(x)$ del Ejemplo 4.3, cuyo cero es $\alpha = \sqrt{1+\ln(5)} - 1 \approx 0.615375471$. Usemos el método de Müller para hallar una aproximación de α de tal manera que el error no exceda $\varepsilon = 10^{-8}$. Corrimos la función `muller`, tomando $x_0 = 0.1$, $x_1 = 0.5$ y $x_2 = 1.0$ como aproximaciones iniciales de la raíz α.

```
>>> def f(x): return np.exp(x**2)-(5/np.exp(2*x))
>>> x,it = muller(.1,.5,1.,f,tol=1.0e-8)
Terminó satisfactoriamente
>>> print('Solución:',x,'\nIteraciones:',it)
Solución: 0.6153754710389923
Iteraciones: 8
```

En la Tabla 4.4 se muestran las iteraciones de `muller`.

Tabla 4.4. Determinación del cero $\alpha = \sqrt{1+\ln(5)} - 1 \approx 0.615375471$ de la función $f(x) = e^{x^2} - 5e^{-2x}$ usando el método de Müller con iterados iniciales $x_0 = 0.1$, $x_1 = 0.5$ y $x_2 = 1.0$

k	x_k	$\lvert x_k - \alpha \rvert$
0	0.10000000000	0.5153754710389
1	0.50000000000	0.1153754710389
2	1.00000000000	0.3846245289611
3	0.55813759786	0.0572378731790
4	0.60931100948	0.0060644615616
5	0.61599388121	0.0006184101758
6	0.61537113272	0.0000043383147
7	0.61537547334	0.0000000023045
8	0.61537547104	0.0000000000001

Note que el método de Müller necesitó de 8 iteraciones para alcanzar la precisión deseada. ■

4.1.5 Iteración de Punto Fijo

En este apartado se proporciona un marco completamente general para encontrar las raíces de una función no lineal. El método que estudiaremos está basado en el hecho que, para una función dada $f : [a,b] \to \mathbb{R}$, siempre es posible transformar el problema $f(x) = 0$ en un problema equivalente $x - \phi(x) = 0$, donde la función auxiliar $\phi : [a,b] \to \mathbb{R}$ debe ser escogida de manera que $\phi(\alpha) = \alpha$ siempre que $f(\alpha) = 0$. Por lo tanto, aproximar los ceros de una función se transforma en el problema de encontrar los puntos fijos de la función ϕ, que se realiza por el siguiente algoritmo iterativo: dado $x_0 \in [a,b]$ se genera una sucesión $\{x_k\}$ como

$$x_{k+1} = \phi(x_k), \quad k \geq 0. \tag{4.11}$$

Se dice que (4.11) es una *iteración de punto fijo* y ϕ es su *función de iteración* asociada. Algunas veces, (4.11) también se referencia como *iteración de Picard* o *iteración funcional* para la solución de $f(x) = 0$.

La escogencia de ϕ no es única. Por ejemplo, una función de la forma $\phi(x) = x + F(f(x))$, donde F es una función continua tal que $F(0) = 0$ es una función de iteración admisible. En la Figura 4.6 se observan ejemplos gráficos de la iteración de punto fijo. En el caso de la izquierda la sucesión $\{x_k\}$ generada converge al punto fijo, en cambio la sucesión generada en el caso de la derecha no converge.

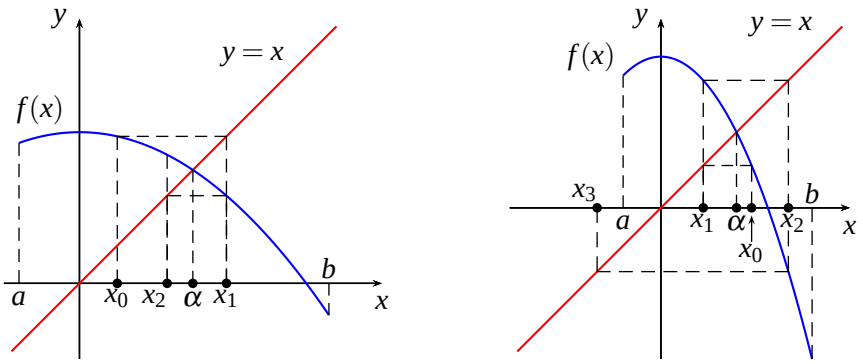


Figura 4.6. Representación de la iteración de punto fijo para dos funciones diferentes.

Los ejemplos de la Figura 4.6 nos permiten asegurar que existen funciones de iteración para las que la sucesión generada por (4.11) no converge a un punto fijo. Veremos un teorema que proporciona las condiciones suficientes para que la iteración de punto fijo (4.11) converja a un punto fijo, que también es una raíz del problema equivalente $f(x) = 0$. Antes de ello, mostramos a continuación el Algoritmo 4.5 de la iteración de punto fijo y en el Programa 4.5 su implementación.

Algoritmo 4.5 (Iteración de punto fijo)**.**

Entrada: $\phi(x)$ (función de iteración) y $x_0 \in [a,b]$ iterado inicial.

1: $k = 0$
2: **while** $|\phi(x_k) - x_k| > 0$ **do**
3: $\quad x_{k+1} = \phi(x_k)$
4: $\quad k = k + 1$
5: **end while**

Programa 4.5. Método de iteración de punto fijo

```
def pfijo(x0,phi,maxiter=100,tol=1e-4):
    x = x0
    k = 0
    while True:
        if (k >= maxiter):
            return
        phix = phi(x)
        if(abs(x-phix)<=tol):
            return x,k
        x = phix
        k += 1
```

Teorema 4.5 (Convergencia de la iteración de punto fijo)**.** *Considere la sucesión* $\{x_k\}$ *generada por la iteración de punto fijo* (4.11)*, comenzando en* $x_0 \in [a,b]$ *dado. Asumamos que:*

1. $\phi : [a,b] \to [a,b]$*;*
2. ϕ *es una función continuamente diferenciable;*
3. *existe* $\lambda < 1$ *tal que* $|\phi'(x)| < \lambda$ *para todo* $x \in [a,b]$*.*

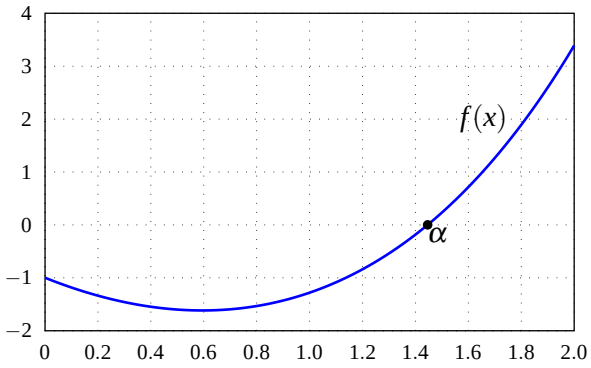
Entonces, ϕ *tiene un único punto fijo* α *en* $[a,b]$ *y la sucesión* $\{x_k\}$ *converge a* α *para cualquier iterado inicial* $x_0 \in [a,b]$*. Además, se tiene que*

$$\lim_{k\to\infty} \frac{x_{k+1}-\alpha}{x_k-\alpha} = \phi'(\alpha). \tag{4.12}$$

De la ecuación (4.12) uno deduce que la convergencia de la iteración de punto fijo al menos es Q-lineal, esto es, para k suficientemente grande el error en la iteración $k+1$ es menor que el error en la iteración k multiplicado por una constante $\phi'(\alpha)$ la cual es independiente de k y cuyo valor absoluto es menor que 1. Se deja como ejercicio para el lector determinar qué hipótesis del Teorema 4.5 no cumple la función iterativa del caso de la derecha de la Figura 4.6. En los siguientes ejemplos se muestra el uso del Teorema 4.5.

Ejemplo 4.5. La función $\phi_1(x) = \cos x$ satisface todas las hipótesis del Teorema 4.5 en el intervalo $[0,\pi/3]$. En efecto, ϕ_1 es continuamente diferenciable en $[0,\pi/3]$; además, $\phi_1(x) \in [0,\pi/3]$ y $|\phi_1'(x)| = |\text{sen}(x)| < 1$, para todo $x \in [0,\pi/3]$. En cambio, la función $\phi_2(x) = x^2 - 1$, que posee dos puntos fijos $\alpha_\pm = (1 \pm \sqrt{5})/2$, no satisface la hipótesis 3 del Teorema 4.5, ya que $|\phi_2'(\alpha_\pm)| = |1 \pm \sqrt{5}| > 1$.

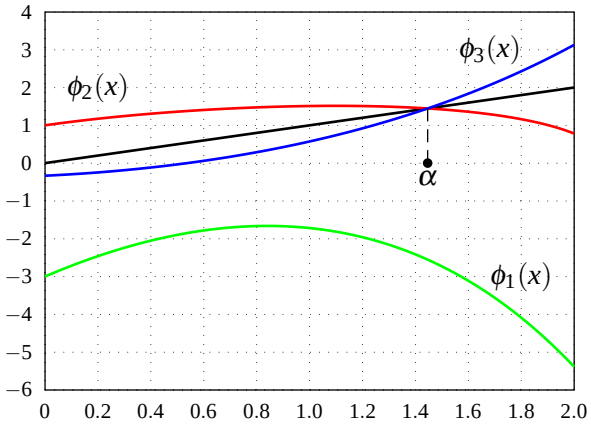
Ejemplo 4.6. Se sabe que la función $f(x) = x^2 - 3x + e^x - 2$ tiene dos raíces: una negativa y otra positiva. Determinemos la raíz positiva de $f(x)$ usando la iteración de punto fijo.



La gráfica de $f(x)$ muestra que su raíz positiva está en el intervalo $[0,2]$; más aún, la raíz está en el intervalo $[1,2]$. Ahora, realizando operaciones algebraicas en la ecuación $f(x) = 0$ (esto es, sumar, restar, multiplicar o despejar x en la ecuación $f(x) = 0$), obtenemos las siguientes tres funciones de iteración:

$$\phi_1(x) = 4x - x^2 - e^x - 2, \qquad \phi_2(x) = \sqrt{3x - e^x + 2}, \qquad \phi_3(x) = \frac{x^2 + e^x - 2}{3}.$$

Seguidamente mostramos las gráficas de estas funciones de iteración en el intervalo $[0,2]$. Se deja como ejercicio para el lector probar con otras funciones de iteración. Ahora bien, se observa que ϕ_2 y ϕ_3 tienen un punto fijo en el intervalo, pero ϕ_1 no tiene. También observamos que existen puntos $x \in [0,2]$ tales que $\phi_3(x)$ no pertenece al intervalo $[0,2]$.

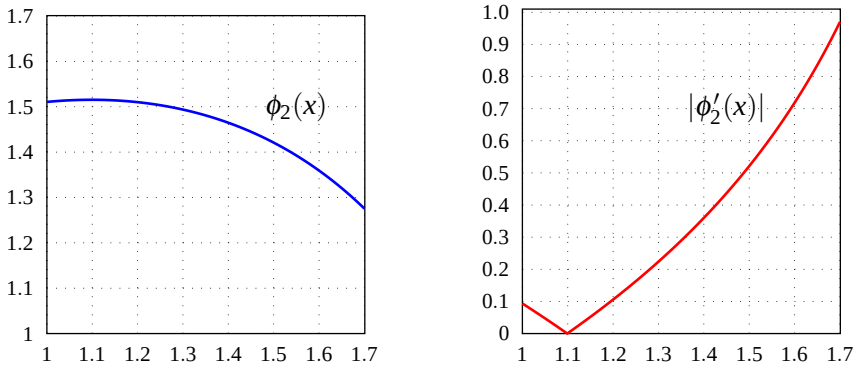


De esta forma, con lo que hemos verificado podemos descartar a la función ϕ_1, ya que no posee puntos fijos en el intervalo $[1,2]$. La función ϕ_2 satisface la hipótesis 2 del Teorema 4.5, pero no satisface la hipótesis 1. Con esta información pudiéramos descartar a la función ϕ_2, pero aún no haremos eso, debido a que también se debe cumplir la hipótesis 3, lo cual nos podría conducir a reducir el intervalo de búsqueda, es decir, hallar un subintervalo de $[1,2]$ donde también se cumpla la hipótesis 3. Ahora, la función ϕ_2 satisface las hipótesis 1 y 2 del Teorema 4.5, pero se puede verificar que ella no satisface la hipótesis 3 en el intervalo $[1,2]$ (se deja al lector verificar esta

afirmación). Encontremos un subintervalo de $[1,2]$ tal que allí las funciones ϕ_2 y ϕ_3 satisfagan las hipótesis del Teorema 4.5. Tenemos que

$$\phi_2'(x) = \frac{3-e^x}{2\sqrt{3x-e^x+2}} \quad \text{y} \quad \phi_3'(x) = \frac{2x+e^x}{3}.$$

Se deja al lector verificar que no existe un subintervalo de $[1,2]$ tal que la función ϕ_3 cumple con las hipótesis del Teorema 4.5.



En cambio, la función ϕ_2 es continuamente diferenciable en el intervalo $[1,1.7]$, además, observando la gráfica de $\phi_2(x)$ y de $|\phi_2'(x)|$, se tiene que las hipótesis 1 y 3 del Teorema 4.5 se satisfacen. En otras palabras, ϕ_2 cumple con todas las hipótesis de éste último teorema. En consecuencia, usaremos a

$$x_{k+1} = \phi_2(x_k) = \sqrt{3x_k - e^{x_k} + 2}, \quad k \geq 0,$$

como iteración de punto fijo, para determinar la raíz positiva de $f(x)$. Tomamos $x_0 = 1.2$ como aproximación inicial y pararemos las iteraciones cuando se satisfaga $|x_k - x_{k-1}| < 10^{-8}$. Corrimos la función `pfijo` tomando $x_0 = 1.2$ y `tol=1.0e-8`.

```
>>> def phi(x): return np.sqrt(3*x-np.exp(x)+2)
>>> x,it = pfijo(1.2,phi,1.0e-8)
Punto fijo termino satisfactoriamente
>>> print('Solución:',x,'\nIteraciones:',it)
Solución: 1.4462386893463444
Iteraciones: 21
```

Observe que la iteración de punto fijo requiere de 22 iteraciones para alcanzar la precisión deseada y encontrar el punto fijo $\alpha = 1.446238685$ de $\phi_2(x)$, que a su vez es la raíz positiva de la función $f(x) = x^2 - 3x + e^x - 2$. En la Tabla 4.5 se muestran las iteraciones de `pfijo`.

Tabla 4.5. Determinación de la raíz positiva de la función $f(x) = x^2 - 3x + e^x - 2$, usando la iteración de punto fijo con la función de iteración $\phi_2(x) = \sqrt{3x - e^x + 2}$ e iterado inicial $x_0 = 1.2$.

k	x_k	$\lvert x_k - x_{k-1}\rvert$
0	1.200000000	
1	1.509928170	0.309928170
2	1.415407666	0.094520504
3	1.458786495	0.043378830
4	1.440701944	0.018084552
5	1.448601492	0.007899549
6	1.445215483	0.003386009
7	1.446679011	0.001463527
8	1.446048682	0.000630328
9	1.446320578	0.000271896
10	1.446203372	0.000117206
11	1.446253911	0.000050539
12	1.446232122	0.000021789
13	1.446241516	0.000009395
14	1.446237466	0.000004051
15	1.446239212	0.000001746
16	1.446238459	0.000000753
17	1.446238784	0.000000325
18	1.446238644	0.000000140
19	1.446238704	0.000000060
20	1.446238678	0.000000026
21	1.446238689	0.000000011
22	1.446238685	0.000000005

El procedimiento descrito en el Ejemplo 4.6 para encontrar una iteración funcional, aunque parezca largo, nos garantiza teóricamente que la iteración funcional convergerá a una raíz. En la práctica, desafortunadamente este procedimiento solo lo realizan los usuarios que tienen cierto conocimiento de la teoría de punto fijo. En general, un usuario cualquiera simplemente obtiene una función iterativa y luego corre la iteración de punto fijo hasta que satisfaga un criterio de parada o alcance un máximo número de iteraciones. Si se alcanza el máximo número de iteraciones, entonces el usuario busca otro iterado inicial o crea otra función de iteración y prueba nuevamente. Este procedimiento de ensayo y error se repite hasta que se determine la solución deseada.

Según el Teorema 4.5 y como comprobamos en el Ejemplo 4.6, existen funciones de iteración con las que se consigue convergencia lineal al punto fijo. Esto nos conduce a preguntarnos: ¿existirán funciones de iteración con las que se pueda obtener un orden de velocidad de convergencia superior a 1? La respuesta a esta pregunta es afirmativa. Una de ellas es la función iteración

$$\phi(x) = x - \frac{f(x)}{f'(x)},$$

cuya iteración funcional da a lugar al método de Newton y, además, la velocidad de convergencia obtenida es Q-cuadrática. En el siguiente teorema damos las condiciones que debe satisfacer una función de iteración $\phi(x)$ para que la convergencia de la iteración de punto fijo sea de orden p.

Teorema 4.6. *Sean p un entero positivo y $\phi(x)$ una función p-veces continuamente diferenciable en el intervalo $[a,b]$. Supongamos que $\phi(x)$ tiene un solo punto fijo $\alpha \in [a,b]$. Si se cumple*

$$\phi'(\alpha) = \phi''(\alpha) = \cdots = \phi^{(p-1)}(\alpha) = 0 \tag{4.13}$$

y

$$\phi^{(p)}(\alpha) \neq 0, \tag{4.14}$$

entonces la sucesión $\{x_k\}$ generada por la iteración de punto fijo (4.11) *converge a α para cualquier iterado inicial $x_0 \in [a,b]$. Además, existe una constante $C > 0$ tal que*

$$|x_{k+1} - \alpha| \leq C\,|x_k - \alpha|^p, \quad \textit{para } k \geq 0. \tag{4.15}$$

Como consecuencia inmediata del Teorema 4.6 obtenemos que la velocidad de convergencia de la iteración de punto fijo es de orden p. Usaremos el Teorema 4.6 para demostrar que el método de Newton tiene convergencia Q-cuadrática. Asumamos que $f(x)$ es una función tres veces continuamente diferenciable en $[a,b]$. Supongamos que $f'(\alpha) \neq 0$ y $f''(\alpha) \neq 0$. Sea

$$\phi(x) = x - \frac{f(x)}{f'(x)}$$

la función de iteración de punto fijo. Como $f(\alpha) = 0$, $f'(\alpha) \neq 0$ y

$$\phi'(x) = 1 - \frac{(f'(x))^2 - f(x)f''(x)}{(f'(x))^2} = \frac{f(x)f''(x)}{(f'(x))^2},$$

entonces $\phi'(\alpha) = 0$, es decir, (4.13) es cierta. Ahora, como $f''(\alpha) \neq 0$ y

$$\phi''(x) = \frac{f''(x)}{f'(x)} + f(x)\left[\frac{f'(x)f'''(x) - 2(f''(x))^2}{(f'(x))^3}\right],$$

entonces $\phi''(\alpha) \neq 0$, es decir, (4.13) se cumple. Dado que (4.13) y (4.14) son ciertas, por el Teorema 4.6, la iteración de punto fijo

$$x_{k+1} = \phi(x_k) = x_k - \frac{f(x_k)}{f'(x_k)}$$

satisface la Ecuación (4.15) con $p = 2$, esto es, converge Q-cuadráticamente a α; en otras palabras, hemos demostrado que la velocidad de convergencia de Newton es Q-cuadrática.

4.2 Sistemas de Ecuaciones no Lineales

Hasta este punto, hemos limitado nuestra atención a encontrar la solución de una ecuación no lineal $f(x) = 0$, cuando f es una función definida de $\mathbb{R}$ en sí mismo. Ahora consideraremos el problema de resolver un *sistema de ecuaciones no lineales*

$$\left\{\begin{array}{rcl} f_1(x_1,x_2,\ldots,x_n) &=& 0 \\ f_2(x_1,x_2,\ldots,x_n) &=& 0 \\ \vdots & & \\ f_n(x_1,x_2,\ldots,x_n) &=& 0 \end{array}\right. \tag{4.16}$$

el cual se puede escribir en forma vectorial

$$\mathbf{F}(\mathbf{x}) = 0, \tag{4.17}$$

donde $\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$ y $\mathbf{F}(\mathbf{x}) = (f_1(\mathbf{x}), \ldots, f_n(\mathbf{x}))$ es una función definida de $\mathbb{R}^n$ en sí mismo. Hallar una solución de éstas n ecuaciones no lineales simultáneas es una tarea más compleja que el encontrar una raíz de una sola ecuación. Más aún, sólo verificar la existencia de soluciones (o el número de ellas) de un sistema de ecuaciones no lineales es laborioso y en algunos casos casi imposible, porque debe realizarse un análisis que involucra el cálculo de derivadas y otras técnicas más avanzadas.

Para determinar la existencia de soluciones de una ecuación no lineal $f(x) = 0$, simplemente se construye la gráfica de la función $f(x)$ y luego se identifican gráficamente los puntos de cortes de $f(x)$ con el eje x. En el caso de sistemas de ecuaciones no lineales, para dimensiones $n \geq 3$, el método gráfico es inviable. Cada ecuación de un sistema de ecuaciones no lineales define una hipersuperficie (un hiperplano si la ecuación es lineal) en $\mathbb{R}^n$, y una solución para el sistema es cualquier punto que pertenece a la intersección de estas hipersuperficies. Pero, las hipersuperficies pueden intersectarse, o no se intersectan, de muchas formas posibles. Como resultado de ello, es que el número de soluciones de un sistema no lineal puede ser cualquiera. Ilustremos esto con algunos ejemplos de sistemas de dimensión dos.

Ejemplo 4.7. Sea

$$\mathbf{F}(\mathbf{x}) = \begin{bmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

donde $f_1 : \mathbb{R}^2 \to \mathbb{R}$ y $f_2 : \mathbb{R}^2 \to \mathbb{R}$. Consideremos las funciones

$$f_1(x_1, x_2) = x_1^2 - x_2 + \beta, \quad f_2(x_1, x_2) = -x_1 + x_2^2 + \beta,$$

donde $\beta \in [-1, 1]$ es un parámetro. Ambas funciones $f_1(x_1, x_2)$ y $f_2(x_1, x_2)$, representan parábolas en $\mathbb{R}^2$. Los siguientes casos aparecen (ver Figura 4.7):

(a) $\beta = 1$: el sistema no tiene solución;

(b) $\beta = \frac{1}{4}$: el sistema tiene solución única: $x_1 = x_2 = \frac{1}{2}$;

(c) $\beta = 0$: el sistema tiene dos soluciones: $x_1 = x_2 = 0$; $x_1 = x_2 = 1$;

(d) $\beta = -1$: el sistema tiene cuatro soluciones: $x_1 = -1$, $x_2 = 0$; $x_1 = 0$, $x_2 = -1$; $x_1 = x_2 = \frac{1}{2}(1 \pm \sqrt{5})$.

De igual forma, el sistema

$$\begin{cases} f_1(x_1, x_2) = \frac{1}{2} x_1 \operatorname{sen}(\frac{1}{2}\pi x_1) - x_2 = 0 \\ f_2(x_1, x_2) = x_2^2 - x_1 + 1 = 0 \end{cases}$$

tiene un conjunto infinito numerable de soluciones (ver Figura 4.8 (a)), mientras que el sistema

$$\begin{cases} f_1(x_1, x_2) = x_1^2 - |x_2| = 0 \\ f_2(x_1, x_2) = x_1^2 - x_2 = 0 \end{cases}$$

tiene un conjunto continuo de soluciones de la forma: $x_1 = \pm\sqrt{t}$, $x_2 = t$, con $t > 0$ (ver Figura 4.8 (b)).

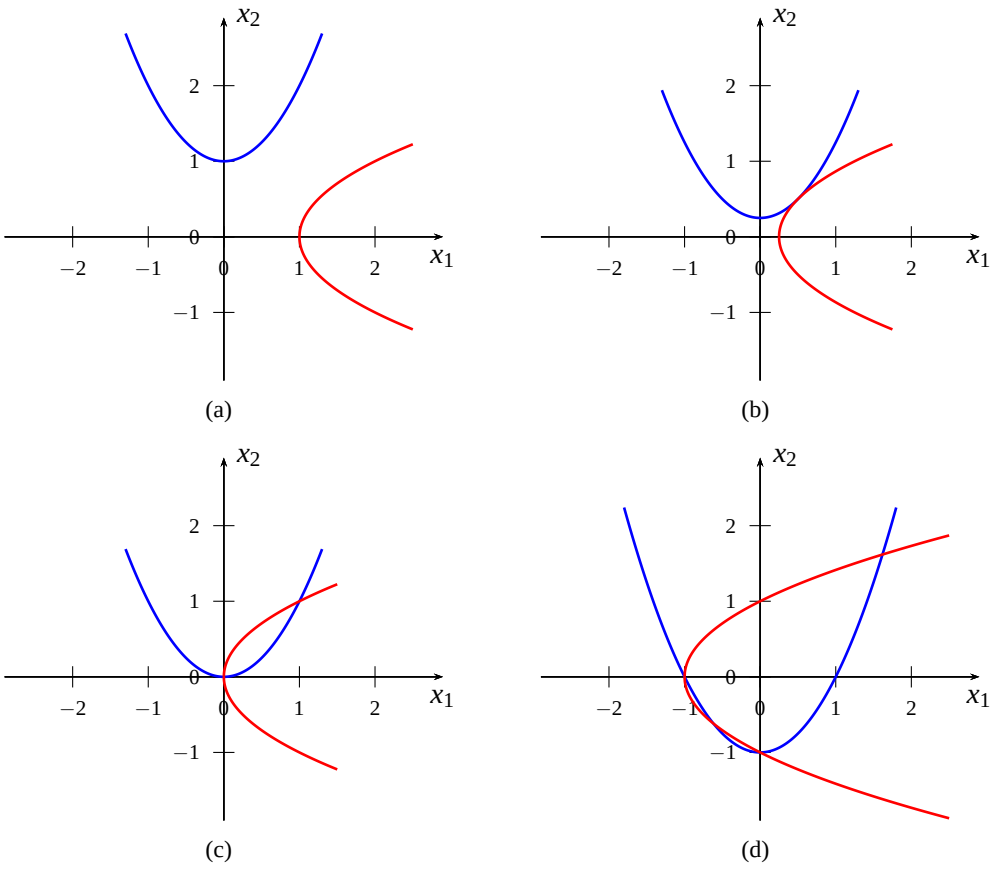


Figura 4.7.

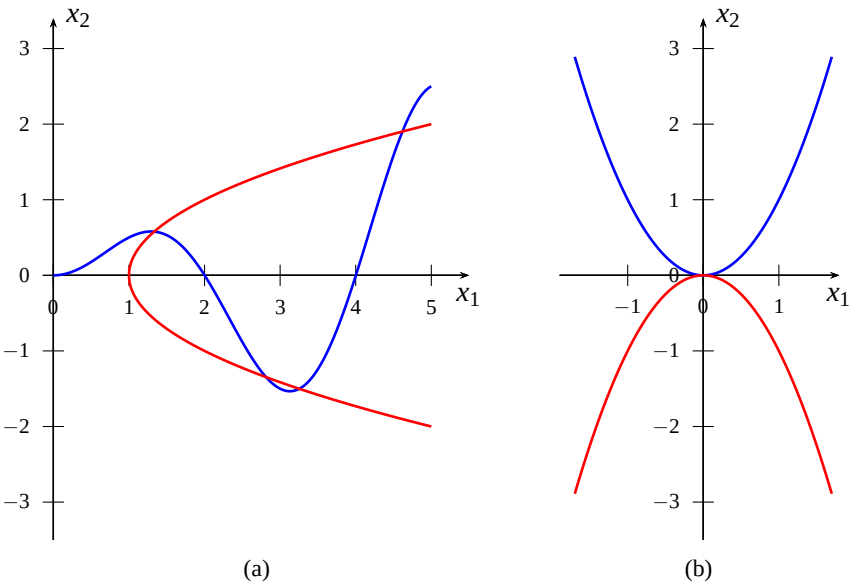


Figura 4.8.

Estos ejemplos muestran la necesidad de conocer algunos de los principales resultados de existencia y unicidad de soluciones para sistemas de ecuaciones no lineales. Un estudio exhaustivo de los teoremas de existencia estaría más allá del alcance de este libro. Se recomienda el libro [21] para tratar la mayoría de los enfoques y resultados principales de existencia y unicidad para sistemas de ecuaciones no lineales.

Asumiendo que el sistema de ecuaciones no lineales (4.17) posee soluciones, nuestro interés es la descripción de los métodos para aproximar estas soluciones. En contraste con el caso de los sistemas de ecuaciones lineales, los métodos directos para la solución de ecuaciones no lineales son aplicables solo para sistemas pequeños de una forma muy especial. En consecuencia, nuestra atención estará restringida a métodos iterativos.

Describiremos métodos iterativos que generan una sucesión de vectores $\{\mathbf{x}_k\}$ que convergen, bajo ciertas condiciones, a una solución de (4.17), para cierta aproximación inicial $\mathbf{x}_0 \in \mathbb{R}^n$. La aproximación inicial es fundamental para el buen funcionamiento del método iterativo, tan importante es que los resultados teóricos establecen convergencia local, es decir, suponen que el iterado inicial está muy cerca de una solución. En otras palabras, generalmente, si no se tiene una buena aproximación inicial, no se puede garantizar una buena solución, salvo que el iterado inicial sea sugerido por la información física del problema.

La forma general de los métodos iterativos es la siguiente: dado un iterado inicial $\mathbf{x}_0 \in \mathbb{R}^n$, se genera $\mathbf{x}_{k+1}$ como

$$\mathbf{x}_{k+1} = \mathbf{x}_k + A_k\mathbf{x}_k, \quad k = 0,1,2,\ldots$$

donde $A_k \in \mathbb{R}^{n\times n}$ es una matriz cuadrada de orden n. La escogencia o definición de esta matriz define un método iterativo. Nos centraremos en los siguientes métodos iterativos: iteración de punto fijo, método de Newton y los métodos Quasi-Newton.

4.2.1 Iteración de Punto Fijo

El enfoque de punto fijo para la resolución de la ecuación no lineal $f(x) = 0$, puede extenderse al caso de sistemas de ecuaciones no lineales. Resolver un sistema de ecuaciones no lineales en su forma vectorial $\mathbf{F}(\mathbf{x}) = 0$, es equivalente a resolver la ecuación

$$\mathbf{x} = \mathbf{G}(\mathbf{x}),$$

donde $\mathbf{G}$ se escoge de manera que $\mathbf{G}(\mathbf{x}^*) = \mathbf{x}^*$ siempre que $\mathbf{F}(\mathbf{x}^*) = 0$. En otras palabras, resolver un sistema de ecuaciones no lineales es equivalente a encontrar un punto fijo de determinada función vectorial. De esta forma, podemos encontrar una solución de la ecuación $\mathbf{x} = \mathbf{G}(\mathbf{x})$ mediante el siguiente proceso iterativo denominado *iteración de punto fijo*: dado $\mathbf{x}_0 \in \mathbb{R}^n$, generar

$$\mathbf{x}_{k+1} = \mathbf{G}(\mathbf{x}_k), \quad k = 0,1,2,\ldots \tag{4.18}$$

Este método iterativo también es llamado *iteración de Richardson*, *iteración de Picard* o *método de sustitución sucesiva*.

Antes de presentar la convergencia de la iteración de punto fijo, daremos dos definiciones.

Definición 4.1. Una función vectorial $\mathbf{G} : \mathbb{R}^n \to \mathbb{R}^n$ se denomina *ℓ-Lipschitz continua* en un conjunto $D \subset \mathbb{R}^n$, si existe una constante $\ell > 0$ tal que

$$\|\mathbf{G}(x) - \mathbf{G}(y)\| \leq \ell\|x - y\|,$$

para todo $x, y \in D$ y una norma vectorial $\|\cdot\|$. Al número ℓ generalmente se le denomina *constante de Lipschitz*.

Definición 4.2. Una función vectorial $\mathbf{G}:\mathbb{R}^n \to \mathbb{R}^n$ se dice *contractiva* en $D \subset \mathbb{R}^n$, si $\mathbf{G}$ es ℓ-Lipschitz continua en D y $\ell < 1$.

En el siguiente teorema se establece la convergencia de la iteración de punto fijo.

Teorema 4.7. *Sea $D \subset \mathbb{R}^n$ un conjunto cerrado y sea $\mathbf{G}:\mathbb{R}^n \to \mathbb{R}^n$ una función contractiva en D, con constante de Lipschitz ℓ, de manera que $\mathbf{G}(x) \in D$, para todo $x \in D$. Entonces, existe un único punto fijo de $\mathbf{G}$, $\mathbf{x}^* \in D$, y la iteración definida por* (4.18) *converge Q-linealmente a $\mathbf{x}^*$, con taza de convergencia igual a ℓ.*

Como la escogencia de $\mathbf{G}$ no es única, existen muchas formas iterativas de punto fijo. Existe una en particular que se denomina *forma iterativa de Jacobi* para sistemas no lineales. Dado el sistema no lineal

$$\mathbf{F}(\mathbf{x}) = \begin{bmatrix} f_1(x_1,x_2,\ldots,x_n) \\ f_2(x_1,x_2,\ldots,x_n) \\ \vdots \\ f_x(x_1,x_2,\ldots,x_n) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

una forma iterativa de Jacobi consiste en generar una iteración de punto fijo (4.18), en donde la función $\mathbf{G}(\mathbf{x}) = (g_1(\mathbf{x}),\ldots,g_n(\mathbf{x}))$ se define de manera que

$$x_i = g_i(\mathbf{x}), \quad \text{para } i = 1,2,\ldots,n;$$

en otras palabras, cada función $g_i(\mathbf{x})$ se obtiene despejando la variable x_i en la ecuación

$$f_i(x_1,x_2,\ldots,x_n) = 0,$$

Veamos un ejemplo.

Ejemplo 4.8. Determinemos una fórmula iterativa de Jacobi para resolver el sistema de ecuaciones no lineales

$$\begin{cases} \ln(x_2 - x_1 + 2 - \pi) + x_2^2 = \pi^2 \\ x_1 x_2 + \operatorname{sen} x_2 = \pi \end{cases}$$

el cual tiene una solución de la forma $\mathbf{x}^* = (1,\pi)^T$. Despejando x_2 en la primera ecuación y x_1 en la segunda ecuación tenemos que

$$\begin{aligned} x_2 &= g_2(x_1,x_2) \equiv \sqrt{\pi^2 - \ln(x_2 - x_1 + 2 - \pi)} \\ x_1 &= g_1(x_1,x_2) \equiv \frac{\pi - \operatorname{sen} x_2}{x_2} \end{aligned}$$

Luego, la función $\mathbf{G}$ está definida como

$$\mathbf{G}(x_1,x_2) = \begin{bmatrix} (\pi - \operatorname{sen} x_2)/x_2 \\ \sqrt{\pi^2 - \ln(x_2 - x_1 + 2 - \pi)} \end{bmatrix}$$

y la fórmula iterativa de Jacobi está dada por:

$$\begin{aligned} x_{k+1} &= \frac{\pi - \operatorname{sen} y_k}{y_k}, \quad k = 0,1,\ldots \\ y_{k+1} &= \sqrt{\pi^2 - \ln(y_k - x_k + 2 - \pi)}, \quad k = 0,1,\ldots \end{aligned}$$

Por otro lado, el Jacobiano de $\mathbf{G}(x,y)$ es:

$$\mathbf{G}'(x,y) = \begin{bmatrix} 0 & -\frac{\cos(y)}{y} - \frac{\pi-\sin(y)}{y^2} \\ -\frac{1}{2\sqrt{\pi^2-\log(y-x-\pi+2)}\,(\pi+x-y-2)} & \frac{1}{2\sqrt{\pi^2-\log(y-x-\pi+2)}\,(\pi+x-y-2)} \end{bmatrix},$$

además,

$$\mathbf{G}'(1,\pi) = \begin{bmatrix} 0 & 0 \\ -\frac{1}{2\pi} & \frac{1}{2\pi} \end{bmatrix} \qquad \Rightarrow \qquad \|\mathbf{G}'(1,\pi)\|_1 = \frac{1}{2\pi} < 1,$$

por lo que $\mathbf{G}$ es una función contractiva en la norma $\|\cdots\|_1$. Así, por el Teorema 4.7, se garantiza que el proceso iterativo $\mathbf{x}_{k+1} = \mathbf{G}(\mathbf{x}_k)$ converge a $\mathbf{x}^* = (1,\pi)^T$, cuando $\mathbf{x}_0$ se escoge muy cercano de $\mathbf{x}^*$.

En el ejemplo anterior se halló una forma iterativa de Jacobi tal que $\|G'(\mathbf{x}^*)\| < 1$. En otras palabras, si $\|G'(\mathbf{x}^*)\| < 1$ y $\mathbf{x}_0$ se toma lo suficientemente cercano a $\mathbf{x}^*$, entonces el proceso iterativo (4.18) converge a $\mathbf{x}^*$. Esto es una condición teórica, ya que en la práctica, en general (salvo situaciones muy particulares), no se conoce el valor de $\mathbf{x}^*$, además, demostrar que $\mathbf{G}$ es contractiva no es una tarea fácil.

4.2.2 Método de Newton

Recordemos que el método de Newton para encontrar un cero de la ecuación $f(x) = 0$ tiene la forma:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, \quad k = 0,1,2,\ldots$$

Ahora, usando el desarrollo de Taylor multidimensional de $\mathbf{F}$, podemos mostrar que el proceso iterativo de Newton anterior adquiere la forma:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{F}'(\mathbf{x}_k)^{-1}\mathbf{F}(\mathbf{x}_k), \quad k = 0,1,2,\ldots \tag{4.19}$$

donde $\mathbf{F}'(\mathbf{x})$ es el Jacobiano de $\mathbf{F}$ evaluado en $\mathbf{x}$. La forma iterativa (4.19) se conoce como *método de Newton*. Para la computación práctica, no se usa directamente (4.19), sino que es preferible la forma iterativa

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k, \quad k = 0,1,2,\ldots \tag{4.20}$$

donde $\mathbf{s}_k$ es la solución del sistema de ecuaciones lineales:

$$\mathbf{F}'(\mathbf{x}_k)\,\mathbf{s}_k = -\mathbf{F}(\mathbf{x}_k). \tag{4.21}$$

En el Algoritmo 4.6 se muestra el pseudocódigo del método de Newton y su implementación se aprecia en el Programa 4.6.

Algoritmo 4.6 (Método de Newton)**.**

Entrada: $\mathbf{F}(\mathbf{x})$ diferenciable tal que $J(\mathbf{x})$ sea no singular, iterado inicial $\mathbf{x}_0$.

1: $k = 0$
2: **while** $\|F(\mathbf{x}_k)\| > 0$ **do**
3: Resuelve el sistema $\mathbf{F}'(\mathbf{x}_k)\,\mathbf{s}_k = -\mathbf{F}(\mathbf{x}_k)$
4: $\mathbf{x}_{k+1} = x_k + \mathbf{s}_k$
5: $k = k+1$
6: **end while**

Programa 4.6. Método de Newton para sistemas de ecuaciones no lineales

```
def newtonv(x0,F,DF,maxiter=100,tol=1e-4):
    x = np.asarray(x0,dtype=float)
    Fx = F(x0)
    k = 0
    while True:
        if(np.linalg.norm(Fx)<=tol):
            print('Terminó satisfactoriamente')
            return x,k
        if (k >= maxiter):
            print('Alcanzó el número máximo de iteraciones')
            return x,k
        s = np.linalg.solve(DF(x),-Fx)
        x = x + s
        Fx = F(x)
        k += 1
```

A continuación damos ciertas condiciones denominadas *hipótesis estándar*, que son necesarias para establecer la convergencia del método de Newton y, además, que emplearemos en las subsiguientes secciones.

Hipótesis 4.1 (Hipótesis Estándar)**.**

1. El sistema de ecuaciones no lineales (4.17) tiene una solución $\mathbf{x}^* \in D \subset \mathbb{R}^n$.
2. El Jacobiano $\mathbf{F}'(\mathbf{x})$ es ℓ-Lipschitz continuo en D.
3. La matriz $\mathbf{F}'(\mathbf{x}^*)$ es no singular.

También denotaremos con $\mathscr{B}(\mathbf{x}^*, r)$ a la bola de centro $\mathbf{x}^*$ y radio $r > 0$,

$$\mathscr{B}(\mathbf{x}^*, r) = \{\mathbf{x} \in \mathbb{R}^n : \|\mathbf{x} - \mathbf{x}^*\| < r\}.$$

En el siguiente teorema se establece que el método de Newton converge localmente con una la velocidad de convergencia Q-cuadráctica.

Teorema 4.8. *Asumamos que la hipótesis estándar es cierta. Entonces existe $\delta > 0$ tal que si $\mathbf{x}_0 \in \mathscr{B}(\mathbf{x}^*, \delta)$, entonces la sucesión $\{\mathbf{x}_k\}$ generada por la iteración de Newton* (4.21) *converge Q-cuadráticamente a $\mathbf{x}^*$.*

En general, el supuesto de que la aproximación inicial esté "lo suficientemente cerca" de la solución ($\mathbf{x}_0 \in \mathscr{B}(\mathbf{x}^*, \delta)$) es artificial en muchas aplicaciones. Sin embargo, existen situaciones en las que la aproximación inicial se puede tomar muy cercana a la raíz. Un ejemplo es la integración implícita de ecuaciones diferenciales ordinarias, donde los iterados iniciales dependen la solución hallada en cada paso de integración. En todo caso, cuando el iterado inicial está lejos de la solución, existen métodos denominados *técnicas de globalización* que al combinarse con el método de Newton garantizan convergencia a la solución. Tales técnicas están más allá del alcance de este libro, por lo que se recomiendan los libros [21] y [16].

Note que cada iteración del método de Newton (4.20) envuelve la solución del sistema de ecuaciones lineales (4.21). Además, en cada iteración se tiene que evaluar n^2 elementos del Jacobiano.

Ejemplo 4.9. Empleemos el método de Newton para resolver el sistema de ecuaciones no lineales

$$\mathbf{F}(\mathbf{x}) = \begin{bmatrix} x_1^2 + 2\,\mathrm{sen}(x_2) + x_3 \\ \cos(x_2) - x_3 - 2 \\ x_1^2 + x_2^2 + x_3^2 - 2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}.$$

Una solución del sistema es $\mathbf{x}^* = (1, 0, -1)$. El Jacobiano de $\mathbf{F}$ es

$$\mathbf{F}'(\mathbf{x}) = \begin{bmatrix} 2x_2 & 2\cos(x_2) & 1 \\ 0 & -\mathrm{sen}(x_2) & -1 \\ 2x_1 & 2x_2 & 2x_3 \end{bmatrix}$$

Inicialmente definamos la función $\mathbf{F}(\mathbf{x})$ y su Jacobiano:

```
>>> def F(x): return np.array([x[0]**2+2*np.sin(x[1])+x[2],
...     np.cos(x[1]) - x[2] -2,x[0]**2 + x[1]**2 + x[2]**2 -2])
>>> def J(x): return np.array([[2*x[0],2*np.cos(x[1]),1],
...     [0,-np.sin(x[1]),-1],[2*x[0],2*x[1],2*x[2]]])
```

Cuando comenzamos con el iterado $\mathbf{x}_0 = (1.1, 0.1, -0.9)$, la función `newtonv`, tomando una tolerancia de `tol=1.0e-8`, arroja el siguiente resultado:

```
>>> x,k = newtonv([1.1,0.1,-0.9],F,J,tol=1.0e-8)
Terminó satisfactoriamente
>>> print('Solución:\n',x,'\nIteraciones:',k)
Solución:
 [ 1.0000000e+00 -3.9053334e-13 -1.0000000e+00]
Iteraciones: 4
```

Es decir, Newton encontró la aproximación

$$\mathbf{x}_k = (1, -3.90536809 \times 10^{-13}, -1)$$

en 4 iteraciones. Los iterados producidos por `newtonv` y el error $\|\mathbf{x}_k - \mathbf{x}^*\|_2$, se muestran en la siguiente tabla.

k	$\mathbf{x}_k$	$\|\mathbf{x}_k - \mathbf{x}^*\|_2$
0	(1.1, 0.1, -0.9)	1.73e-01
1	(1.02167409, -0.02292724, -0.99272359)	3.24e-02
2	(1.00073975e+00, -6.41095951e-04, -9.99751903e-01)	1.01e-03
3	(1.00000071e+00, -5.43521408e-07, -9.99999795e-01)	9.21e-07
4	(1.00000000e+00, -3.90536809e-13, -1.00000000e+00)	7.08e-13

Sin embargo, cuando comenzamos con el iterado $\mathbf{x}_0 = (1,1,0)$, el método de Newton no converge. Se deja como ejercicio para el lector, correr la función `newtonv` con el iterado $\mathbf{x}_0 = (1,1,0)$ para verificar que Newton no converge (tome un máximo de 10000 iteraciones).

El ejemplo anterior muestra el típico comportamiento del método de Newton. Cuando el método trabaja, éste lo hace extremadamente bien, reflejando una convergencia Q-cuadrática. Por otro lado, si no se cuenta con un buen iterado inicial, el proceso completo puede fallar espectacularmente y se puede hacer mucho trabajo sin lograr nada.

El método de Newton ha sido estudiado extensamente dado que sus propiedades teó-ricas lo hacen muy atractivo. Éste tiene convergencia Q-cuadrática y se han establecido teoremas que dicen exactamente que tan cerca se debe tomar el iterado inicial para garantizar convergencia. Desde un punto de vista práctico, existen algunas dificultades. Primero, se debe proveer el Jacobiano el cual requiere las expresiones de n^2 derivadas parciales. Si el Jacobiano no está disponible, se usa la diferenciación numérica pero, como veremos en el Capítulo 5, esto llevará a una pérdida de precisión. Además, en cada paso se requiere la solución de un sistema de ecuaciones lineales y esto puede ser costoso. La principal dificultad del método de Newton es que requiere una buena aproximación inicial. Cuando no se tiene un buen iterado inicial, los iterados pueden comportarse de forma errática y vagar por mucho tiempo en el espacio n-dimensional.

Debido a los inconvenientes prácticos del método de Newton, se han diseñado métodos que tratan de imitar el comportamiento de Newton, pero no requieren el Jacobiano o no necesitan resolver un sistemas de ecuaciones lineales en cada iteración. Tales métodos, por supuesto, no convergen Q-cuadráticamente, pero su comportamiento práctico es muy satisfactorio en algunas aplicaciones. A continuación veremos algunos de esos métodos.

4.2.3 Métodos Quasi-Newton

Los método Quasi-Newton son métodos tipo Newton, es decir, generan una sucesión de vectores $\{\mathbf{x}_k\}$ tales que

$$\mathbf{x}_{k+1} = \mathbf{x}_k - H_k^{-1}\mathbf{F}(\mathbf{x}_k), \quad \text{para } k = 0,1,2,\ldots \tag{4.22}$$

donde la matriz H_k aproxima al Jacobiano $\mathbf{F}'(\mathbf{x}_k)$. Después del cálculo de $\mathbf{x}_{k+1}$, la matriz H_{k+1} se obtiene utilizando H_k. En el Algoritmo 4.7 se describe la forma general de los métodos Quasi-Newton.

Algoritmo 4.7 (Método Quasi-Newton)**.**

Entrada: $\mathbf{x}_0 \in \mathbb{R}^n$, $H_0 \in \mathbb{R}^{n\times n}$.

1: $k = 0$
2: **while** $\|F(\mathbf{x}_k)\| > 0$ **do**
3: Resuelve el sistema $H_k\mathbf{s}_k = -\mathbf{F}(\mathbf{x}_k)$
4: $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$
5: $k = k+1$
6: **end while**

La ventaja de estos métodos es que la resolución del sistema lineal $H_k\mathbf{s}_k = -\mathbf{F}(\mathbf{x}_k)$ resulta a menudo más barata que la resolución del sistema $\mathbf{F}'(\mathbf{x}_k)\mathbf{s}_k = -\mathbf{F}(\mathbf{x}_k)$. Los métodos Quasi-Newton más utilizados son: *Newton modificado*, *Newton con diferencia finita* y *métodos tipo secante*.

Newton Modificado

La iteración de Newton modificado está dada por:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - F'(\mathbf{x}_0)^{-1}F(\mathbf{x}_k), \quad \text{para } k \geq 0. \tag{4.23}$$

En el Algoritmo 4.8 se muestra el esquema general del método de Newton modificado.

Algoritmo 4.8 (Método de Newton Modificado)**.**

Entrada: $\mathbf{x}_0 \in \mathbb{R}^n$.

1: $k = 0$
2: **while** $\|F(\mathbf{x}_k)\| > 0$ **do**
3: Resuelve el sistema $\mathbf{F}'(\mathbf{x}_0)\mathbf{s}_k = -\mathbf{F}(\mathbf{x}_k)$
4: $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$
5: $k = k+1$
6: **end while**

Observe que la forma iterativa de Newton modificado se obtiene de (4.22) colocando $H_k = \mathbf{F}'(\mathbf{x}_0)$, en cada iteración $k \geq 0$. Esto puede ser ventajoso, ya que la matriz inicial $H_0 = \mathbf{F}'(\mathbf{x}_0)$ pudiera tener una estructura especial que permite resolver eficientemente el sistema de ecuaciones lineales $H_k\mathbf{s}_k = -\mathbf{F}(\mathbf{x}_k)$. Pero, mantener fija la matriz H_k en todas las iteraciones, también podría ser una desventaja, ya que la velocidad de convergencia se ve afectada, reduciéndose a Q-lineal. Esto último se establece en el siguiente teorema.

Teorema 4.9. *Asumamos que la hipótesis estándar es cierta. Entonces, existen constantes $K > 0$ y $\delta > 0$ tales que si $\mathbf{x} \in \mathscr{B}(\mathbf{x}^*, \delta)$, la sucesión $\{\mathbf{x}_k\}$ generada por la iteración de Newton modificado* (4.23) *converge Q-linealmente a $\mathbf{x}^*$ y*

$$\|\mathbf{x}_{k+1} - \mathbf{x}^*\| \leq K\|\mathbf{x}_k - \mathbf{x}^*\|, \quad \textit{para } k \geq 0.$$

Método de Newton con Diferencia Finita

Inicialmente definamos una aproximación por diferencias finitas del Jacobiano de $\mathbf{F}(\mathbf{x})$.

Definición 4.3. Sean $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^n$ una función vectorial y $h > 0$. Se define una aproximación del Jacobiano $\mathbf{F}'(x)$ por diferencias finitas en el punto $\mathbf{x} \in \mathbb{R}^n$, a la matriz $H_h(\mathbf{x}) \in \mathbb{R}^{n\times n}$ cuya j-ésima columna está dada por:

$$(H_h(\mathbf{x}))_j = \begin{cases} \dfrac{\mathbf{F}(\mathbf{x} + h\|\mathbf{x}\|\mathbf{e}_j) - \mathbf{F}(\mathbf{x})}{h\|\mathbf{x}\|}, & \mathbf{x} \neq 0, \\ \dfrac{\mathbf{F}(h\mathbf{e}_j) - \mathbf{F}(\mathbf{x})}{h}, & \mathbf{x} = 0, \end{cases} \tag{4.24}$$

donde $\mathbf{e}_j \in \mathbb{R}^n$ es el vector cuya coordenada j-ésima es igual a 1 y las restantes son cero.

Dado $h > 0$ y un iterado inicial $\mathbf{x}_0 \in \mathbb{R}^n$, la iteración de Newton con diferencia finita está dada por:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - H_h(\mathbf{x}_k)^{-1}\mathbf{F}(\mathbf{x}_k), \quad \text{para } k \geq 0, \tag{4.25}$$

donde $H_h(\mathbf{x})$ es la matriz de diferencia finita definida en (4.24). En el Algoritmo 4.9 se muestra el esquema general del método de Newton con diferencia finita.

Algoritmo 4.9 (Método de Newton con Diferencia Finita)**.**

Entrada: $\mathbf{x}_0 \in \mathbb{R}^n$ and $h > 0$.

1: $k = 0$
2: **while** $\|F(\mathbf{x}_k)\| > 0$ **do**
3: Para $j = 1, 2, \ldots, n$, calcula la j-ésima columna de la matriz H_k,

$$(H_k)_j = \begin{cases} \dfrac{\mathbf{F}(\mathbf{x}_k + h\|\mathbf{x}_k\|\mathbf{e}_j) - \mathbf{F}(\mathbf{x}_k)}{h\|\mathbf{x}_k\|}, & \mathbf{x}_k \neq 0, \\ \dfrac{\mathbf{F}(h\mathbf{e}_j) - \mathbf{F}(\mathbf{x}_k)}{h}, & \mathbf{x}_k = 0 \end{cases}$$

4: Resuelve el sistema $H_k\mathbf{s}_k = -\mathbf{F}(\mathbf{x}_k)$
5: $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$
6: $k = k + 1$
7: **end while**

La implementación del Algoritmo 4.9 se muestra en el Programa 4.7. Aquí se usa la aproximación del Jacobiano $\mathbf{F}'(x)$ por diferencias finitas implementada en el Programa 4.8.

En el Algoritmo 4.9, la matriz H_k aproxima al Jacobiano $\mathbf{F}'(\mathbf{x}_k)$, $(H_k)_j$ es la j-ésima columna de H_k y $h > 0$ es pequeño ($h \approx 10^{-8}$). En el Teorema 4.10 se muestra un resultado de convergencia del método de Newton con diferencias finitas.

Programa 4.7. Newton con Diferencias Finitas

```
def newton_df(x0,F,maxiter=100,tol=1e-4):
    x = np.asarray(x0,dtype=float)
    Fx = F(x0)
    k = 0
    while True:
        if(np.linalg.norm(Fx)<=tol):
            print('Terminó satisfactoriamente')
            return x,k
        if (k >= maxiter):
            print('Alcanzó el número máximo de iteraciones')
            return x,k
        Jx = Jacobiano_DF(F,x,Fx)
        s = np.linalg.solve(Jx,-Fx)
        x = x + s
        Fx = F(x)
        k += 1
```

Programa 4.8. Aproximación por Diferencias Finitas del Jacobiano

```
def Jacobiano_DF(F,x,h):
    n = len(x)
    I = np.eye(n)
    H = np.zeros((n,n))
    nx = np.linalg.norm(x)
    Fx = F(x)
    if nx == 0:
        for j in range(n):
            H[:,j] = (F(h*I[:,j])-Fx)/h
    else:
        hnx = h*np.linalg.norm(x)
        for j in range(n):
            H[:,j] = (F(x+hnx*I[:,j])-Fx)/hnx
    return H
```

Teorema 4.10. *Asumamos que la hipótesis estándar es cierta. Sea $h > 0$. Existen constantes positivas K, $\widehat{\varepsilon}$ y δ tales que si $\mathbf{x} \in \mathscr{B}(\mathbf{x}^*, \delta)$, $\|\varepsilon(\mathbf{x})\| < \widehat{\varepsilon}$ para todo $\mathbf{x} \in \mathscr{B}(\mathbf{x}^*, \delta)$, con $\varepsilon(\mathbf{x})$ el error de redondeo de punto-flotante, y existe una constante $M > 0$ tal que $h > M\sqrt{\widehat{\varepsilon}}$, entonces la sucesión $\{\mathbf{x}_k\}$ generada por la iteración de Newton con diferencia finita* (4.25) *satisface*

$$\|\mathbf{x}_{k+1} - \mathbf{x}^*\| \leq K\left(\widehat{\varepsilon} + (\|\mathbf{x}_k - \mathbf{x}^*\| + h)\|\mathbf{x}_k - \mathbf{x}^*\|\right), \quad \textit{para } k \geq 0.$$

Observe que el Teorema 4.10 no garantiza que la iteración de Newton con diferencia finita converja. Incluso, no existe la garantía que el error de redondeo de punto-flotante tienda a cero, esto es, $\varepsilon(x_k) \to 0$ cuando $\mathbf{x}_k \to \mathbf{x}^*$. Lo que puede esperarse es que la sucesión $\{\|\mathbf{x}_k - \mathbf{x}^*\|\}$ tienda a ser constante y decrecer hasta $\|\mathbf{x}_k - \mathbf{x}^*\| \approx \widehat{\varepsilon}$. Por tanto, en las iteraciones iniciales, mientras $\|\mathbf{x}_k - \mathbf{x}^*\| \gg \sqrt{\widehat{\varepsilon}}$, el progreso de la iteración no se distinguirá del método de Newton implementado con un Jacobiano exacto. Cuando, $\|\mathbf{x}_k - \mathbf{x}^*\| \ll \sqrt{\widehat{\varepsilon}}$, el Teorema 4.10 dice que el error $\|\mathbf{x}_{k+1} - \mathbf{x}^*\|$ es de orden $\mathbf{O}(\sqrt{\widehat{\varepsilon}})$. Luego, se verá una convergencia Q-cuadrática hasta que el error en $\mathbf{F}$ no admita una reducción adicional.

Para problemas con constante de Lipschitz de $\mathbf{F}'(\mathbf{x})$ grandes, el error en la aproximación en diferencias finitas del Jacobiano también será grande. Si $\mathbf{F}'(\mathbf{x}^*)$ está mal-condicionado, el comportamiento de la iteración de Newton con diferencia finita será, en general, deficiente.

Métodos Tipo Secante

Los métodos tipo secante generan un iterado $\mathbf{x}_k$ a través de la fórmula iterativa

$$\mathbf{x}_{k+1} = \mathbf{x}_k - A_k F(\mathbf{x}_k), \quad \text{para } k \geq 0, \tag{4.26}$$

donde la matriz $A_k \in \mathbb{R}^{n\times n}$ satisface la *ecuación de la secante*,

$$A_k \mathbf{s}_{k-1} = \mathbf{y}_{k-1}, \tag{4.27}$$

donde $\mathbf{s}_{k-1} = \mathbf{x}_k - \mathbf{x}_{k-1}$ y $\mathbf{y}_{k-1} = \mathbf{F}(\mathbf{x}_k) - \mathbf{F}(\mathbf{x}_{k-1})$. En el Algoritmo 4.10 se muestra el esquema general de los métodos tipo secante.

Algoritmo 4.10 (Método tipo secante)**.**

Entrada: $\mathbf{x}_0 \in \mathbb{R}^n$, $A_0 \in \mathbb{R}^{n\times n}$.

1: $k = 0$
2: **while** $\|F(\mathbf{x}_k)\| > 0$ **do**
3: Resuelve el sistema $A_k\mathbf{s}_k = -\mathbf{F}(\mathbf{x}_k)$
4: $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$
5: $k = k+1$
6: **end while**

Para establecer la convergencia de los métodos tipo secante necesitamos la *condición de Denis-Moré*,

$$\lim_{k\to\infty} \frac{\|E_k s_k\|}{\|\mathbf{s}_k\|} = 0, \tag{4.28}$$

donde $\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$ y $E_k = A_k - \mathbf{F}'(\mathbf{x}_k)$ es el error en la aproximación del Jacobiano.

Teorema 4.11. *Asumamos que la hipótesis estándar es cierta. Sean $x_0 \in \mathbb{R}^n$ la aproximación inicial, $\{A_k\}$ una sucesión de matrices no singulares que satisface* (4.27) *y $\{\mathbf{x}_k\}$ la sucesión generada por iteración tipo secante* (4.26)*. Supongamos que $\mathbf{x}_k \neq \mathbf{x}^*$, para algún k. Entonces, $\mathbf{x}_k \to \mathbf{x}^*$ Q-superlinealmente si, y sólo si la sucesión $\{\mathbf{x}_k\}$ converge a $\mathbf{x}^*$ y la condición de Denis-Moré* (4.28) *es cierta.*

El teorema anterior presenta el resultado general de la convergencia Q-suplerlineal de los métodos tipo secante. Allí se exige que los métodos tipo secante generen sucesiones $\{\mathbf{x}_k\}$ y $\{A_k\}$ de manera que se satisfaga la condición de Denis-Moré (4.28). La manera de construir matrices A_k no singulares que satisfagan la ecuación de la secante (4.27), origina un método tipo secante en particular; además, la demostración de su convergencia se basa en probar que, con tal escogencia de A_k, se satisface la condición de Denis-Moré. Éste último procedimiento requiere de un análisis más especializado que está fuera del alcance de este libro, por ello se recomienda el libro [16] para ver detalles de este proceso.

En la literatura existen diversas maneras de construir la matriz A_k. Describiremos tres de ellas por su gran uso y por su sentido histórico: *método de Broyden*, *método malo de Broyden* y *método de Broyden inverso*.

Método de Broyden

Dados $\mathbf{x}_0 \in \mathbb{R}^n$ y una matriz $A_0 \in \mathbb{R}^{n\times n}$, el método de Broyden calcula cada iterado a través de la siguiente fórmula:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - A_k^{-1}\mathbf{F}(\mathbf{x}_k), \quad \text{para } k \geq 0, \tag{4.29}$$

donde $A_{k+1} \in \mathbb{R}^{n\times n}$ se define por

$$A_{k+1} = A_k + \frac{(\mathbf{y}_k - A_k\mathbf{s}_k)\mathbf{s}_k^T}{\mathbf{s}_k^T\mathbf{s}_k}, \quad \text{para } k \geq 0,$$

donde $\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k +$ y $\mathbf{y}_k = \mathbf{F}(\mathbf{x}_{k+1}) - \mathbf{F}(\mathbf{x}_k)$. La matriz A_{k+1} satisface la ecuación de la secante (4.27). En efecto, se tiene que

$$A_{k+1}\mathbf{s}_k = \left[A_k + \frac{(\mathbf{y}_k - A_k\mathbf{s}_k)\mathbf{s}_k^T}{\mathbf{s}_k^T\mathbf{s}_k}\right]\mathbf{s}_k = A_k\mathbf{s}_k + \frac{(\mathbf{y}_k - A_k\mathbf{s}_k)\mathbf{s}_k^T\mathbf{s}_k}{\mathbf{s}_k^T\mathbf{s}_k} = \mathbf{y}_k.$$

En sí, el método de Broyden "escoge A_{k+1} lo más cercana posible A_k de manera que satisfaga la ecuación de la secante (4.27)". El esquema iterativo del método de Broyden se muestra en el Algoritmo 4.11 y su implementación se muestra en el Programa 4.9.

Algoritmo 4.11 (Método de Broyden)**.**

Entrada: $\mathbf{x}_0 \in \mathbb{R}^n$, $A_0 \in \mathbb{R}^{n\times n}$.

1: $k = 0$
2: **while** $\|F(\mathbf{x}_k)\| > 0$ **do**
3: Resuelve el sistema $A_k\mathbf{s}_k = -\mathbf{F}(\mathbf{x}_k)$
4: $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$
5: $\mathbf{y}_k = \mathbf{F}(\mathbf{x}_{k+1}) - \mathbf{F}(\mathbf{x}_k)$
6: $A_{k+1} = A_k + \dfrac{(\mathbf{y}_k - A_k\mathbf{s}_k)\mathbf{s}_k^T}{\mathbf{s}_k^T\mathbf{s}_k}$
7: $k = k+1$
8: **end while**

Programa 4.9. Método de Broyden

```
def broyden(x0,F,A0,maxiter=100,tol=1e-4):
    x = np.asarray(x0,dtype=float)
    A = np.asarray(A0,dtype=float)
    Fx = F(x0)
    k = 0
    while True:
        if (np.linalg.norm(Fx)<=tol):
            return x,k
        if (k >= maxiter):
            return x,k
        s = np.linalg.solve(A,-Fx)
        x = x + s
        y = Fx
        Fx = F(x)
        y = Fx - y
        A = A + ((y-A.dot(s))*s.reshape((x.shape[0],1)))/(s.dot(s))
        k += 1
```

Método Malo de Broyden

Dados $\mathbf{x}_0 \in \mathbb{R}^n$ y una matriz $A_0 \in \mathbb{R}^{n\times n}$, el método malo de Broyden calcula cada iterado a través de la siguiente fórmula:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - A_k^{-1}\mathbf{F}(\mathbf{x}_k), \quad \text{para } k \geq 0, \tag{4.30}$$

donde $A_{k+1} \in \mathbb{R}^{n\times n}$ se define por

$$A_{k+1} = A_k + \frac{(\mathbf{y}_k - A_k\mathbf{s}_k)\mathbf{y}_k^T}{\mathbf{y}_k^T A_k\mathbf{s}_k}A_k, \quad \text{para } k \geq 0,$$

donde $\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k +$ y $\mathbf{y}_k = \mathbf{F}(\mathbf{x}_{k+1}) - \mathbf{F}(\mathbf{x}_k)$. La matriz A_{k+1} satisface la ecuación de la secante (4.27). En efecto, se tiene que

$$A_{k+1}\mathbf{s}_k = \left[\frac{(\mathbf{y}_k - A_k\mathbf{s}_k)\mathbf{y}_k^T}{\mathbf{y}_k^T A_k\mathbf{s}_k}A_k\right]\mathbf{s}_k = A_k\mathbf{s}_k + \frac{(\mathbf{y}_k - A_k\mathbf{s}_k)}{\mathbf{y}_k^T A_k\mathbf{s}_k}(\mathbf{y}_k^T A_k\mathbf{s}_k) = \mathbf{y}_k.$$

El esquema iterativo del método malo de Broyden se muestra en el Algoritmo 4.12

Algoritmo 4.12 (Método Malo de Broyden)**.**

Entrada: $\mathbf{x}_0 \in \mathbb{R}^n$, $A_0 \in \mathbb{R}^{n\times n}$.

1: $k = 0$
2: **while** $\|F(\mathbf{x}_k)\| > 0$ **do**
3: Resuelve el sistema $A_k\mathbf{s}_k = -\mathbf{F}(\mathbf{x}_k)$
4: $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$
5: $\mathbf{y}_k = \mathbf{F}(\mathbf{x}_{k+1}) - \mathbf{F}(\mathbf{x}_k)$
6: $A_{k+1} = A_k + \dfrac{(\mathbf{y}_k - A_k\mathbf{s}_k)\mathbf{y}_k^T}{\mathbf{y}_k^T A_k \mathbf{s}_k} A_k$
7: $k = k+1$
8: **end while**

Método de Broyden Inverso

Dados $\mathbf{x}_0 \in \mathbb{R}^n$ y una matriz $A_0 \in \mathbb{R}^{n\times n}$, el método de Broyden inverso calcula cada iterado a través de la siguiente fórmula:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - A_k^{-1}\mathbf{F}(\mathbf{x}_k), \quad \text{para } k \geq 0, \tag{4.31}$$

donde la matriz inversa de $A_{k+1} \in \mathbb{R}^{n\times n}$ se define por

$$A_{k+1}^{-1} = A_k^{-1} + \frac{(\mathbf{s}_k - A_k^{-1}\mathbf{y}_k)\mathbf{s}_k^T A_k^{-1}}{\mathbf{s}_k^T A_k^{-1}\mathbf{y}_k}, \quad \text{para } k \geq 0,$$

donde $\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k +$ y $\mathbf{y}_k = \mathbf{F}(\mathbf{x}_{k+1}) - \mathbf{F}(\mathbf{x}_k)$. La matriz A_{k+1} satisface la ecuación de la secante (4.27). En efecto, se tiene que

$$A_{k+1}^{-1}\mathbf{y}_k = \left[A_k^{-1} + \frac{(\mathbf{s}_k - A_k^{-1}\mathbf{y}_k)\mathbf{s}_k^T A_k^{-1}}{\mathbf{s}_k^T A_k^{-1}\mathbf{y}_k}\right]\mathbf{y}_k = A_k^{-1}\mathbf{y}_k + \frac{(\mathbf{s}_k - A_k^{-1}\mathbf{y}_k)}{\mathbf{s}_k^T A_k^{-1}\mathbf{y}_k}(\mathbf{s}_k^T A_k^{-1}\mathbf{y}_k) = \mathbf{s}_k.$$

Multiplicando por A_{k+1} en ambos lados de la ecuación anterior, obtenemos

$$A_{k+1}\mathbf{s}_k = \mathbf{y}_k,$$

es decir, A_{k+1} satisface la ecuación de la secante (4.27). El método de Broyden inverso construye la matriz A_k^{-1} de tal manera que A_k satisfaga la ecuación de la secante (4.27). El esquema iterativo del método de Broyden inverso se muestra en el Algoritmo 4.13

Algoritmo 4.13 (Método de Broyden Inverso)**.**

Entrada: $\mathbf{x}_0 \in \mathbb{R}^n$, $A_0 \in \mathbb{R}^{n\times n}$.

1: $k = 0$
2: **while** $\|F(\mathbf{x}_k)\| > 0$ **do**
3: Resuelve el sistema $A_k\mathbf{s}_k = -\mathbf{F}(\mathbf{x}_k)$
4: $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$
5: $\mathbf{y}_k = \mathbf{F}(\mathbf{x}_{k+1}) - \mathbf{F}(\mathbf{x}_k)$
6: $A_{k+1}^{-1} = A_k^{-1} + \dfrac{(\mathbf{s}_k - A_k^{-1}\mathbf{y}_k)\mathbf{s}_k^T A_k^{-1}}{\mathbf{s}_k^T A_k^{-1}\mathbf{y}_k}$
7: $k = k+1$
8: **end while**

En el siguiente ejemplo aplicamos el método de Broyden al sistema de ecuaciones no lineales del Ejemplo 4.9.

Ejemplo 4.10. Consideremos el sistema de ecuaciones lineales del Ejemplo 4.9. En este caso la función $\mathbf{F}(\mathbf{x})$ y su Jacobiano están dadas por:

```
>>> def F(x): return np.array([x[0]**2+2*np.sin(x[1])+x[2],
...     np.cos(x[1]) - x[2] -2,x[0]**2 + x[1]**2 + x[2]**2 -2])
>>> def J(x): return np.array([[2*x[0],2*np.cos(x[1]),1],
...     [0,-np.sin(x[1]),-1],[2*x[0],2*x[1],2*x[2]]])
```

Cuando comenzamos con el iterado $\mathbf{x}_0 = (1.1, 0.1, -0.9)$, la función `broyden` dada en el Programa 4.9, tomando una tolerancia de `tol=1.0e-8`, arroja el siguiente resultado:

```
>>> x,k = broyden([1.1,0.1,-0.9],F,J([1.1,0.1,-0.9]),tol=1e-8)
Terminó satisfactoriamente
>>> print('Solución:\n',x,'\nIteraciones:',k)
Solución:
 [ 1.00000001e+00 -1.20332120e-08 -9.99999993e-01]
Iteraciones: 9
```

Es decir, Broyden encontró la aproximación

$$\mathbf{x}_k = (1.00000001, -1.2033212 \times 10^{-8}, -9.99999993 \times 10^{-1})$$

en 9 iteraciones. Los iterados producidos por `broyden` y el error $\|\mathbf{x}_k - \mathbf{x}^*\|_2$, se muestran en la siguiente tabla.

k	$\mathbf{x}_k$	$\|\mathbf{x}_k - \mathbf{x}^*\|_2$
0	(1.1, 0.1, -0.9)	1.73e-01
1	(1.02167409, -0.02292724, -0.99272359)	3.24e-02
2	(0.99850181, 0.00328087, -1.00059701)	3.66e-03
3	(1.00038135e+00, -5.73791413e-04, -9.99834999e-01)	7.08e-04
4	(9.99969052e-01, 6.23764764e-05, -1.00000685e+00)	7.00e-05
5	(1.00000796e+00, -1.21588750e-05, -9.99996385e-01)	1.50e-05
6	(9.99999544e-01, 1.03792884e-06, -9.99999949e-01)	1.13e-06
7	(1.00000021e+00, -2.99472606e-07, -9.99999884e-01)	3.83e-07
8	(1.00000001e+00, -2.79842526e-09, -9.99999980e-01)	2.30e-08
9	(1.00000001e+00, -1.20332120e-08, -9.99999993e-01)	1.69e-08

Por el contrario, al comenzar con el iterado $\mathbf{x}_0 = (1,1,0)$, el método de Broyden no converge. Se deja como ejercicio para el lector, correr la función `broyden` con el iterado $\mathbf{x}_0 = (1,1,0)$ para verificar que Broyden no converge. Note que Broyden converge más lentamente que Newton (ver Ejemplo 4.9).

El módulo `Optimize` de la librería `Scipy` de Python, provee diferentes funciones para encontrar una solución de la ecuación $f(x) = 0$, tanto para el caso escalar como para el caso vectorial. Para conocer las propiedades de cada una de estas funciones se recomienda el manual "SciPy Reference Guide", disponible en https://docs.scipy.org/doc/. Nosotros centraremos nuestra atención en la función `fsolve`,

Se deja como ejercicio para el lector, investigar que métodos utiliza internamente `fsolve`.

```
fsolve(f,x0,args=(),fprime=None,full_output=0,col_deriv=0,xtol=1.49012e-08,
       maxfev=0,band=None,psfcn=None,actor=100,diag=None)
```

la cual encuentra un cero de una función `f` (escalar o vectorial). Note que `fsolve` tiene diferentes parámetros, muchos de los cuales son opcionales y cuyo propósito escapa del alcance de este libro. A continuación listamos los parámetros de `fsolve` que debemos tener presente:

Parámetros

`f`: función
: Función definida en Python que devuelve un número o un arreglo.

`x0`: arreglo
: Aproximación inicial de la raíz de la ecuación `f(x)=0`.

`args`: n-upla, opcional
: Contiene argumentos extras para la función `f`.

`fprime`: función, opcional
: Una función para calcular el Jacobiano de `f`. Si éste is `None` (default), entonces el Jacobiano será estimado por diferencias finita.

`full_output`: lógica, opcional
: Si `full_output` es `False`, se arroja la raíz. Si `full_output` es `True`, se arroja salidas adicionales.

`xtol`: número positivo, opcional
: El cálculo finalizará si el error relativo entre dos iteraciones consecutivas es como máximo `xtol`.

La función `fsolve` arroja las siguientes variables como salida:

Salida

`x`: arreglo
: La solución.

`infodict`: dict
: Un diccionario de salidas opcionales con los elementos:
  - `nfev` número de llamadas de la función.
  - `njev` número de llamadas del Jacobiano.
  - `fvec` evaluación de la función en la salida.

`ier`: entero
: Un indicador entero. Toma el valor de 1 si la solución fue encontrada, de lo contrario refiere a `mesg` para mayor información.

`mesg`: string
: Si la solución no es encontrada, `mesg` muestra detalles de la causa de la falla.

Seguidamente describiremos con ejemplos el uso de la función `fsolve`.

Ejemplo 4.11. • Empleemos `fsolve` para calcular el cero de $f(x) = x|\cos(x)| - 1$, tomando a $x_0 = 2$ como iterado inicial.

```
>>> from scipy.optimize import fsolve
>>> def f(x): return x*abs(np.cos(x))-1
>>> fsolve(f,2)
array([ 2.07393281])
```

- Empleemos `fsolve` para calcular el cero de $f(x) = \cos^2(2x) - x^2$, utilizando $x_0 = 0.2$ como iterado inicial y la derivada $f'(x) = -4\cos(2x)\sin(2x) - 2x$.

```
>>> def f(x): return np.cos(2*x)**2-x**2
>>> def df(x): return -4*np.cos(2*x)*np.sin(2*x)-2*x
>>> fsolve(f,0.2,fprime=df)
array([ 0.51493326])
```

- Calculemos el cero de la función $f(x)$ anterior sin introducir la derivada de $f(x)$.

```
>>> fsolve(f,0.2)
array([ 0.51493326])
```

- La función `fsolve` también puede emplearse para resolver sistemas de ecuaciones no lineales. Consideremos el sistema de ecuaciones no lineales del Ejemplo 4.9,

$$\mathbf{F}(\mathbf{x}) = \begin{bmatrix} x_1^2 + 2\,\text{sen}(x_2) + x_3 \\ \cos(x_2) - x_3 - 2 \\ x_1^2 + x_2^2 + x_3^2 - 2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}.$$

El Jacobiano de $\mathbf{F}(\mathbf{x})$ es:

$$\mathbf{F}'(\mathbf{x}) = \begin{bmatrix} 2x_2 & 2\cos(x_2) & 1 \\ 0 & -\,\text{sen}(x_2) & -1 \\ 2x_1 & 2x_2 & 2x_3 \end{bmatrix}$$

Resolvamos el sistema no lineal usando la función `fsolve`. Para ello, tomemos el iterado inicial $\mathbf{x}_0 = (1.1, 0.1, -0.9)$ y una tolerancia de `xtol=1.0e-9`; además, coloquemos `full_output = True`, para obtener información adicional en la salida.

```
>>> def F(x): return np.array([x[0]**2+2*np.sin(x[1])+x[2],
...     np.cos(x[1]) - x[2] -2,x[0]**2 + x[1]**2 + x[2]**2 -2])
>>> def J(x): return np.array([[2*x[0],2*np.cos(x[1]),1],
...     [0,-np.sin(x[1]),-1],[2*x[0],2*x[1],2*x[2]]])
>>> sal=fsolve(F,[1.1,0.1,-0.9],fprime=J,xtol=1.0e-9,full_output=True)
>>> print('Solución:',sal[0])
Solución: [  1.00000000e+00   5.39482137e-14  -1.00000000e+00]
>>> print('Número de evaluaciones de la función:',sal[1]['nfev'])
Número de evaluaciones de la función: 9
>>> print('Número de evaluaciones del Jacobiano:',sal[1]['njev'])
Número de evaluaciones del Jacobiano: 1
>>> print('Evaluación de la función en la solución:\n ',sal[1]['fvec'])
Evaluación de la función en la solución:
  [  1.77635684e-15   1.15463195e-14  -7.21644966e-14]
>>> print('ier:',sal[2])
ier: 1
>>> print('mesg:',sal[3])
mesg: The solution converged.
```

Es decir, `fsolve` encontró la aproximación

$$\mathbf{x}^* = (1.00000000, 5.39482137 \times 10^{-14}, -1.00000000)$$

empleando 9 evaluaciones de la función y una evaluación del Jacobiano; además,

$$\mathbf{F}(\mathbf{x}^*) = (1.77635684 \times 10^{-15}, 1.15463195 \times 10^{-14}, -7.21644966 \times 10^{-14}).$$

Ahora bien, `fsolve` también puede resolver el sistema sin usar el Jacobiano. Veamos como lo hace.

```
>>> sal=fsolve(F,[1.1,0.1,-0.9],xtol=1.0e-9,full_output=True)
>>> print('Solución:',sal[0])
Solución: [  1.00000000e+00   5.39572173e-14  -1.00000000e+00]
>>> print('Número de evaluaciones de la función:',sal[1]['nfev'])
Número de evaluaciones de la función: 12
>>> print('Evaluación de la función en la solución:\n ',sal[1]['fvec'])
Evaluación de la función en la solución:
  [  1.77635684e-15   1.15463195e-14  -7.21644966e-14]
>>> print('ier:',sal[2])
ier: 1
>>> print('mesg:',sal[3])
mesg: The solution converged.
```

Observe que `fsolve` con y sin derivada encuentra la misma solución, pero necesitó más evaluaciones de la función cuando no usó la derivada.

Por otra parte, `fsolve` puede fallar resolviendo algún sistema de ecuaciones no lineales; por ejemplo, corramos la función `fsolve` con el sistema de ecuaciones no lineales anterior y el iterado inicial $\mathbf{x}_0 = (1,1,0)$.

```
>>> sal=fsolve(F,[1,1,0],fprime=J,xtol=1.0e-9,full_output=True)
>>> print('Solución:',sal[0])
Solución: [  3.09584971e-06   6.76338925e-01  -1.24053096e+00]
>>> print('Número de evaluaciones de la función:',sal[1]['nfev'])
Número de evaluaciones de la función: 28
>>> print('Número de evaluaciones del Jacobiano:',sal[1]['njev'])
Número de evaluaciones del Jacobiano: 7
>>> print('ier:',sal[2])
ier: 5
>>> print('mesg:',sal[3])
mesg: The iteration is not making good progress, as measured by the
  improvement from the last ten iterations.
```

En este caso, `fsolve` no halla una solución, sino que arroja los resultados después de 10 iteraciones, indicando que no observa ningún progreso y por ello detuvo la ejecución.

4.3 Aplicaciones

4.3.1 Diseño de Líneas de Transmisión

Esta aplicación consiste en el diseño de las líneas de losa acopladas por aire, para valores dados de las impedancias características Z_{0_e} y Z_{0_o}. La sección transversal de estas líneas de transmisión sin pérdida se muestra en la Figura 4.9. (Ejemplo tomado de [27, pp. 58–62].)

El conductor externo de estas líneas está formado por dos planos de conducción equipotenciales paralelos. Las líneas de este tipo se utilizan ampliamente en diversos dispositivos de microondas, como diferentes tipos de filtros, acopladores direccionales, etc., destinados principalmente a trabajar en niveles de potencia media y alta. Los parámetros eléctricos básicos de las líneas acopladas son las dos impedancias características definidas para las excitaciones simétricas (en fase) y antisimétricas, indicadas por Z_{0_e} y Z_{0_o}, respectivamente.

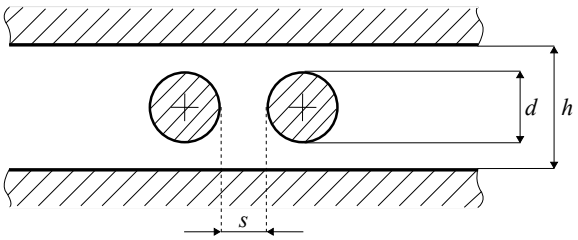


Figura 4.9.

Las impedancias características Z_{0_e} y Z_{0_o} de las líneas consideradas dependen de las dimensiones geométricas d, h, s, ver Figura 4.9. Bajo ciertas consideraciones físicas (ver [27] para detalles), las impedancias características se pueden calcular por las siguientes fórmulas:

$$\begin{aligned} Z_{0_e}(x,y) &= 59.952 \ln\left[\frac{0.523962}{q_1(x)q_2(x,y)q_3(x,y)}\right] \ \Omega \\ Z_{0_o}(x,y) &= 59.952 \ln\left[\frac{0.523962\, q_3(x,y)}{q_1(x)q_4(x,y)}\right] \ \Omega \end{aligned} \tag{4.32}$$

donde:

$$\begin{aligned} x &= d/h, \quad y = s/h, \\ q_1(x) &= x\frac{a(x)}{b(x)}, \\ q_2(x,y) &= \begin{cases} c(y) - x\,d(y) + e(x)g(y), & \text{para } y < 0.9 \\ 1 + 0.004\exp(0.9-y), & \text{para } y \geq 0.9, \end{cases} \\ q_3(x,y) &= \tanh\left(\pi\,\frac{x+y}{2}\right), \\ q_4(x,y) &= \begin{cases} k(y) - x\,l(y) + m(x)n(y), & \text{para } y < 0.9 \\ 1, & \text{para } y \geq 0.9, \end{cases} \\ a(x) &= 1 + \exp(16x - 18.272), \\ b(x) &= \sqrt{5.905 - x^4}, \\ c(y) &= -0.8107y^3 + 1.3401y^2 - 0.6929y + 1.0892 + \frac{0.014002}{y} - \frac{0.000636}{y^2}, \\ d(y) &= 0.11 - 0.83y + 1.64y^2 - y^3, \\ e(x) &= -0.15\exp(-13x), \\ g(y) &= 2.23\exp\left(-7.01y + 10.24y^2 - 27.58y^3\right), \\ k(y) &= 1 + 0.01\left(-0.0726 - \frac{0.2145}{y} + \frac{0.222573}{y^2} - \frac{0.012823}{y^3}\right), \\ l(y) &= 0.01\left(-0.26 + \frac{0.6866}{y} + \frac{0.0831}{y^2} - \frac{0.0076}{y^3}\right), \\ m(x) &= -0.1098 + 1.2138x - 2.2535x^2 + 1.1313x^3, \\ n(y) &= -0.019 - \frac{0.016}{y} + \frac{0.0362}{y^2} - \frac{0.00234}{y^3}. \end{aligned}$$

Las ecuaciones dadas en (4.32) garantizan la exactitud de las impedancias $Z_{0_e}(x,y)$ y $Z_{0_o}(x,y)$ no más bajo del 70%, para $0.1 \leq x \leq 0.8$ e $y \geq 0.1$ ([27]).

Para Z'_{0_e} y Z'_{0_o}, valores dados de las impedancias, el diseño de las líneas de transmisión consiste en determinar los valores de los parámetros $x = d/h$ e $y = s/h$, ver la Figura 4.9, para lo cual se cumplen las siguientes ecuaciones:

$$Z_{0_e}(x,y) - Z'_{0_e} = 0 \qquad \text{y} \qquad Z_{0_o}(x,y) - Z'_{0_o} = 0;$$

Se deja como ejercicio para el lector determinar la expresión matemática del Jacobiano de la función $\mathbf{F}(x,y)$ dada en (4.33)

en otras palabras, se debe resolver el sistema de ecuaciones no lineales

$$\mathbf{F}(x,y) = \begin{bmatrix} Z_{0_e}(x,y) - Z'_{0_e} \\ Z_{0_o}(x,y) - Z'_{0_o} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{4.33}$$

donde $Z_{0_e}(x,y)$ y $Z_{0_o}(x,y)$ están dadas en (4.32).

El sistema de ecuaciones no lineales (4.33) puede resolverse usando el método de Newton, pero la expresión del Jacobiano de $\mathbf{F}$ no es sencilla de determinar. En este caso, es recomendable el uso del método de Newton con diferencias finitas. A modo de comparación, usamos el método de Newton con diferencias finitas del Programa 4.7 y la función `fsolve` de la librería `Scipy`, para resolver el sistema (4.33). Tomamos $\mathbf{x}_0 = (0.1, 0.1)$ como iterado inicial y una tolerancia de 10^{-6}. Con la ejecución del Programa 4.10 se hallan los valores de los parámetros x e y, usando las funciones `fsolve` y `newton_df`, para cinco valores distintos de las impedancias Z'_{0_e} y Z'_{0_o}. Los resultados obtenidos se muestran en las Tablas () y ().

Programa 4.10. Diseño de Líneas de Transmisión

```
"""
Diseño de líneas de transmisión
"""

## Carga de librerías necesarias
import numpy as np
import calnum as ca
import matplotlib.pyplot as plt
from scipy.optimize import fsolve

## Sistema de ecuaciones no lineales
def Sistema(xx,Z0e,Z0o):
    F = np.zeros(2)
    x = xx[0]
    y = xx[1]
    a = 1 + np.exp(16*x - 18.272)
    b = np.sqrt(5.905-x**4)
    c = -0.8107*y**3+1.3401*y**2-0.6929*y+1.0892+(0.014002/y)
    -(0.000636/y**2)
    d = 0.11-0.83*y+1.64*y**2-y**3
    e = -0.15*np.exp(-13*x)
    g = 2.23*np.exp(-7.01*y+10.24*y**2-27.58*y**3)
    k = 1+0.01*(-0.0726-(0.2145/y)+(0.222573/y**2)-(0.012823/y**3))
    l = 0.01*(-0.26+(0.6866/y)+(0.0831/y**2)-(0.0076/y**3))
    m = -0.1098+1.2138*x-2.2535*x**2+1.1313*x**3
    n = -0.019-(0.016/y)+(0.0362/y**2)-(0.00234/y**3)
    q1 = x*a/b
    if y < 0.9:
        q2 = c - x*d + e*g
        q4 = k - x*l + m*n
```

```
    else:
        q2 = 1 + 0.004*np.exp(0.9-y)
        q4 = 1
    q3 = np.tanh(np.pi*(x+y)/2)
    Ze = 59.952*np.log(0.523962/(q1*q2*q3))
    Zo = 59.952*np.log(0.523962*q3/(q1*q4))
    F[0] = Ze - Z0e
    F[1] = Zo - Z0o
    return F

## Distintos valores de las impedancias Z0e y Z0o
Z0e = [51.607, 52.895, 55.277, 59.845, 69.371]
Z0o = [48.443, 47.263, 45.227, 41.774, 36.038]

## Iterado inicial:
xx0 = [0.1, 0.1]

## Tolerancia
tol = 1e-6

## Solución del sistema y generación de resultados
encab ="""
_____________________________________________________
Z0e     Z0o     x         y         nfev   ||F(x,y)||
_____________________________________________________"""
pie="""
_____________________________________________________"""
tex_f = 'Resultados de fsolve'+encab+'\n'
tex_n = '\nResultados de Newton con Diferencias Finitas'+encab+'\n'
for i in range(len(Z0e)):
    ## Función F(x) para Z0e y Z0o dados
    F  = lambda xx: Sistema(xx,Z0e[i],Z0o[i])

    ## Resolución con fsolve
    sal = fsolve(F,xx0,xtol=tol,full_output=True)
    sol = sal[0]
    nfev = sal[1]['nfev']
    NF = np.linalg.norm(sal[1]['fvec'])
    texto = '{0:1.3f}{1:8.3f}{2:10.6f}{3:10.6f}{4:5.0f}
             {5:12.3e}'.format(Z0e[i],Z0o[i],sol[0],sol[1],nfev,NF)
    tex_f = tex_f + texto

    ## Resolución con Newton con Diferencias Finitas
    sol,k = ca.newton_df(xx0,F,tol=tol)
    nfev = (len(xx0)+1)*k
    NF = np.linalg.norm(F(sol))
    texto = '{0:1.3f}{1:8.3f}{2:10.6f}{3:10.6f}{4:5.0f}
             {5:12.3e}'.format(Z0e[i],Z0o[i],sol[0],sol[1],nfev,NF)
    tex_n = tex_n + texto

    if i < len(Z0e)-1:
        tex_f = tex_f+'\n'
        tex_n = tex_n+'\n'

tex_f = tex_f + pie
tex_n = tex_n + pie
print(tex_f)
print(tex_n)
```

Tabla 4.6. Resultados de `fsolve` para el sistema (4.33)

Z'_{0_e}	Z'_{0_o}	$x = d/h$	$y = s/h$	nfev	$\|\mathbf{F}(x,y)\|$
51.607	48.443	0.548031	0.784771	19	7.718e-12
52.895	47.263	0.545476	0.607225	20	1.139e-10
55.277	45.227	0.540853	0.439607	17	1.078e-07
59.845	41.774	0.525972	0.275383	18	1.244e-09
69.371	36.038	0.480362	0.141765	15	1.017e-08

Tabla 4.7. Resultados de `newton_df` para el sistema (4.33)

Z'_{0_e}	Z'_{0_o}	$x = d/h$	$y = s/h$	nfev	$\|\mathbf{F}(x,y)\|$
51.607	48.443	0.548031	0.784771	24	7.512e-09
52.895	47.263	0.545476	0.607225	21	1.285e-08
55.277	45.227	0.540853	0.439607	21	5.759e-13
59.845	41.774	0.525972	0.275383	18	4.796e-09
69.371	36.038	0.480362	0.141765	15	1.872e-07

En las Tablas 4.6 y 4.7 se reportan las soluciones halladas de los parámetros x e y, el número de evaluaciones de la función $\mathbf{F}(x,y)$ y la norma del residual $\|\mathbf{F}(x,y)\|$ en la solución hallada por el método. Note que `fsolve` y `newton_df` hallan, prácticamente, las mismas soluciones para todos los valores de las impedancias, pero `fsolve` es más eficiente, ya que requirió menos evaluaciones de la función para hallar cada solución.

Cabe mencionar que en [27] toman el iterado inicial, $\mathbf{x}_0 = (x_0, y_0)$, donde

$$x_0 = \frac{4}{\pi} \exp\left(\frac{-Z_0}{59.952\sqrt{0.987 - 0.171\alpha - 1.723\alpha^3}} \right),$$

$$y_0 = \frac{1}{\pi} \ln\left(\frac{r+1}{r-1} \right) - x_0,$$

con

$$Z_0 = \sqrt{Z'_{0_e} Z'_{0_o}}, \qquad \alpha = \frac{Z'_{0_e} - Z'_{0_o}}{Z'_{0_e} + Z'_{0_o}}, \quad r = \left(\frac{4}{\pi x_0} \right)^{0.001 + 1.117\alpha - 0.683\alpha^2}.$$

Este iterado inicial (muy particular) está muy cerca de la solución. Un ejercicio interesante es generar las Tablas 4.6 y 4.7 utilizando este iterado inicial.

4.4 Ejercicios

1. Halle un intervalo $[a,b]$ de manera que la ecuación $f(x) = 0$ posea una única solución para cada una de las siguientes funciones $f(x)$.

a) $f(x) = e^x - 2 - x$

b) $f(x) = \cos(x) + 1 - x$

c) $f(x) = \ln(x) - 5 + x$

d) $f(x) = x^2 - 10x + 23$

2. Sea $\{a_k\}$ la sucesión generada por el método de bisección. Dé un ejemplo para el cual $a = a_0 < a_1 = a_2 < a_3 = a_4 < a_5 = a_6 < a_7 = \cdots$.

3. Dé una fórmula que involucre a $(b-a)$ y ε para calcular el número de iteraciones que debe realizar el método de bisección y que garantice $|x^\star - c_k| < \varepsilon$, donde $x^\star \in [a,b]$ es el único cero de la ecuación $f(x) = 0$.

4. Aplique el método de bisección a una función continua para obtener intervalos $[a_0,b_0]$, $[a_1,b_1]$, $[a_2,b_2]$, y así sucesivamente. Sea $x^\star = \lim_{k\to\infty} a_k$. ¿Cuáles de los siguientes enunciados son falsos?

a) $a_0 \le a_1 \le a_2 \le \cdots$.

b) $|x^\star - 2^{-1}(a_k + b_k)| \le 2^{-k}(b_0 - a_0)$, para $k \ge 0$.

c) $[a_{k+1}, b_{k+1}] \subset [a_k, b_k]$, para $k \ge 0$.

d) $|x^\star - c_k| < |x^\star - c_{k-1}|$, para $k \ge 1$.

5. Encuentre el punto inicial positivo más pequeño para el cual el método de Newton diverge, usando la función $f(x) = \tan^{-1} x$.

6. Si el método de Newton se aplica a $f(x) = x^3 - 2$ comenzando en $x_0 = 1$, ¿cuál es el valor de x_2?

7. ¿Cuál es el propósito de la siguiente fórmula de iteración?

$$x_{k+1} = 2x_k - x_k^2.$$

Identifíquela como la iteración de Newton para cierta función.

8. Si se aplica el método de Newton a la ecuación no lineal $f(x) = x^2 - 1 = 0$, con $x_0 = 10^8$, ¿cuántas iteraciones se requerirían para obtener una raíz con un error de 10^{-8}? (Encuentre la respuesta analíticamente, no experimentalmente.)

9. Si se utiliza el método de la secante a la ecuación no lineal $f(x) = x^2 - 2 = 0$, con $x_0 = 0$ y $x_1 = 1$, ¿cuál es el valor de x_2?

10. ¿Cuánto vale x_2 si $x_0 = 1$, $x_1 = 2$, $f(x_0) = 2$, y $f(x_1) = -1$ en una aplicación del método de la secante?

11. Sean $f \in C^2[a,b]$ y $x^\star \in [a,b]$ un cero de $f(x)$, tales que $|f''(x^\star)/f'(x^\star)| = 2$. Suponga también que la magnitud del k-ésimo error del método de la secante es 0.5, es decir, $|e_k| = |x_k - x^\star| = 0.5$. ¿Cuántas iteraciones (a partir de la k-ésima) son necesarias para que el método de la secante reduzca el error a 10^{-15}?

12. Determine los valores de las constantes $a \ge 0$ y $b \ge 0$ para que cada una de las siguientes funciones $g(x)$ posea un único punto fijo en el intervalo dado.

a) $g(x) = a - bx^2$ en $[0,1]$

b) $g(x) = a + 2^{-bx}$ en $[0,1]$

c) $g(x) = a/(x+b)$ en $[0.5,2]$

13. Considere el proceso de iteración de punto fijo con $g(x) = -0.0001x^2 + x$ y $x_0 = 1$.

a) Pruebe que $x_0 > x_1 > \cdots > x_k > x_{k+1} > \cdots$

b) Pruebe que $x_k > 0$ para todo n.

c) Calcule $\lim_{k\to\infty} x_k$.

14. Considere el proceso de iteración de punto fijo con $g(x) = x/2$. Halle el valor de $|x_{k+1} - x_k|/|x_{k+1}|$.

15. Si el proceso de iteración de punto fijo se aplica a la función $g(x) = 2 + (x-2)^4$, empezando por $x_0 = 2.5$, ¿cuál es el orden de convergencia? Encuentre el intervalo de valores iniciales para el cual converge esta iteración de punto fijo. Dése cuenta de que 2 es un punto fijo.

16. Use iteraciones de punto fijo para calcular $\sqrt{2}$ 2 con una precisión de 10^{-4}.

17. Estudie la convergencia de las iteraciones de punto fijo para encontrar el cero $x = 2$ de $f(x) = x3.7x + 6$, con diversas funciones $g(x)$. Represente las iteraciones de punto fijo gráficamente en cada caso.

18. Exprese la iteración de Newton para resolver cada uno de los siguientes sistemas de ecuaciones no lineales.

a) $\begin{cases} x_1^2 + x_2^2 &= 0 \\ x_1^2 - x_2 &= 0 \end{cases}$

b) $\begin{cases} x_1^2 + x_1 x_2^3 &= 9 \\ 3x_1^2 x_2 - x_2^3 &= 4 \end{cases}$

c) $\begin{cases} x_1 + x_2 - 2x_1 x_2 &= 0 \\ 3x_1^2 + x_2^2 - 2x_1 + 2x_2 &= -1 \end{cases}$

d) $\begin{cases} x_1^3 - x_2^2 &= 0 \\ x_1 - x_1^2 x_2 &= 2 \end{cases}$

e) $\begin{cases} 2\operatorname{sen}(x_1) + \cos(x_2) - 5x_1 &= 0 \\ 4\cos(x_1) + 2\operatorname{sen}(x_2) - 5x_2 &= 0 \end{cases}$

Problemas Computacionales

19. Aplique el método de la secante a $f(x) = x3 - 3x + 2$ para aproximar la raíz $x = 1$ con $x_0 = 0$ y $x_1 = 2$. Ilustre que el factor de convergencia es aproximadamente 1.62.

20. La *ecuación de Kepler*,

$$M = E - e\sin(E)$$

relaciona la anomalía media M con la anomalía excéntrica E de una órbita elíptica de excentricidad e, donde $0 < e < 1$.

a) Pruebe que la iteración del punto-fijo usando la función de iteración

$$g(E) = M + e\operatorname{sen}(E)$$

es localmente convergente.

b) Use el esquema de iteración de punto-fijo de la parte a) para resolver la ecuación de Kepler para la anomalía excéntrica E correspondiente a una anomalía media de $M = 1$ (radianes) y una excentricidad de $e = 0.0167$ (excentricidad de la Tierra).

c) Para resolver el mismo problema dado en b), use:

- Método de bisección,
- Método de la secante,
- Método de Newton,
- La función `fsolve`.

Para escoger iterados iniciales apropiados o un intervalo apropiado para el método de bisección, grafique la función $f(E) = M + e\,\mathrm{sin}(E) - E$.

21. La distancia vertical y que un paracaidista cae antes de abrir el paracaídas está dada por la ecuación

$$y = \frac{\ln(\cosh(t\sqrt{gk}))}{k},$$

donde t es el tiempo transcurrido en segundos, $g = 9.8065\ \mathrm{m/s^2}$ es la aceleración debida a la gravedad, y $k = 0.00341\ \mathrm{m^{-1}}$ es una constante relacionada con la resistencia del aire. Escoja un método iterativo para encontrar ceros de funciones y utilícelo convenientemente para determinar el tiempo transcurrido antes de abrir el paracaídas para caer una distancia de 1 km.

22. La *ecuación de estado de Van der Waals*,

$$\left(p + \frac{\alpha}{v^2}\right)(v - \beta) = RT,$$

donde p es la presión relativa, v es el volumen específico, T es la temperatura, R es una constante universal y α y β son constantes que dependen del gas en particular. En unidades apropiadas, $R = 0.082054$, y para el dióxido de carbono, $\alpha = 3.592$ y $\beta = 0.04267$. Escoja un método iterativo para encontrar ceros de funciones y utilícelo convenientemente para calcular el volumen específico v para una temperatura de 300 K y para presiones de 1 atm, 10 atm y 100 atm. Compare sus resultados con los de la ley del gas ideal, $pv = RT$. Esto último se puede usar como una estimación inicial para un método iterativo para resolver la ecuación de van der Waals.

23. Utilizando el método de Newton, resuelva numéricamente cada uno de los siguientes sistemas de ecuaciones no lineales. En cada caso escoja un iterado inicial $\mathbf{x}_0$.

a) $$\begin{cases} x_1 + 2x_2^2 &= 1 \\ |x_1| - x_2^2 &= 0 \end{cases}$$

b) $$\begin{cases} x_1^2 + 2\,\mathrm{sen}(x_2) + x_3 &= 0 \\ \cos(x_2) - x_3 &= 2 \\ x_1^2 + x_2^2 + x_3^2 &= 2 \end{cases}$$

c) $$\begin{cases} x_1 + 10x_2 &= 0 \\ x_3 - \sqrt{|x_4|} &= 0 \\ (x_2 - 2x_3)^2 - 1 &= 0 \\ x_1 - x_4 = 0 \end{cases}$$

d) $\mathbf{F}(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_n(\mathbf{x})) = 0$, donde

$$\begin{aligned} f_i(\mathbf{x}) &= \ln(x_i+1) - \frac{x_i}{n}, \qquad i = 1,2,\ldots,n, \\ \mathbf{x} &= (x_1, x_2, \ldots, x_n). \end{aligned}$$

Para este sistema tome $n = 10$ y el iterado inicial $\mathbf{x}_0 = (1,1,\ldots,1)$.

24. Usando el método de Newton y el método de Broyden, encuentre una solución del siguiente sistema de ecuaciones no lineales:

$$\begin{cases} x^3 + 3y^2 &= 21 \\ x^2 + 2y + 2 &= 0 \end{cases}$$

Haga dibujos de las gráficas para localizar valores aproximados de las intersecciones y de esta forma escoger un iterado inicial $\mathbf{x}_0$.

25. Al dividir por su coeficiente principal distinto de cero, cualquier ecuación cúbica puede ponerse en la forma

$$x^3 + ax^2 + bx + c = 0.$$

Según el *Teorema de Vieta*, las tres raíces de tal ecuación cúbica satisfacen el sistema de ecuaciones no lineales

$$\begin{cases} -(x_1 + x_2 + x_3) &= 1 \\ x_1x_2 + x_2x_3 + x_1x_2 &= b \\ -x_1x_2x_3 &= c \end{cases}$$

Escriba un programa basado en el método de Newton, o cualquier otro método de su elección, para resolver este sistema para cualquier polinomio cúbico dado. Tenga en cuenta que dos de las raíces pueden ser complejas incluso si los coeficientes son reales, por lo que deberá usar un cálculo de inicio complejo y una aritmética compleja para encontrar las raíces complejas. Pruebe su programa en una variedad de ecuaciones cúbicas y compare sus resultados con las raíces calculadas por la función `roots` de la librería `Numpy`.

26. Un modelo para la combustión de propano en aire produce el sistema de ecuaciones no lineales

$$\begin{cases} x_1 + x_4 &= 3 \\ 2x_1 + x_2 + x_4 + x_7 + x_8 + x_9 + 2x_{10} &= R + 10 \\ 2x_2 + 2x_5 + x_6 + x_7 &= 8 \\ 2x_3 + x_5 &= 4R \\ x_1x_5 - 0.193x_2x_4 &= 0 \\ x_6\sqrt{x_2} - 0.002597\sqrt{x_2x_4\sum_{i=1}^{10} x_i} &= 0 \\ x_7\sqrt{x_4} - 0.003448\sqrt{x_1x_4\sum_{i=1}^{10} x_i} &= 0 \\ x_4x_8 - 0.00001799x_2\sum_{i=1}^{10} x_i &= 0 \\ x_4x_9 - 0.0002155x_1\sqrt{x_3\sum_{i=1}^{10} x_i} &= 0 \\ x_4^2\left(x_{10} - 0.00003846\sum_{i=1}^{10} x_i\right) &= 0 \end{cases}$$

donde $R = 4.056734$. Emplee convenientemente el método de Newton o el método de Broyden para resolver este sistema de ecuaciones no lineales. Compare sus resultados con los resultados de la función `fsolve`. (*Ayuda*: si alguna raíz cuadrada falla porque una variable se vuelve negativa, reemplace esa variable por su valor absoluto.)

5 Diferenciación e Integración Numérica

Las operaciones elementales del Cálculo, diferenciación e integración, a menudo deben ser atacadas numéricamente. En muchas aplicaciones surgen funciones que poseen expresiones matemáticas muy complejas o, más aún, existen modelos que no tienen expresiones matemática, sino un procedimiento algorítmico que arroja un valor. En tales situaciones no se pueden obtener expresiones cerradas de la derivada o la integral. Simplemente consideremos un experimento donde se toman mediciones y no se conoce el modelo matemático que describe el proceso físico involucrado. En este caso y en muchos otros, son necesarias aproximaciones numéricas de la derivada y la integral.

A lo largo de este capítulo desarrollaremos fórmulas que aproximan la derivada de una función, a través del polinomio interpolante y el método de *diferencias finitas*, que tienen una importante aplicación en la aproximación de ecuaciones diferenciales. En este capítulo también presentaremos algunas fórmulas para la aproximación de la integral de una función. Tanto en diferenciación como en integración, realizaremos un análisis del error.

5.1 Diferenciación Numérica

Como es conocido del curso de Cálculo, la derivada de una función $f : \mathbb{R} \to \mathbb{R}$ en un punto $a \in \mathbb{R}$, se define como

$$f'(a) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h}. \tag{5.1}$$

Esta definición, denominada *límite del cociente incremental*, se puede usar para aproximar el valor de $f'(a)$. Sea $h > 0$ un número pequeño, entonces una aproximación de la derivada de $f(x)$ en $x = a$, usando (5.1), está dada por:

$$f'(a) \approx D_h(a) = \frac{f(a+h) - f(a)}{h}. \tag{5.2}$$

Surge la siguiente pregunta: ¿qué valor de h hay que elegir para asegurar que $D_h(a)$ es una buena aproximación de la derivada de $f(x)$ en $x = a$? Para responder esta pregunta veamos un ejemplo.

Ejemplo 5.1. Consideremos la función $f(x) = e^x$ y aproximemos el valor de $f'(1)$ usando la fórmula (5.2). Como $f'(x) = e^x$, entonces $f'(1) = e^1 \approx 2.718281828459045$. En la Tabla 5.1 se muestran los valores de $D_h(1)$ y el error $|f'(1) - D_h(1)|$, para distintos valores de h.

Tabla 5.1. Valores de $D_h(1)$ para $f(x) = e^x$

h	$D_h(1)$	$\lvert f'(1) - D_h(1)\rvert$
0.1	2.858841955	0.140560126
0.01	2.731918656	0.013636827
0.001	2.719641423	0.001359594
0.0001	2.718417747	0.000135919
0.00001	2.718295420	0.000013591
1.0e-06	2.718283187	0.000001359
1.0e-07	2.718281968	0.000000140
1.0e-08	2.718281822	0.000000007
1.0e-09	2.718282044	0.000000215
1.0e-10	2.718283376	0.000001548
1.0e-11	2.718314462	0.000032634
1.0e-12	2.718714143	0.000432314
1.0e-13	2.717825964	0.000455864
1.0e-14	2.708944180	0.009337648
1.0e-15	3.108624469	0.390342640

Se observa que el valor de $D_{0.1}(1)$ no proporciona una buena aproximación; en cambio, $D_{10^{-8}}(1)$ proporciona la mejor aproximación. Pero, a medida que disminuye $h < 10^{-8}$, la aproximación $D_h(1)$ se hace menos precisa, es decir, aumenta la diferencia con respecto al valor real de $f'(1)$.

El comportamiento observado en el Ejemplo 5.1 indica que aproximar la derivada con h pequeño produce inestabilidad, es decir, para valores muy pequeños de h el valor de $D_h(a)$ se aleja de $f'(a)$. Esto se debe a que para valores pequeños de h, los puntos a y $a+h$ son muy cercanos, con lo cual la diferencia $f(a+h) - f(a)$ está muy cercana a cero, ocasionando la pérdida de cifras significativas. Generalmente, en la práctica, se elige $h \approx 10^{-8}$ para que (5.2) sea una aproximación de la derivada $f'(a)$.

El Ejemplo 5.1 proporciona una idea del valor de h que hay que elegir para que (5.2) sea una buena aproximación de $f'(a)$. Ahora bien, la fórmula (5.2) posee una precisión del orden $\mathbf{O}(h)$. Expliquemos que significa esto. Como $D_h(a)$ aproxima a $f'(a)$, entonces $D_h(a)$ es una aproximación de la pendiente de la recta tangente a la curva $y = f(x)$, en el punto $(a, f(a))$. Particularmente, $D_h(a)$ es la pendiente de la recta secante que pasa por los puntos $(a, f(a))$ y $(a+h, f(a+h))$, cuya ecuación es:

$$y = D_h(a)(x-a) + f(a).$$

En otras palabras, la fórmula (5.2) es la derivada del polinomio $p_1(x) = a_0 + a_1 x$, que interpola a $f(x)$ en los puntos $(a, f(a))$ y $(a+h, f(a+h))$. Esto significa que muy cerca de a estamos aproximando a $f(x)$ con el polinomio $p_1(x)$. Entonces, por el Teorema 3.2 se tiene que, para cada $x \in [a, a+h]$, existe $\beta_x \in (a, a+h)$ tal que

$$f(x) = p_1(x) + \frac{f^{(2)}(\beta_x)}{2}(x-a)(x-(a+h)),$$

de lo cual se deduce que

$$f'(x) = p_1'(x) - \frac{f^{(2)}(\beta_x)}{2}\,[2(x-a)-h] \qquad \Rightarrow \qquad \lvert f'(x) - p_1'(x)\rvert = Ch,$$

para $x \in [a, a+h]$ y cierta constante positiva C. En este caso, el error $|f'(a) - D_h(a)| = |f'(a) - p_1'(a)|$ está acotado por un polinomio en h de grado 1, lo cual significa que la fórmula (5.2) es de orden $\mathbf{O}(h)$.

En la práctica, decir que una fórmula de diferenciación tiene orden $\mathbf{O}(h)$ indica, por ejemplo, si $h = 10^{-5}$, entonces esa fórmula genera una aproximación cuyas 5 primeras cifras significativas son iguales a las 5 primeras cifras del valor real de la derivada. Si quisiéramos obtener una aproximación con 10 cifras significativas usando una fórmula de orden $\mathbf{O}(h)$, necesitaríamos elegir $h = 10^{-10}$, lo cual produciría con toda seguridad una mala aproximación de la derivada, por elegir un valor de h muy pequeño. Este hecho nos hace preguntar: ¿existen fórmulas de orden superior a $\mathbf{O}(h)$, digamos $\mathbf{O}(h^m)$, para aproximar la derivada de $f(x)$? Si existiese una fórmula, $E_h(a)$, que aproxime a la derivada $f'(a)$, por ejemplo de orden $\mathbf{O}(h^2)$, entonces se satisface que $|f'(a) - E_h(a)| \leq Ch^2$; en este caso, si quisiéramos obtener una aproximación de $f'(a)$ con 10 cifras significativas, elegiríamos $h = 10^{-5}$, con lo cual no tendríamos inestabilidad. Estas fórmulas de orden superior a $\mathbf{O}(h)$ existen y las veremos a continuación.

5.1.1 Fórmulas de Diferencias Finitas

Supongamos que la función $f(x)$ posee derivadas de cualquier orden. Seguidamente deduciremos fórmulas para aproximar la primera, segunda, tercera y cuarta derivada de una función $f(x)$, con distintos órdenes de precisión. Para la deducción de estas fórmulas necesitamos los siguientes desarrollos de Taylor.

Se deja como ejercicio para el lector deducir estos desarrollos de Taylor

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2!}f''(x) + \frac{h^3}{3!}f'''(x) + \frac{h^4}{4!}f^{(4)}(x) + \cdots \tag{5.3}$$

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2!}f''(x) - \frac{h^3}{3!}f'''(x) + \frac{h^4}{4!}f^{(4)}(x) - \cdots \tag{5.4}$$

$$f(x+2h) = f(x) + 2hf'(x) + \frac{(2h)^2}{2!}f''(x) + \frac{(2h)^3}{3!}f'''(x) + \frac{(2h)^4}{4!}f^{(4)}(x) + \cdots \tag{5.5}$$

$$f(x-2h) = f(x) - 2hf'(x) + \frac{(2h)^2}{2!}f''(x) - \frac{(2h)^3}{3!}f'''(x) + \frac{(2h)^4}{4!}f^{(4)}(x) - \cdots \tag{5.6}$$

donde $h > 0$.

Primeras Diferencias Centrales

Primero deduciremos una fórmula que involucra la evaluación de la función $f(x)$ en los puntos $x-h$ y $x+h$, para aproximar la primera derivada de $f(x)$. Restando las ecuaciones (5.3) y (5.4) tenemos

$$f(x+h) - f(x-h) = 2hf'(x) + \frac{2h^3}{3!}f'''(x) + \cdots$$

Despejando $f'(x)$ de esta última ecuación obtenemos:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{6}f'''(x) - \cdots$$

Considerando sólo los primeros términos del lado derecho de la igualdad, obtenemos la fórmula

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathbf{O}(h^2),$$

que se denomina aproximación por *primeras diferencias centrales* de $f'(x)$, donde el término $\mathbf{O}(h^2)$ nos dice que esta fórmula tiene un error de truncamiento de orden h^2.

Deduzcamos ahora la fórmula central para aproximar la segunda derivada de $f(x)$. Sumando las ecuaciones (5.3) y (5.4), tenemos:

$$f(x+h)+f(x-h)=2f(x)+\frac{2h^2}{2!}f''(x)+\frac{2h^4}{4!}f^{(4)}(x)+\cdots$$

Despejando $f''(x)$ obtenemos:

$$f''(x)=\frac{f(x+h)-2f(x)+f(x-h)}{h^2}-\frac{h^2}{12}f^{(4)}(x)+\cdots$$

o, equivalentemente,

$$f''(x)=\frac{f(x+h)-2f(x)+f(x-h)}{h^2}+\mathbf{O}(h^2),$$

que se denomina aproximación por primeras diferencias centrales de $f''(x)$.

Sumando o restando las ecuaciones (5.3), (5.4), (5.5) y (5.6) se obtienen las siguientes aproximaciones por primeras diferencias centrales de $f'''(x)$ y $f^{(4)}(x)$, que poseen error de truncamiento de orden $\mathbf{O}(h^2)$. Se deja como ejercicio para el lector encontrar las ecuaciones involucradas en cada caso.

$$f'''(x)=\frac{f(x+2h)-2f(x+h)+2f(x-h)-f(x-2h)}{2h^3}+\mathbf{O}(h^2),$$

$$f^{(4)}(x)=\frac{f(x+2h)-4f(x+h)+6f(x)-4f(x-h)+f(x-2h)}{h^4}+\mathbf{O}(h^2).$$

Primeras Diferencias No Centrales

A continuación mostramos algunas primeras diferencias no centrales de $f'(x)$, $f''(x)$, $f'''(x)$ y $f^{(4)}(x)$. Cada una de estas diferencias no centrales posee error de truncamiento de orden $\mathbf{O}(h)$. Se deja como ejercicio para el lector determinar cuáles de las ecuaciones (5.3), (5.4), (5.5) y (5.6), están involucradas en cada caso.

Primeras Diferencias Hacia Adelante

$$f'(x)=\frac{f(x+h)-f(x)}{h}+\mathbf{O}(h),$$

$$f''(x)=\frac{f(x+2h)-2f(x+h)+f(x)}{h^2}+\mathbf{O}(h),$$

$$f'''(x)=\frac{f(x+3h)-3f(x+2h)+3f(x+h)-f(x)}{h^3}+\mathbf{O}(h),$$

$$f^{(4)}(x)=\frac{f(x+4h)-4f(x+3h)+6f(x+2h)-4f(x+h)+f(x)}{h^4}+\mathbf{O}(h).$$

Primeras Diferencias Hacia Atrás

$$f'(x) = \frac{f(x) - f(x-h)}{h} + \mathbf{O}(h),$$

$$f''(x) = \frac{f(x) - 2f(x-h) + f(x-2h)}{h^2} + \mathbf{O}(h),$$

$$f'''(x) = \frac{f(x) - 3f(x-h) + 3f(x-2h) - f(x-3h)}{h^3} + \mathbf{O}(h),$$

$$f^{(4)}(x) = \frac{f(x) - 4f(x-h) + 6f(x-2h) - 4f(x-3h) + f(x-4h)}{h^4} + \mathbf{O}(h).$$

Segundas Diferencias No Centrales

Las aproximaciones mayormente usadas son las de orden $\mathbf{O}(h^2)$. Seguidamente damos diferencias no centrales de orden $\mathbf{O}(h^2)$.

Segundas Diferencias Hacia Adelante

$$f'(x) = \frac{-f(x+2h) + 4f(x+h) - 3f(x)}{2h} + \mathbf{O}(h^2),$$

$$f''(x) = \frac{-f(x+3h) + 4f(x+2h) - 5f(x+h) + 2f(x)}{h^2} + \mathbf{O}(h^2),$$

$$f'''(x) = \frac{-3f(x+4h) + 14f(x+3h) - 24f(x+2h) + 18f(x+h) - 5f(x)}{2h^3} + \mathbf{O}(h^2),$$

$$f^{(4)}(x) = \frac{-2f(x+5h) + 11f(x+4h) - 24f(x+3h) + 26f(x+2h) - 14f(x+h) + 3f(x)}{h^4} + \mathbf{O}(h^2).$$

Segundas Diferencias Hacia Atrás

$$f'(x) = \frac{3f(x) - 4f(x-h) + f(x-2h)}{2h} + \mathbf{O}(h^2),$$

$$f''(x) = \frac{2f(x) - 5f(x-h) + 4f(x-h) - f(x-3h)}{h^2} + \mathbf{O}(h^2),$$

$$f'''(x) = \frac{5f(x) - 18f(x-h) + 24f(x-2h) - 14f(x-3h) + 3f(x-4h)}{2h^3} + \mathbf{O}(h^2),$$

$$f^{(4)}(x) = \frac{3f(x) - 14f(x-h) + 26f(x-2h) - 24f(x-3h) + 11f(x-4h) - 2f(x-5h)}{h^4} + \mathbf{O}(h^2).$$

En los Programas 5.1 y 5.1 se implementan las primeras diferencias no centrales y las primeras diferencias centrales, respectivamente. Las funciones `pndc` y `pndc` tienen como salida una función `d(x)` que aproxima la m-ésima derivada $f(x)$, para $m = 1,2,3,4$.

Programa 5.1. Primeras Diferencias No Centrales

```
def pdnc(f, m, h=1.0e-4):
    def d(x):
        if m == 1:
            return (f(x+h)-f(x))/h
        elif m == 2:
            return (f(x+2*h)-2*f(x+h)+f(x))/(h**2)
        elif m == 3:
            return (f(x+3*h)-3*f(x+2*h)+3*f(x+h)-f(x))/(h**3)
        elif m == 4:
            return (f(x+4*h)-4*f(x+3*h)+6*f(x+2*h)-4*f(x+h)
                    +f(x))/(h**4)
        else:
            return (f(x+h)-f(x))/h
    return d
```

Programa 5.2. Primeras Diferencias Centrales

```
def pdc(f, m, h=1.0e-4):
    def d(x):
        if m == 1:
            return (.5*f(x+h)-.5*f(x-h))/h
        elif m == 2:
            return (f(x+h)-2*f(x)+f(x-h))/(h**2)
        elif m == 3:
            return (.5*f(x+2*h)-f(x+h)+f(x-h)-.5*f(x-2*h))/(h**3)
        elif m == 4:
            return (f(x+2*h)-4*f(x+h)+6*f(x)-4*f(x-h)+f(x-2*h))/(h**4)
        else:
            return (.5*f(x+h)-.5*f(x-h))/h
    return d
```

Examinemos lo que sucede cuando estas fórmulas se usan para aproximar la derivada de una función dada. Todas estas fórmulas de diferencias finitas convergen cuando $h \to \infty$. En un ajuste teórico podemos obtener una exactitud arbitrariamente buena haciendo h suficientemente pequeño, pero como vimos anteriormente, en la práctica elegir h muy pequeño produce la pérdida de cifras significativas. Veamos un ejemplo donde comparamos distintas fórmulas de diferencias finitas para aproximar la primera derivada de una función.

Ejemplo 5.2. Estudiemos el comportamiento numérico de las primeras diferencias centrales y las primeras diferencias no centrales hacia adelante, en la aproximación de la primera derivada de $f(x) = \cos(x)$ en $x = 0.5$. Se tiene que

$$f'(x) = -\operatorname{sen}(x) \qquad \text{y} \qquad f'(0.5) = -\operatorname{sen}(0.5) = -0.479425538604203$$

En la Tabla 5.2 se muestran los valores de las primeras diferencias centrales, DC_h, y de las primeras diferencias no centrales hacia adelante, DNC_h, y el error de truncamiento de cada una de esta fórmulas, para distintos valores de h.

Tabla 5.2. Comportamiento numérico de las primeras diferencias centrales y primeras diferencias no centrales hacia adelante, en la aproximación la primera derivada de $f(x) = \cos(x)$ en $x = 0.5$, para distintos valores de h

h	$DNC_h(0.5)$	$\lvert DNC_h - f'(0.5)\rvert$	$DC_h(0.5)$	$\lvert DC_h(0.5)(0.5) - f'(0.5)\rvert$
1.0e-01	-0.522469469807	4.304e-02	-0.478626895466	7.986e-04
1.0e-02	-0.483805424462	4.380e-03	-0.479417548219	7.990e-06
1.0e-03	-0.479864249944	4.387e-04	-0.479425458700	7.990e-08
1.0e-04	-0.479469416934	4.388e-05	-0.479425537805	7.988e-10
1.0e-05	-0.479429926514	4.388e-06	-0.479425538596	7.725e-12
1.0e-06	-0.479425977473	4.389e-07	-0.479425538602	2.174e-12
1.0e-07	-0.479425582567	4.396e-08	-0.479425538713	1.088e-10
1.0e-08	-0.479425554811	1.621e-08	-0.479425543709	5.105e-09
1.0e-09	-0.479425610322	7.172e-08	-0.479425554811	1.621e-08
1.0e-10	-0.479426498501	9.599e-07	-0.479425943389	4.048e-07
1.0e-11	-0.479427608724	2.070e-06	-0.479422057609	3.481e-06

Las mejores aproximaciones obtenidas por las diferencias finitas son:

$$DNC_{10^{-8}}(0.5) = -0.479425554811 \qquad \text{y} \qquad DC_{10^{-6}}(0.5) = -0.479425538602$$

El error de $DC_{10^{-6}}(0.5)$ es 2.173×10^{-12}, en cambio el error de $DNC_{10^{-8}}(0.5)$ es 1.621×10^{-8}. Claramente se aprecia que las diferencias centrales encuentran mejores aproximaciones de $f'(0.5)$, pero ellas también adolecen de inestabilidad.

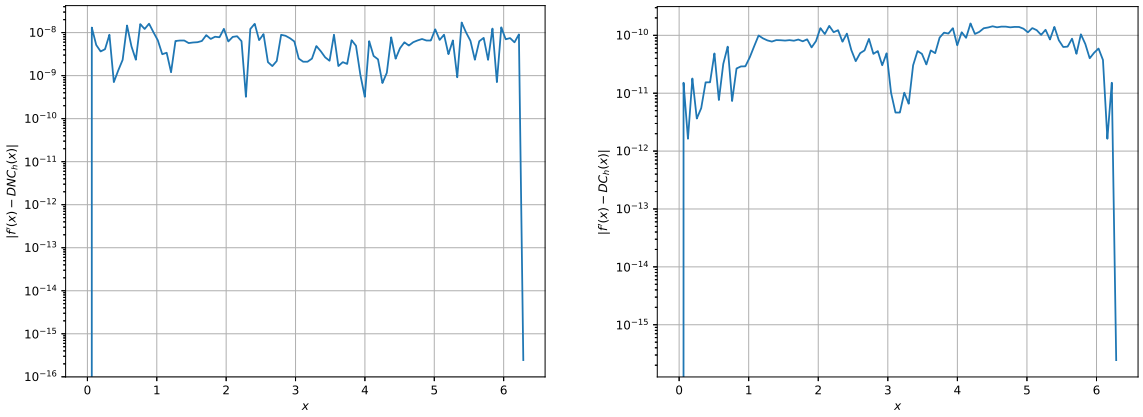
Las funciones `pndc` y `pndc` pueden emplearse para aproximar $f'(x)$ en un intervalo. Por ejemplo, en la Figura 5.1 se muestra la gráfica de los errores

$$|f'(x) - DNC_h(x)| \qquad \text{y} \qquad |f'(x) - DC_h(x)|,$$

obtenidos al aproximar $f'(x) = -\text{sen}(x)$, en el intervalo $[0, 2\pi]$, con las primeras diferencias no centrales y las primeras diferencias centrales, tomando $h = 10^{-8}$ y $h = 10^{-6}$, respectivamente. Tales gráficas se generan ejecutando el siguiente script.

```
df_dc = cn.pdc(f, 1, h=1.0e-6)
df_dnc = cn.pdnc(f, 1, h=1.0e-8)
x = np.linspace(0,2*np.pi,100)
plt.figure(1)
plt.plot(x,np.abs(df_dnc(x)+np.sin(x)))
plt.yscale('log')
plt.grid()
plt.xlabel('$x$')
plt.ylabel('$|f´(x)-DNC_{h}(x)|$')
plt.figure(2)
plt.plot(x,np.abs(df_dc(x)+np.sin(x)))
plt.yscale('log')
plt.grid()
plt.xlabel('$x$')
plt.ylabel('$|f´(x)-DC_{h}(x)|$')
plt.show()
```

Note que `df_dnc(x)` y `df_dc(x)` son funciones que aproximan a $f'(x) = -\text{sen}(x)$ en el intervalo $[0, 2\pi]$ con un error máximo menor que 10^{-7} y 10^{-9}, respectivamente.



(a) Error $|f'(x) - DNC_h(x)|$ (b) Error $|f'(x) - DC_h(x)|$

Figura 5.1. Errores $|f'(x) - DNC_h(x)|$ y $|f'(x) - DC_h(x)|$ en el intervalo $[0, 2\pi]$, para la función $f(x) = \cos(x)$.

5.2 Integración Numérica

Para una función $f : \mathbb{R} \to \mathbb{R}$, la definición de la integral de $f(x)$ en el intervalo $[a,b]$,

$$I(f) = \int_a^b f(x)\,dx,$$

está basada en sumas de Riemann de la forma

$$R_n = \sum_{i=0}^{n-1} (x_{i+1} - x_i)\, f(\xi_i),$$

donde $a = x_0 < x_1 < \cdots < x_{n-1} < x_n = b$ y $\xi \in [x_{i+1}, x_i]$, $i = 0, 1, \ldots, n-1$. Sea $h_n = \text{máx}\{x_{i+1} - x_i : i = 0, 1, \ldots, n-1\}$. Si para alguna elección de x_i tal que $h_n \to 0$ y alguna escogencia de ξ_i, se tiene $\lim_{n\to\infty} R_n = R < \infty$, entonces $f(x)$ es integrable en $[a,b]$ y el valor de la integral es $I(f) = R$. Esta definición ya sugiere un método para aproximar una integral: solo debemos usar una suma finita de Riemann con n elegido lo suficientemente grande para lograr la precisión deseada. Esta idea funciona, siempre y cuando los puntos x_i y ξ_i se elijan con cuidado, además, requiere la evaluación de la función integrando f en muchos más puntos que el número mínimo necesario para lograr una precisión dada.

Basados en la definición de la integral como una suma Riemann, la integración numéri-ca envuelve reemplazar una integral por una suma. Supongamos que $f(x)$ es una función integrable en el intervalo $[a,b]$ y sean $x_0, x_1, \ldots, x_n$ puntos distintos en el intervalo $[a,b]$. Entonces la aproximación

$$\int_a^b f(x)\,dx \approx \sum_{i=0}^{n} w_i\, f(x_i), \tag{5.7}$$

con una apropiada escogencia de w_i, es una regla de integración numérica. La regla (5.7), por razones históricas, se denomina *cuadratura*, el cual es un término que se usa a menudo como sinónimo para la integración numérica en una dimensión. Los números w_i son los *pesos* y los puntos x_i los *nodos*. Todas las reglas de integración que discutiremos aquí tendrán la forma (5.7); solamente la diferencia entre ellas es la elección de los pesos y nodos de la cuadratura.

Toda cuadratura es una aproximación al valor real de la integral; por ello existe un error de aproximación. En otras palabras, se tiene que

$$\int_a^b f(x)\,dx = \sum_{i=0}^{n} w_i\, f(x_i) + E_f,$$

donde E_f se llama error de *truncamiento de la cuadratura*. Un aspecto importante en todas las aplicaciones es el conocimiento del grado de precisión de la cuadratura, es decir, el orden del error de truncamiento.

Definición 5.1 (Grado de precisión)**.** El grado de precisión de una cuadratura es el número natural m que satisface lo siguiente: $E_{P_i} = 0$ para todos los polinomios $P_i(x)$ de grado $i \leq m$, y existe un polinomio $P_{m+1}(x)$ de grado $m+1$ tal que $E_{P_{n+1}} \neq 0$.

A continuación deduciremos cuadraturas con distintos grados de precisión. Un grupo de ellas son las fórmulas de Newton-Cotes, que se basan en la interpolación polinómica. También describiremos un procedimiento general para obtener cuadraturas con un grado de precisión dado.

5.2.1 Fórmulas de Newton-Cotes

La fórmulas de Newton-Cotes son cuadraturas que se basan en la interpolación polinómica. Sean $x_0, x_1, \ldots, x_n$ nodos en el intervalo $[a,b]$. Recordemos que existe un único polinomio $p_n(x)$ de grado menor o igual a n que satisface las ecuaciones de interpolación $f(x_i) = p_n(x_i)$, para $i = 0, 1, \ldots, n$; además, por el Teorema 3.2 se cumple:

$$f(x) = p_n(x) + \frac{1}{(n+1)!}\, f^{(n+1)}(\beta_x) \prod_{i=0}^{n} (x - x_i),$$

donde $\beta_x \in (a,b)$. De esta forma, la integral de $f(x)$ en el intervalo $[a,b]$ es:

$$\begin{aligned}\int_a^b f(x)\,dx &= \int_a^b p_n(x)\,dx + \int_a^b \left[\frac{1}{(n+1)!}\, f^{(n+1)}(\beta_x) \prod_{i=0}^{n} (x - x_i)\right] dx \\ &= \int_a^b p_n(x)\,dx + \frac{1}{(n+1)!} \int_a^b \left[f^{(n+1)}(\beta_x) \prod_{i=0}^{n} (x - x_i)\right] dx. \end{aligned} \tag{5.8}$$

Ahora, expresando a $p_n(x)$ convenientemente en su forma de Lagrange,

$$p_n(x) = \sum_{i=0}^{n} f(x_i)\ell_i(x),$$

la Ecuación (5.8) adquiere la forma

$$\int_a^b f(x)\,dx = \sum_{i=0}^{n} w_i\, f(x_i) + \frac{1}{(n+1)!} \int_a^b \left[f^{(n+1)}(\beta_x) \prod_{i=0}^{n} (x - x_i)\right] dx, \tag{5.9}$$

donde

$$w_i = \int_a^b \ell_i(x)\,dx$$

son los pesos y el término

$$\frac{1}{(n+1)!}\int_a^b \left[f^{(n+1)}(\beta_x)\prod_{i=0}^{n}(x-x_i)\right] dx$$

es el error de truncamiento. Si los nodos están uniformemente espaciados en el intervalo $[a,b]$,

$$x_i = a+ih, \qquad i=0,1,\ldots,n,$$

la expresión (5.9) se denomina *Fórmula de Newton-Cotes*, donde $h=(b-a)/n$. En el siguiente teorema se describe el error asociado con las fórmulas de Newton-Cotes.

Teorema 5.1. *Sean $x_i = a+ih$, para $i=0,1,\ldots,n$ y $h=(b-a)/n$, nodos igualmente espaciados en el intervalo $[a,b]$. Entonces, existe $\alpha\in(a,b)$, para el cual se cumple*

$$\int_a^b f(x)\,dx = Q(f) + \frac{h^{n+3}f^{(n+2)}(\alpha)}{(n+2)!}\int_0^n t\prod_{j=0}^{n}(t-j)\,dt, \tag{5.10}$$

si n es par, y

$$\int_a^b f(x)\,dx = Q(f) + \frac{h^{n+2}f^{(n+1)}(\alpha)}{(n+1)!}\int_0^n \prod_{j=0}^{n}(t-j)\,dt, \tag{5.11}$$

si n es impar, donde

$$Q(f)=\sum_{i=0}^{n} w_i f(x_i), \qquad w_i=\int_a^b \ell_i(x)\,dx$$

y $\ell_0(x),\ell_2(x),\ldots,\ell_n(x)$ son los polinomios de Lagrange.

El Teorema 5.1 nos permite calcular el grado de precisión de una fórmula de Newton-Cotes. Por (5.10) y (5.11), el error de truncamiento de la fórmula de Newton-Cotes es

$$E_f = \frac{h^{n+3}f^{(n+2)}(\alpha)}{(n+2)!}\int_0^n t\prod_{j=0}^{n}(t-j)\,dt$$

si n es par, y es

$$E_f = \frac{h^{n+2}f^{(n+1)}(\alpha)}{(n+1)!}\int_0^n \prod_{j=0}^{n}(t-j)\,dt$$

si n es impar. Si $f(x)$ es un polinomio de grado n, entonces $f^{(n+1)}(x)=f^{(n+2)}(x)=0$; por lo que, $E_f=0$ para cualquier n. En otras palabras, una fórmula de Newton-Cotes es exacta para polinomios de grado $n+1$, si n es par, y es exacta para polinomios de grado n, si n es impar.

Regla del Trapecio

La regla del trapecio tiene una interpretación geométrica. La regla del trapecio dice que la integral de $f(x)$ en el intervalo $[a,b]$, se aproxima como el área del trapecio formado por los puntos $(a,0)$, $(a,f(a))$, $(b,f(b))$ y $(b,0)$ (observe la Figura 5.2).

La regla del trapecio es el caso más sencillo de las fórmulas de Newton-Cotes, ocurre cuando $n=1$ y los nodos son $x_0=a$, $x_1=b$. En este caso, $h=(b-a)$ y los polinomios de Lagrange son

$$\ell_0(x)=\frac{b-x}{b-a}, \qquad \ell_1(x)=\frac{x-a}{b-a}.$$

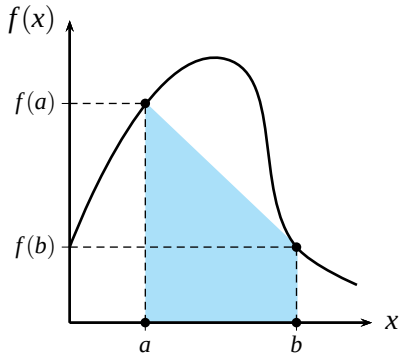


Figura 5.2. Regla del Trapecio

En consecuencia, los pesos de la fórmula de Newton-Cotes resultante son

$$w_0 = \int_a^b \ell_0(x)\,dx = \frac{b-a}{2} = \int_a^b \ell_1(x)\,dx = w_1.$$

Así, la cuadratura correspondiente es

$$\int_a^b f(x)\,dx \approx Q(f) = \frac{(b-a)}{2}\left[f(a)+f(b)\right].$$

Esta expresión se conoce como *regla del trapecio*; además, por el Teorema 5.1, su término de error es

$$E_f = \frac{h^3 f^{(2)}(\alpha)}{2!}\int_0^1 t(t-1)\,dt = -\frac{(b-a)^3}{12}\,f^{(2)}(\alpha),$$

donde $\alpha \in (a,b)$. Por lo tanto, el grado de precisión de la regla del trapecio es 1, es decir, es exacta para polinomios de grado menor o igual a 1. En efecto, la integral de un polinomio $p(x) = a_0 + a_1 x$ es

$$\int_a^b p(x)\,dx = a_0\int_a^b dx + a_1\int_a^b x\,dx = \frac{(b-a)}{2}\left[2a_0 + a_1(b+a)\right].$$

Ahora, si aplicamos la regla del trapecio con $f(x) = p(x)$, obtenemos:

$$\begin{aligned} Q(p) &= \frac{(b-a)}{2}\left[p(a)+p(b)\right] \\ &= \frac{(b-a)}{2}\left[a_0 + a_1 a + a_0 + a_1 b\right] \\ &= \frac{(b-a)}{2}\left[2a_0 + a_1(b+a)\right] \\ &= \int_a^b p(x)\,dx. \end{aligned}$$

Note que el error de la regla del trapecio depende de la longitud del intervalo $[a,b]$. Si la longitud del intervalo es grande, digamos 10^8, entonces el error de trapecio pudiera ser muy grande. Para evitar esta discrepancia, se puede subdividir el intervalo $[a,b]$ en subintervalos donde se aplique la regla del trapecio, para luego sumar sus aproximaciones. En otras palabras, en el intervalo $[a,b]$ se hace un partición como la siguiente:

$$a = x_0 < x_1 < \cdots < x_n = b,$$

donde $x_i = a + ih$ y $h = (b-a)/n$. Se tiene que

$$\begin{aligned}\int_a^b f(x)\,dx &= \int_{x_0}^{x_1} f(x)\,dx + \int_{x_1}^{x_2} f(x)\,dx + \cdots + \int_{x_{n-1}}^{x_n} f(x)\,dx \\ &= \sum_{i=1}^{n} \int_{x_{i-1}}^{x_i} f(x)\,dx.\end{aligned}$$

Entonces, al aplicar la regla del trapecio en cada uno de los subintervalos $[x_{i-1}, x_i]$, para $i = 1, \ldots, n$, obtenemos:

$$\int_a^b f(x)\,dx \approx \frac{1}{2}\sum_{i=1}^{n}(x_i - x_{i-1})\left[f(x_{i-1}) + f(x_i)\right].$$

Como los nodos x_i están uniformemente espaciados, la expresión anterior adquiere la forma

$$\int_a^b f(x)\,dx \approx Q(f) = \frac{h}{2}\left[f(a) + 2\sum_{i=1}^{n-1} f(x_i) + f(b)\right], \tag{5.12}$$

que se conoce como *regla del trapecio compuesta*, cuyo término de error es

$$E_f = -\frac{(b-a)}{12}h^2 f''(\alpha),$$

el cual se obtiene sumando el término de error de la regla del trapecio correspondiente a cada intervalo $[x_{i-1}, x_i]$, donde $\alpha \in (a,b)$.

En el Programa 5.3 se muestra la implementación de la regla del trapecio compuesta. Si se toma $n = 1$, la función `trapecio` es la regla del trapecio.

Programa 5.3. Regla del trapecio compuesta

```
def trapecio(f,a,b,n=1):
    if(n==1):
        h = b-a
        return (h/2.)*(f(a)+f(b))
    else:
        h = float((b-a)/n)
        x = np.linspace(a,b,n+1)
        fx = f(x)
        return (h/2)*(fx[0] + 2.*sum(fx[1:n]) + fx[n])
```

Regla de Simpson

Al igual que la regla del trapecio, la regla de Simpson también tiene una interpretación geométrica. La regla de Simpson dice que la integral de $f(x)$ en el intervalo $[a,b]$, se aproxima como el área comprendida entre el eje de las abscisas y la parábola que pasa por los puntos $(a, f(a))$, $\left(\frac{(a+b)}{2}, f\left(\frac{(a+b)}{2}\right)\right)$ y $(b, f(b))$ (observe la Figura 5.3).

Tomemos $n = 2$ y los nodos $x_0 = a$, $x_1 = (a+b)/2$ y $x_2 = b$ y veamos que forma adquiere la fórmula de Newton-Cotes. En este caso, $h = (b-a)/2$ y los polinomios de Lagrange son

$$\ell_0(x) = \frac{2(x - \frac{a+b}{2})(x-b)}{(b-a)^2}, \quad \ell_1(x) = -\frac{4(x-a)(x-b)}{(b-a)^2}, \quad \ell_2(x) = \frac{2(x-a)(x - \frac{a+b}{2})}{(b-a)^2}.$$

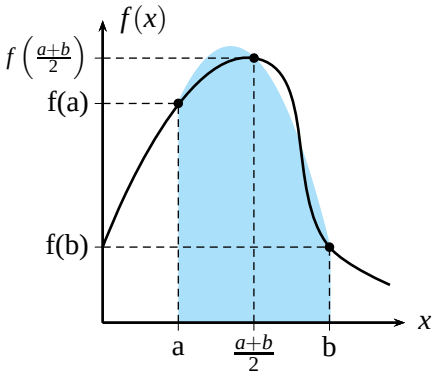


Figura 5.3. Regla de Simpson

En consecuencia, los pesos de la fórmula de Newton-Cotes resultante son

$$w_0 = \int_a^b \ell_0(x)\,dx = \frac{(b-a)}{6},$$
$$w_1 = \int_a^b \ell_1(x)\,dx = 4\,\frac{(b-a)}{6},$$
$$w_2 = \int_a^b \ell_2(x)\,dx = \frac{(b-a)}{6}.$$

Así, la cuadratura correspondiente es

$$\int_a^b f(x)\,dx \approx Q(f) = \frac{(b-a)}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right],$$

que se conoce como *regla de Simpson*; además, por el Teorema 5.1, su término de error es

$$E_f = \frac{h^5 f^{(4)}(\alpha)}{4!}\int_0^2 t^2(t-1)(t-2)\,dt = -\frac{(b-a)^5}{2880}\,f^{(4)}(\alpha),$$

donde $\alpha \in (a,b)$. El grado de precisión de la regla de Simpson es 3.

Pasemos ahora a deducir la *regla de Simpson compuesta*. Sea n un número par y tomemos los nodos igualmente espaciados $x_i = a + ih$, $i = 0,1,\ldots,n$, $h = (b-a)/n$. Se tiene que

$$\begin{aligned}\int_a^b f(x)\,dx &= \int_{x_0}^{x_2} f(x)\,dx + \int_{x_2}^{x_4} f(x)\,dx + \cdots + \int_{x_{n-2}}^{x_n} f(x)\,dx \\ &= \sum_{i=1}^{n/2}\int_{x_{2i-1}}^{x_{2i}} f(x)\,dx.\end{aligned}$$

Entonces, al aplicar la regla de Simpson en cada uno de los subintervalos $[x_{2i-2}, x_{2i}]$, para $i = 1,2,\ldots,n/2$, obtenemos:

$$\begin{aligned}\int_a^b f(x)\,dx &\approx \frac{h}{3}\sum_{i=1}^{n/2}\left[f(x_{2i-2}) + 4f(x_{2i-1}) + f(x_{2i})\right] \\ &= \frac{h}{3}\left[f(a) + 2\sum_{i=2}^{n/2} f(x_{2i-2}) + 4\sum_{i=1}^{n/2} f(x_{2i-1}) + f(b)\right],\end{aligned}$$

que se conoce como regla de Simpson compuesta, cuyo término de error es

$$E_f = -\frac{(b-a)}{180} h^4 f^{(4)}(\alpha),$$

donde $\alpha \in (a,b)$. En el Programa 5.4 se muestra la implementación de la regla de Simpson compuesta. Si se toma $n = 2$, la función `simpson` es la regla de Simpson.

Programa 5.4. Regla de Simpson compuesta

```
def simpson(f,a,b,n=2):
    if(n==2):
        h = (b-a)/2.
        return (h/3.)*(f(a)+ 4.*f((a+b)/2.) + f(b))
    else:
        h = float((b-a)/n)
        x = np.linspace(a,b,n+1)
        fx = f(x)
        return (h/3)*(fx[0] + 2.*sum(fx[range(2,n-1,2)]) +
               4.*sum(fx[range(1,n,2)]) + fx[n])
```

Ejemplo 5.3. Empleemos las reglas de trapecio, Simpson, trapecio compuesta y Simpson compuesta, para calcular la integral

$$\int_0^{2\pi} xe^{-x}\cos(2x)\,dx = \frac{3\left(e^{-2\pi}-1\right)-10\pi e^{-2\pi}}{25} \simeq -0.122122605 \tag{5.13}$$

Utilicemos las funciones `trapecio` y `simpson`. El comando `trapecio(f,a,b)` arroja la aproximación de la integral de $f(x)$ en el intervalo $[a,a]$, usando la regla del trapecio; ahora el comando `trapecio(f,a,b,m)`, con m un entero mayor que 1, arroja la aproximación de la integral usando la regla del trapecio compuesta. En cuanto a la función `simpson`, el comando `simpson(f,a,b)` arroja la aproximación de la integral usando la regla de simpson y el comando `simpson(f,a,b,m)`, con m un número par mayor que 2, arroja la aproximación de la integral usando la regla de Simpson compuesta.

Tabla 5.3. Comportamiento de las reglas de trapecio, Simpson, trapecio compuesta y Simpson compuesta en la aproximación de la integral (5.13)

Trapecio			Simpson		
n	$Q(f)$	E_f	n	$Q(f)$	E_f
1	0.036861842	1.590e-01	2	0.580959651	7.031e-01
4	-0.356950842	2.348e-01	4	-0.624246190	5.021e-01
8	-0.178475421	5.635e-02	8	-0.118983614	3.139e-03
16	-0.135396849	1.327e-02	16	-0.121037324	1.085e-03
32	-0.125385808	3.263e-03	32	-0.122048794	7.381e-05
64	-0.122934894	8.123e-04	64	-0.122117923	4.682e-06
128	-0.122325457	2.029e-04	128	-0.122122311	2.940e-07
256	-0.122173304	5.070e-05	256	-0.122122586	1.875e-08
512	-0.122135279	1.267e-05	512	-0.122122603	1.529e-09
1024	-0.122125773	3.168e-06	1024	-0.122122605	4.528e-10

En la Tabla 5.3 se muestra el comportamiento de las reglas de trapecio, Simpson, trapecio compuesta y Simpson compuesta en la aproximación de la integral (5.13), para distintos valores

de n. El caso $n = 1$ en la columna Trapecio, corresponde a la aproximación obtenida con la regla de trapecio, en otros casos es la aproximación obtenida con la regla de trapecio compuesta para el n indicado. El caso $n = 2$ en la columna Simpson, corresponde a la aproximación obtenida con la regla de Simpson, en otros casos es la aproximación obtenida con la regla de Simpson compuesta para el n indicado.

En el ejemplo anterior observamos que las reglas de trapecio compuesta y Simpson compuesta, aumentan su precisión a medida que el número de nodos crece. Por supuesto, esto era de esperarse debido a que trapecio compuesta tiene un error de $\mathbf{O}(h^2)$, Simpson compuesta tiene un error de $\mathbf{O}(h^4)$ y $h = (b-a)/n$. Ahora bien, si en alguna aplicación es necesario aproximar el valor de una integral de tal manera de garantizar cierta precisión, es decir, que el error no sobrepase $\varepsilon \in (0,1)$, ¿cuántos nodos son necesarios para que el error de trapecio compuesta o el de Simpson compuesta, sean menores o iguales a ε?. En el siguiente ejemplo damos la respuesta a esta pregunta.

Ejemplo 5.4. Empleemos las reglas del trapecio compuesta y Simpson compuesta para calcular la integral

$$\int_0^{\pi} \left(e^{x/2} + \cos(4x)\right) dx = 2\left(e^{\pi/2} - 1\right) \simeq 7.620954762$$

con una precisión de seis cifras significativas o, equivalentemente, que el error en la aproximación sea menor o igual que 5×10^{-6}. Se tiene que $f(x) = e^{x/2} + \cos(4x)$, $a = 0$, $b = \pi$ y $h = (b-a)/n$. Entonces, el valor absoluto del error de la regla del trapecio compuesta es

$$|E_f| = \left| -\frac{(b-a)}{12} h^2 f''(\alpha) \right| = \frac{\pi}{12} \left(\frac{\pi}{n}\right)^2 |f''(\alpha)| = \left(\frac{\pi^3}{12n^2}\right) |f''(\alpha)|.$$

Como

$$f''(x) = \frac{e^{x/2}}{4} - 16\cos(4x),$$

entonces $|f''(\alpha)| \leq 18$, para $\alpha \in (0, \pi)$. De esta forma obtenemos:

$$|E_f| \leq \left(\frac{\pi^3}{12n^2}\right) 18 = \frac{3\pi^3}{2} \cdot \frac{1}{n^2}$$

Como deseamos que $|E_f| \leq 5 \times 10^{-6}$, entonces de la desigualdad anterior se deduce:

$$\frac{3\pi^3}{2} \cdot \frac{1}{n^2} \leq 5 \times 10^{-6} \qquad \Rightarrow \qquad n \geq \sqrt{3\pi^3 10^5} \simeq 3049.899$$

en otras palabras, se necesitan $n = 3050$ nodos en la regla del trapecio compuesta para obtener una precisión de 6 cifras significativas en la aproximación de la integral dada. Verifiquemos esta afirmación. Corramos la función `trapecio` con $n = 3050$.

```
>>> def f(x): return np.exp(x/2)+np.cos(4*x)
>>> trapecio(f,0,np.pi,3050)
7.6209549303796154
```

Observamos que las seis primeras cifras de la aproximación obtenida, `7.6209549303796154`, coinciden con las seis primeras cifras del valor de la integral, `7.620954762`.

Realicemos un razonamiento similar al anterior para obtener el número de nodos necesarios para que la regla de Simpson compuesta tenga un error menor que 5×10^{-6}. El valor absoluto del error de la regla de Simpson compuesta es

$$|E_f| = \left|-\frac{(b-a)}{180}h^4 f^{(4)}(\alpha)\right| = \frac{\pi}{180}\left(\frac{\pi}{n}\right)^4 |f^{(4)}(\alpha)| = \left(\frac{\pi^5}{180n^4}\right)|f^{(4)}(\alpha)|.$$

Como

$$f^{(4)}(x) = \frac{e^{x/2}}{16} + 256\cos(4x),$$

entonces $|f^{(4)}(\alpha)| \leq 260$, para $\alpha \in (0,\pi)$. Así obtenemos:

$$|E_f| \leq \left(\frac{\pi^5}{180n^4}\right)260 = \frac{13\pi^5}{9}\cdot\frac{1}{n^4}$$

Como deseamos que $|E_f| \leq 5\times10^{-6}$, entonces de la desigualdad anterior se deduce:

$$\frac{13\pi^5}{9}\cdot\frac{1}{n^4} \leq 5\times10^{-6} \qquad \Rightarrow \qquad n \geq \left(\frac{13\pi^5 10^6}{45}\right)^{1/4} \simeq 96.966$$

en otras palabras, se necesitan $n = 98$ nodos en la regla del trapecio compuesta para obtener una precisión de 6 cifras significativas en la aproximación de la integral dada. Verifiquemos que esto es cierto. Corramos la función `simpson` con $n = 98$.

```
>>> simpson(f,0,pi,98)
7.6209547647251679
```

Observamos que las ocho primeras cifras de la aproximación obtenida, `7.6209547647251679`, coinciden con las ocho primeras cifras del valor de la integral, `7.620954762`.

5.2.2 Método de los Coeficientes Indeterminados

El *método de los coeficientes indeterminados* consiste en encontrar una cuadratura

$$Q(f) = \sum_{i=0}^{n} w_i f(x_i)$$

de manera que su grado de exactitud sea m o, equivalentemente, que sea exacta para polinomios de grado menor o igual a m. En muchos casos es necesario escoger los pesos w_i o los nodos x_i, dependiendo de la función $f(x)$. A manera de ejemplo observe la Figura 5.4. En esta figura observamos que los nodos x_0, x_1, x_2, x_3 se escogieron según la forma de la gráfica de $f(x)$, entre x_2 y x_3 no es necesario colocar muchos nodos, ya que $f(x)$ es una constante; en cambio, entre x_0 y x_3 la forma de $f(x)$ es parecida a una parábola, por ello se colocó el nodo x_2. Para este ejemplo se tiene la siguiente cuadratura

$$Q(f) = w_0 f(x_0) + w_1 f(x_1) + w_2 f(x_2) + w_3 f(x_3).$$

Ahora bien, ¿cómo calculamos los valores de los pesos w_i? ¿Qué grado de precisión tiene esta cuadratura? Para darle respuesta a estas preguntas emplearemos el método de los coeficientes indeterminados.

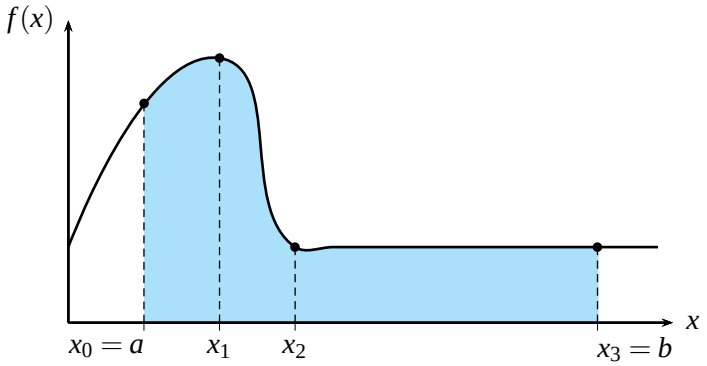


Figura 5.4. Ubicación particular de nodos

Si todos los $n+1$ nodos x_i se conocen, el grado de precisión de una cuadratura está dado por el número de pesos que no son conocidos menos 1; por ejemplo, supongamos que los nodos de la Figura 5.4 son: $x_0 = 1$, $x_1 = 2$, $x_2 = 3$ y $x_3 = 7$. En este caso, la cuadratura asociada a la Figura 5.4 es:

$$Q(f) = w_0 f(1) + w_1 f(2) + w_2 f(3) + w_3 f(7), \tag{5.14}$$

la cual tiene un grado de precisión de 3, es decir, esta cuadratura es exacta para polinomios de grado ≤ 3, ya que los pesos no conocidos son 4, w_0, w_1, w_2, w_3. Para calcular los valores de los pesos de la cuadratura (5.14) se procede de la siguiente manera. Como la cuadratura es exacta para polinomios de grado ≤ 3, usamos como función de prueba los polinomios $f(x) = 1, x, x^2$ y x^3,

$$\begin{aligned}
7 &= \int_1^7 dx = w_0 + w_1 + w_2 + w_3 \\
24 &= \int_1^7 x\,dx = w_0 + 2w_1 + 3w_2 + 7w_3 \\
114 &= \int_1^7 x^2\,dx = w_0 + 4w_1 + 9w_2 + 49w_3 \\
600 &= \int_1^7 x^3\,dx = w_0 + 8w_1 + 27w_2 + 343w_3,
\end{aligned}$$

es decir, los pesos w_i son las incógnitas del sistema de ecuaciones lineales

$$\left\{\begin{array}{rcl}
w_0 + w_1 + w_2 + w_3 &=& 7 \\
w_0 + 2w_1 + 3w_2 + 7w_3 &=& 24 \\
w_0 + 4w_1 + 9w_2 + 49w_3 &=& 114 \\
w_0 + 8w_1 + 27w_2 + 343w_3 &=& 600
\end{array}\right.$$

La solución de este sistema de ecuaciones lineales es

$$w_0 = \frac{13}{2}, \quad w_1 = -\frac{57}{5}, \quad w_2 = \frac{43}{4}, \quad w_3 = \frac{23}{20}.$$

De esta forma, la cuadratura

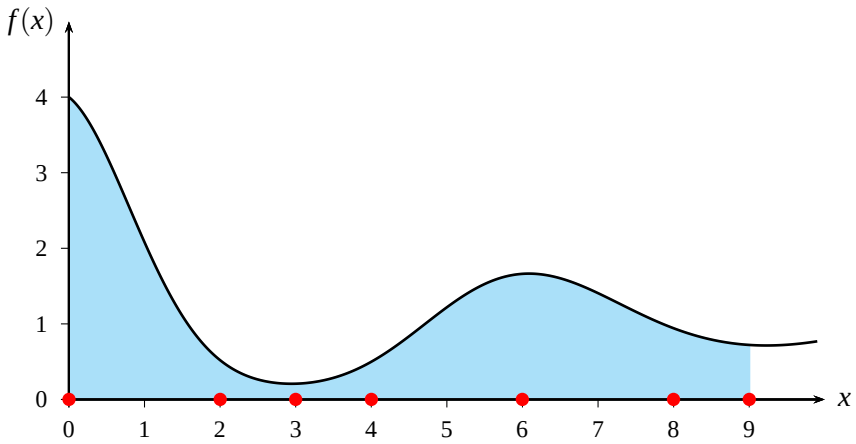
$$\int_1^7 f(x)\,dx \simeq \frac{13}{2} f(1) - \frac{57}{5} f(2) + \frac{43}{4} f(3) + \frac{23}{20} f(7)$$

es exacta para polinomios de grado ≤ 3. Se deja como ejercicio para el lector comprobar que la cuadratura obtenida es exacta para polinomios de la forma $p(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3$.

Ejemplo 5.5. Determinemos una cuadratura que utilice 7 nodos en el intervalo $[0,9]$, para calcular la integral

$$\int_0^9 \left|e^{-x/5}\cos(x)+1\right|^2 dx \simeq 10.83166931355786$$

En la siguiente figura se muestra la gráfica de $f(x)$.



En esta figura también se muestran los 7 nodos escogidos: $0,2,3,4,6,8$ y 9, con los cuales definimos la siguiente cuadratura:

$$Q(f) = w_0 f(0) + w_1 f(2) + w_2 f(3) + w_3 f(4) + w_4 f(6) + w_5 f(8) + w_6 f(9)$$

Programa 5.5. Cuadratura general

```
def cuad(f,a,b,x):
    xi = np.asarray(x,dtype=float)
    n = len(x)
    W = np.zeros((n,n))
    v = np.zeros(n)
    for i in range(0,n):
        v[i] = float((b**(i+1)-a**(i+1))/(i+1))
        W[i,:] = xi**i
    w = np.linalg.solve(W,v)
    fxi = f(xi)
    return np.sum(w*fxi),w
```

En el Programa 5.5 se muestra la función `cuad`, que permite calcular los pesos w_i y, además, arroja la aproximación de la integral de una función $f(x)$ en el intervalo $[a,b]$, tomando como parámetros de entrada: $f(x)$, a, b y los $(n+1)$ nodos $x_0, x_1, \ldots, x_n$.

Ejecutemos la función `cuad` con la función $f(x) = \left|e^{-x/5}\cos(x)+1\right|^2$, $a = 0$, $b = 9$ y los nodos $0,2,3,4,6,8$ y 9.

```
>>> def f(x): return np.abs(np.exp(-x/5.)*np.cos(x)+1)**2.
>>> I,w = cuad(f,0,9,[0,2,3,4,6,8,9])
>>> print(I)
10.2139769793
>>> print(w)
[ 0.54040179  4.13781888 -3.27857143  3.90535714  1.67544643  1.78995536
  0.22959184]
```

Los pesos obtenidos son:

$$w_0 = 0.54040179, \quad w_1 = 4.13781888, \quad w_2 = -3.27857143, \quad w_3 = 3.90535714,$$
$$w_4 = 1.67544643, \quad w_5 = 1.78995536, \quad w_6 = 0.22959184,$$

con los cuales construimos la cuadratura

$$\begin{aligned} Q(f) =& 0.54040179\, f(0) + 4.13781888\, f(2) - 3.27857143\, f(3) + 3.90535714\, f(4) + \\ & 1.67544643\, f(6) + 1.78995536\, f(8) + 0.22959184\, f(9) \end{aligned}$$

La función `cuad` también arrojó la aproximación de la integral dada,

$$Q(f) = 10.2139769793$$

Observamos que el error cometido es: 0.6176923342352758, el cual es bastante grande. Comparemos este resultado con las aproximaciones de las reglas de trapecio compuesta y Simpson compuesta. Ejecutemos las funciones `trapecio` y `simpson` con 8 nodos.

```
>>> def f(x): return np.abs(np.exp(-x/5.)*np.cos(x)+1)**2.
>>> trapecio(f,0,9,8)
10.918562502712827
>>> simpson(f,0,9,8)
10.753616562604318
```

La regla de trapecio compuesta obtuvo la aproximación: 10.918562502712827, con un error: 0.08689318915496713; en cambio, la regla de Simpson compuesta obtuvo la aproximación: 10.753616562604318, con un error: 0.07805275095354247.

En el ejemplo anterior presentamos una aplicación del método de los coeficientes indeterminados, que a simple vista parece muy atractiva, pero la misma tiene inconvenientes numéricos. Uno de estos inconvenientes es el mal condicionamiento de la matriz asociada al sistema de ecuaciones lineales que tiene como incógnita los pesos. El mal condicionamiento de esta matriz aumenta a medida que el número de pesos crece, lo cual produce un aumento en el error de aproximación obtenida por la cuadratura. Para verificar esto numéricamente, se deja como ejercicio para el lector correr la función `cuad` con la función $f(x)$ del ejemplo 5.5, tomando 100 nodos igualmente espaciados en el intervalo $[0,9]$. El método de los coeficientes indeterminados es una gran herramienta teórica, que permite definir cuadraturas cuyo comportamiento numérico en la práctica es eficiente. Una de esas cuadraturas es la Gaussiana.

5.2.3 Cuadraturas de Gauss-Legendre

En esta sección nos centraremos en el cálculo de la integral de $f(x)$ en el intervalo $[-1,1]$. Es claro de la discusión previa que pueden desarrollarse muchas fórmulas de integración numérica y lograr altos órdenes de convergencia, simplemente aumentando el grado de aproximación. En el pensamiento práctico, se suelen preferir métodos compuestos de orden relativamente bajos. Cuadraturas de orden superior, no sólo tienen un patrón de pesos más complicado, sino que pueden aumentar la inestabilidad (observe el Ejemplo 5.5).

El método de coeficientes indeterminados se puede utilizar para generar cuadraturas del tipo

$$\int_{-1}^{1} f(x)\,dx \approx \sum_{i=0}^{n-1} w_i f(x_i),$$

que son exactas para polinomios de grado $\leq n$. En tales fórmulas los nodos $x_0,,\ldots,x_{n-1} \in [-1,1]$ deben ser conocidos. Una vez fijados los nodos, los pesos w_i se obtienen resolviendo un sistema de ecuaciones lineales. Es natural preguntarse, ¿es necesario evaluar la función $f(x)$ en n nodos para que la cuadratura sea exacta para polinomios de grado $\leq n$? La respuesta a esta pregunta es no. A continuación, a manera de ejemplo, construyamos una cuadratura que es exacta para polinomios de grado ≤ 3, que necesita de dos evaluaciones de la función.

Hallemos los nodos x_0, x_1 y los pesos w_0 y w_1 de manera que la cuadratura

$$\int_{-1}^{1} f(x)\,dx \approx w_0 f(x_0) + w_1 f(x_1) \tag{5.15}$$

sea exacta para polinomios de grado ≤ 3. Asumamos que $x_0, x_1 \neq 0$ y $w_0, w_1 \neq 0$. Tomando como función de prueba los polinomios $1, x, x^2$ y x^3, obtenemos:

$$\begin{aligned}
2 &= \int_{-1}^{1} dx = w_0 + w_1 \\
0 &= \int_{-1}^{1} x\,dx = w_0 x_0 + w_1 x_1 \\
\frac{2}{3} &= \int_{-1}^{1} x^2\,dx = w_0 x_0^2 + w_1 x_1^2 \\
0 &= \int_{-1}^{1} x^3\,dx = w_0 x_0^3 + w_1 x_1^3
\end{aligned}$$

es decir, los nodos x_i y los pesos w_i son las incógnitas del sistema de ecuaciones no lineales

$$\begin{cases}
w_0 + w_1 &= 2 \\
w_0 x_0 + w_1 x_1 &= 0 \\
w_0 x_0^2 + w_1 x_1^2 &= \frac{2}{3} \\
w_0 x_0^3 + w_1 x_1^3 &= 0
\end{cases} \tag{5.16}$$

Despejando w_1 en la segunda ecuación de (5.16), $w_1 = -w_0 x_0 / x_1$, luego sustituyendo en la última ecuación de (5.16), considerando que $x_0, x_1 \neq 0$, $w_0, w_1 \neq 0$ y $x_0 \neq x_1$, obtenemos:

$$0 = w_0 x_0^3 - \frac{w_0 x_0}{x_1} \cdot x_1^3 = w_0 x_0 (x_0^2 - x_1^2) \quad \Rightarrow \quad x_0^2 = x_1^2 \quad \Rightarrow \quad x_0 = -x_1.$$

Usando esta última igualdad en combinación con la segunda ecuación de (5.16), encontramos:

$$w_0 = w_1 \left(-\frac{x_1}{x_0} \right) \quad \Rightarrow \quad w_0 = w_1.$$

Utilizando la primera ecuación de (5.16), obtenemos:

$$w_0 + w_1 = 2 \quad \Rightarrow \quad 2w_0 = 2 \quad \Rightarrow \quad w_0 = 1,$$

por lo tanto,

$$w_0 = w_1 = 1.$$

Ahora, teniendo presente que $w_0 = w_1 = 1$ y $x_0 = -x_1$, de la tercera ecuación de (5.16) se deduce:

$$(-x_1)^2 + x_1^2 = \frac{2}{3} \quad \Rightarrow \quad x_1^2 = \frac{1}{3} \quad \Rightarrow \quad x_1 = \frac{\sqrt{3}}{3} \quad \Rightarrow \quad x_0 = -\frac{\sqrt{3}}{3}.$$

Por lo tanto, la cuadratura (5.15) adquiere la forma:

$$\int_{-1}^{1} f(x)\,dx \approx G_2(f) = f\left(-\frac{\sqrt{3}}{3}\right) + f\left(\frac{\sqrt{3}}{3}\right). \tag{5.17}$$

La fórmula (5.17) se denomina cuadratura de *Gauss-Legendre de dos puntos*, que es exacta para polinomios de grado $n \leq 3$.

Pasemos ahora a generalizar el procedimiento anterior para obtener una cuadratura de Gauss-Legendre de n puntos

$$\int_{-1}^{1} f(x)\,dx \approx G_n(f) = \sum_{i=0}^{n-1} w_i\, f(x_i), \tag{5.18}$$

que es exacta para polinomios de grado $2n-1$. Es de notar que la resolución del sistema de ecuaciones no lineales que involucra este procedimiento es muy difícil cuando n aumenta; por ello nosotros primero encontraremos los nodos x_i. Estos nodos son las raíces del *polinomio de Legendre*, $L_n(x)$, definido por la siguiente relación de recurrencia:

$$\begin{aligned} L_0(x) &= 1, \qquad L_1(x) = x, \\ L_{n+1}(x) &= (2n+1)\,xL_n(x) - iL_{n-1}(x), \quad n = 2,3,\ldots \end{aligned}$$

Los cinco primeros polinomios de Legendre son:

$$\begin{aligned} L_0(x) &= 1, & L_1(x) &= x, \\ L_2(x) &= \frac{1}{2}(3x^2-1), & L_3(x) &= \frac{1}{2}(5x^3-3x), \\ L_4(x) &= \frac{1}{8}(35x^4-30x^2+3), & L_5(x) &= \frac{1}{8}(63x^5-70x^3+15x), \end{aligned}$$

Volviendo a la cuadratura de Gauss-Lengendre de n puntos, después de encontrar los nodos x_i, se emplea el método de los coeficientes indeterminados con lo cual obtenemos que los pesos son las soluciones del sistema de ecuaciones lineales

$$\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ x_0 & x_1 & x_2 & \cdots & x_{n-1} \\ & \vdots & & & \\ x_0^i & x_1^i & x_2^i & \cdots & x_{n-1}^i \\ & \vdots & & & \\ x_0^{n-1} & x_1^{n-1} & x_2^{n-1} & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_i \\ \vdots \\ w_{n-1} \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \\ \vdots \\ (1-(-1)^{i+1})/(i+1) \\ \vdots \\ (1-(-1)^n)/n \end{bmatrix} \tag{5.19}$$

Ejemplo 5.6. Encontremos los nodos y pesos de la cuadratura de Gauss-Legendre de 3 puntos sobre el intervalo $[-1,1]$, que es exacta para polinomios de grado ≤ 5. Los nodos son las raíces del polinomio de Legendre $L_3(x) = \frac{1}{2}(5x^3-3x)$, las cuales son:

$$x_0 = -\sqrt{\frac{3}{5}}, \qquad x_1 = 0, \qquad x_2 = \sqrt{\frac{3}{5}}.$$

Ahora el sistema de ecuaciones (5.19) es:

$$\begin{bmatrix} 1 & 1 & 1 \\ -(3/5)^{1/2} & 0 & (3/5)^{1/2} \\ (3/5) & 0 & (3/5) \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \\ 2/3 \end{bmatrix}$$

cuya solución es

$$w_0 = \frac{5}{9}, \qquad w_1 = \frac{8}{9}, \qquad w_2 = \frac{5}{9}.$$

Así, la cuadratura de Gauss-Legendre de 3 puntos está dada por

$$\int_{-1}^{1} f(x)\,dx \approx G_3(f) = \frac{5}{9} f\left(-\sqrt{\frac{3}{5}}\right) + \frac{8}{9} f(0) + \frac{5}{9} f\left(-\sqrt{\frac{3}{5}}\right). \tag{5.20}$$

El procedimiento anterior para hallar los nodos y los pesos de la cuadratura de Gauss-Legendre de n puntos se puede realizar en Python. Con la función `leggauss` de la librería `Numpy`, se hallan los nodos y pesos de la cuadratura de Gauss-Legendre de n puntos. Por ejemplo, para hallar los nodos y pesos de la cuadratura de Gauss-Legendre de 3 puntos se procede de la siguiente forma.

```
>>> from numpy.polynomial.legendre import leggauss
>>> xi,wi = leggauss(3)
>>> print(xi)
[-0.77459667  0.          0.77459667]
>>> print(wi)
[ 0.55555556  0.88888889  0.55555556]
```

Las cuadraturas de Gauss-Legendre también pueden usarse en un intervalo cualquiera $[a,b]$. Supongamos que tenemos los nodos $x_0,\ldots,x_{n-1} \in [-1,1]$ y los pesos $w_0,\ldots,w_{n-1}$ de la cuadratura de Gauss-Legendre de n puntos. Entonces, para aplicar la cuadratura de Gauss-Legendre en un intervalo $[a,b]$, se puede usar el cambio de variables

$$t = \frac{a+b}{2} + \frac{b-a}{2} x. \tag{5.21}$$

De esta forma,

$$\int_a^b f(t)\,dt = \int_{-1}^{1} f\left(\frac{a+b}{2} + \frac{b-a}{2} x\right)\left(\frac{b-a}{2}\right) dx,$$

lo cual indica que la cuadratura de Gauss-Legendre de n puntos en un intervalo $[a,b]$ está dada por

$$\int_a^b f(t)\,dt \approx \left(\frac{b-a}{2}\right) \sum_{i=0}^{n-1} w_i\, f\left(\frac{a+b}{2} + \frac{b-a}{2} x_i\right) \tag{5.22}$$

En el Programa 5.6, se implementa la cuadratura de Gauss-Legendre de n puntos para calcular la integral de una función $f(x)$ en el intervalo $[a,b]$.

Programa 5.6. Cuadratura de Gauss-Legendre

```
def gauss_legendre(f,a,b,n):
    xi,wi = np.polynomial.legendre.leggauss(n)
    ti = (a+b)/2. + ((b-a)/2.)*xi
    return ((b-a)/2.)*np.sum(f(ti)*wi)
```

Ejemplo 5.7. Calculemos la integral

$$\int_0^9 \left| e^{-x/5}\cos(x)+1 \right|^2 dx \simeq 10.83166931355786$$

usando la función `gauss_legendre`. Primero aproximamos el valor de la integral con la cuadratura de Gauss-Legendre de 2 puntos.

```
>>> def f(x): return np.abs(np.exp(-x/5.)*np.cos(x)+1)**2.
>>> gauss_legendre(f,0,9,2)
8.8386742022795808
```

Ahora aproximamos el valor de la integral con la cuadratura de Gauss-Legendre de 3 puntos.

```
>>> gauss_legendre(f,0,9,3)
10.833540902116502
```

Ahora aproximamos el valor de la integral con la cuadratura de Gauss-Legendre de 12 puntos.

```
>>> gauss_legendre(f,0,9,12)
10.83166930920514
```

Se observa que las cuadraturas de Gauss-Legendre de 2 y 3 puntos obtienen una aproximación con 2 cifras significativas; en cambio, la cuadratura de Gauss-Legendre de 12 puntos obtiene una aproximación con 7 cifras significativas.

■

5.2.4 Cuadraturas Adaptativas

Utilizar una regla compuesta con nodos igualmente espaciados puede ser eficiente para muchas funciones integrando, sin embargo, si la función a integrar, en un intervalo dado, cambia de forma suave en algunas partes del intervalo y cambia bruscamente en otras, entonces usar una cuadratura con un paso fijo h en todo el intervalo puede producir un trabajo adicional innecesario. Un enfoque más inteligente es la *cuadratura adaptativa*, en la cual el intervalo de integración se restringe selectivamente para reflejar el comportamiento de cualquier función a integrar. Las cuadraturas adaptativas representan un ajuste dinámico del paso h para lograr que se cometa un error en todo el intervalo menor que una tolerancia establecida previamente.

Una estrategia típica de cuadratura adaptativa funciona de la siguiente manera. Se requieren dos cuadraturas, digamos Q_{n_1} y Q_{n_2}, cuya diferencia proporciona una estimación de error. Un ejemplo simple es usar la regla del trapecio compuesta con un número de puntos de nodos distintos. Sin embargo, en general, una mayor eficiencia se obtiene con reglas de mayor precisión, como las cuadraturas Gaussianas. Otra alternativa es usar una sola regla en dos niveles diferentes de subdivisión; la regla de Simpson es una opción popular en este enfoque.

El procedimiento de adaptación es conceptualmente simple: se aplican ambas reglas Q_{n_1} y Q_{n_2} en el intervalo inicial de integración $[a,b]$. Si los valores aproximados resultantes para la integral difieren en más de la tolerancia deseada, se divide el intervalo en dos o más subintervalos y se repite el procedimiento en cada subintervalo. Si la tolerancia se cumple en un subintervalo dado, entonces no se requerirá una subdivisión adicional de ese subintervalo. Si la tolerancia no se cumple en un subintervalo dado, entonces el proceso de subdivisión se repite nuevamente, y así sucesivamente hasta que se cumpla la tolerancia en todos los subintervalos. Dicha estrategia

conduce a un muestreo no uniforme de la función integrando que coloca muchos nodos en regiones donde la función es difícil de integrar y relativamente pocos nodos donde la función se integra fácilmente.

Existen una serie de problemas importantes de implementación de las cuadraturas adaptativas, entre los cuales podemos nombrar:

- ¿Cómo se debe implementar el criterio de parada? Por ejemplo, ¿debería ser la tolerancia de error relativa (generalmente preferible) o absoluta (en caso de que el valor de la integral esté cerca de cero), o una combinación de los dos?
- ¿Se puede cumplir siempre la tolerancia? ¿Terminará siempre la recursión?
- ¿Cómo podemos evitar perder tiempo subdividiendo subintervalos no convergentes que hacen una contribución insignificante a la integral total?
- ¿Cómo deberíamos permitir los efectos de la aritmética de precisión finita? Por ejemplo, ¿qué sucede si la longitud de un subintervalo es tan pequeña que contiene solo sus extremos?

En muchas rutinas de cuadratura adaptativa, estos problemas se resuelven mediante el uso de una combinación de pruebas de error relativo y absoluto que involucran parámetros dependientes de la máquina (como el epsilon de máquina), junto con un límite superior en el número de niveles de subdivisión permitido. Cuando se enfrentan a un problema difícil, como un integrando ruidoso o no suave o una tolerancia de error poco realista, tales rutinas pueden gastar un gran número de evaluaciones de funciones antes de devolver una respuesta inexacta acompañada de un mensaje de advertencia de que se excedió el límite de subdivisiones.

En las implementaciones actuales, generalmente se utiliza un criterio de terminación robusto, que evita límites arbitrarios. Un ingrediente importante en este criterio es una estimación previa aproximada de la integral, digamos $\widehat{I}$, en el intervalo de integración. Una estimación aproximada se puede obtener de varias maneras, muestreando el integrando $f(x)$ en unos pocos nodos elegidos al azar dentro del intervalo de integración, o la manera más común es utilizar una cuadratura para aproximar el valor de la integral. En un subintervalo, que denotaremos $[a,b]$ (sin pérdida de generalidad), se estima el valor de $\widehat{I}$ utilizando, por ejemplo, la cuadratura Q_{n_1}; luego, se emplea la cuadratura Q_{n_2} para estimar los valores I_L y I_R de las integrales en los subintervalos $[a,(a+b)/2]$ y $[(a+b)/2,b]$, respectivamente. Inmediatamente se compara el valor $\widehat{I}$ con la suma I_L+I_R,

$$\left|\widehat{I}-(I_L+I_R)\right|<\varepsilon, \tag{5.23}$$

donde $0<\varepsilon<1$ es una tolerancia preestablecida. Si el criterio de parada (5.23) se satisface en el intervalo de integración, se detiene el proceso de subdivisión. En el Programa 5.7 se implementa el procedimiento genérico de cuadratura adaptativa con las características descritas previamente y que emplea el criterio de parada (5.23). La implementación dada utiliza un proceso de recursión y una sola cuadratura, es decir, $Q_{n_1}=Q_{n_2}$.

Programa 5.7. Cuadratura Adaptativa

```
def cuadadapt(f, a, b, cuadratura, tol=1e-8):
    integral = 0
    errest = 0
    c = 0.5*(a + b)
    I  = cuadratura(f, a, b)
    Il = cuadratura(f, a, c)
    Ir = cuadratura(f, c, b)
    err = np.abs(I - (Il + Ir))
    if err < tol:
```

```
        return I, err
    else:
        Ii, errI  = cuadadapt(f, a, c, cuadratura, tol/2)
        Id, errD  = cuadadapt(f, c, b, cuadratura, tol/2)
        integral += (Ii  + Id)
        errest += (errI + errD)
    return integral, errest
```

Para usar la función `cuadadapt` es necesario disponer de una cuadratura. Combinando la regla del trapecio compuesta, la regla de Simpson compuesta y la cuadratura de Gauss-Legendre con la función `cuadadapt`, implementamos tres cuadraturas adaptativas que se muestran en los Programas 5.8, 5.9 y 5.10, y que hemos denominado trapecio adaptativa, Simpson adaptativa y Gauss-Legendre adaptativa, respectivamente.

Programa 5.8. Trapecio Adaptativa

```
def trapecioadapt(f,a,b,n=1,tol=1.0e-8):
    cuadratura = lambda f,a,b: trapecio(f,a,b,n=n)
    I, err = cuadadapt(f, a, b, cuadratura, tol=tol)
    return I, err
```

Programa 5.9. Simpson Adaptativa

```
def simpsonadapt(f,a,b,n=2,tol=1.0e-8):
    cuadratura = lambda f,a,b: simpson(f,a,b,n=n)
    I, err = cuadadapt(f, a, b, cuadratura, tol=tol)
    return I, err
```

Programa 5.10. Gauss-Legendre Adaptativa

```
def gaussadapt(f,a,b,n=2,tol=1.0e-8):
    cuadratura = lambda f,a,b: gauss_legendre(f,a,b,n)
    I, err = cuadadapt(f, a, b, cuadratura, tol=tol)
    return I, err
```

En Python existen distintas funciones que permiten calcular la integral de una función. En particular, en la función `quadrature` del módulo `integrate` de la librería `Scipy`, se implementa la cuadratura Gauusiana adaptativa.

Los parámetros y salida de

```
quadrature(func, a, b, args=(), tol=1.49e-08, rtol=1.49e-08, maxiter=50,
     vec_func=True, miniter=1, vec_func=True, miniter=1)
```

son los siguientes:

Parámetros

`func`: función
: Función definida en Python que devuelve un número o un arreglo.

`a`: float
: Límite inferior de integración.

`b`: float
: Límite superior de integración.

`args`: n-upla, opcional
: Contiene argumentos extras para la función `func`.

`tol,rtol`: float, opcional
: Las iteraciones se detienen cuando el error entre dos iterados consecutivos es menor que `tol` o el cambio relativo es menor que `rtol`.

`maxiter`: int, opcional
: Máximo orden de la cuadratura Gaussiana.

`vec_func`: bool, opcional
: `True` o `False` si `func` es una función vectorial. Por defecto es `True`.

`miniter`: int, opcional
: Mínimo orden de la cuadratura Gaussiana.

Salida

`val`: float
: Aproximación obtenida por la cuadratura.

`err`: float
: Diferencia entre dos estimaciones de la integral.

Ejemplo 5.8. Empleemos las funciones de `trapecioadapt`, `simpsonadapt`, `gaussadapt` y `quadrature`, para calcular la integral

$$\int_0^{2\pi} xe^{-x}\cos(2x)\,dx = \frac{3\left(e^{-2\pi}-1\right)-10\pi e^{-2\pi}}{25} \simeq -0.122122604618968$$

Carguemos las librerías necesarias, definamos la función $f(x)$ y el intervalo de integración.

```
>>> import numpy as np
>>> from scipy.integrate import quadrature
>>> def f(x): return x*np.exp(-x)*np.cos(2*x)
>>> a = 0
>>> b = 2*np.pi
```

Tomamos $n = 10$ puntos en las funciones `trapecioadapt`, `simpsonadapt` y `gaussadapt`. Seguidamente se aproxima la integral dada con cada una de estas funciones.

Aproximación con trapecio adaptativa:

```
>>> Q1, err1 = trapecioadapt(f,a,b,n=10)
>>> print('Aproximación: {0}, Error estimado: {1:1.4e}'.format(Q1,err1))
Aproximación: -0.1221226053958866, Error estimado: 5.5173e-09
```

Aproximación con Simpson adaptativa:

```
>>> Q2, err2 = simpsonadapt(f,a,b,n=10)
>>> print('Aproximación: {0}, Error estimado: {1:1.4e}'.format(Q2,err2))
Aproximación: -0.12212260457793833, Error estimado: 3.6047e-09
```

Aproximación con Gauss-Legendre adaptativa:

```
>>> Q3, err3 = gaussadapt(f,a,b,n=10)
>>> print('Aproximación: {0}, Error estimado: {1:1.4e}'.format(Q3,err3))
Aproximación: -0.12212260461883549, Error estimado: 1.3287e-13
```

Para la función `quadrature` utilizamos `miniter=10`, con lo cual exigimos que el orden menor de las cuadraturas Gaussianas sea 10.

```
>>> Q4, err4 = quadrature(f,a,b,miniter=10)
>>> print('Aproximación: {0}, Error estimado: {1:1.4e}'.format(Q4,err4))
Aproximación: -0.12212260456550793, Error estimado: 4.8742e-10
```

Note que `gaussadapt` obtiene el menor error estimado. Verifiquemos que efectivamente esto se cumple.

```
>>> print(np.abs(-0.122122604618968-Q1))
7.769185988992788e-10
>>> print(np.abs(-0.122122604618968-Q2))
4.1029665776015634e-11
>>> print(np.abs(-0.122122604618968-Q3))
1.3250511798901243e-13
>>> print(np.abs(-0.122122604618968-Q4))
5.346006970441408e-11
```

De esta forma, `gaussadapt` obtuvo la mejor aproximación de la integral dada con 13 cifras significativas, lo cual es congruente con el error estimado reportado: 1.3287×10^{-13}.

Veamos dos ejemplos que muestran aplicaciones de la integral.

Ejemplo 5.9 (Óptica)**.** Para planificar una habitación para rayos infrarrojos, nos interesa calcular la energía emitida por un cuerpo negro (es decir, un objeto capaz de irradiar todo el espectro a la temperatura ambiente) en el espectro (infrarrojo) comprendido entre 3μm y 14μm de longitud de onda. La solución de este problema se obtiene calculando la integral,

$$E(T) = 2.39 \times 10^{-11} \int_{3\times10^{-4}}^{14\times10^{-4}} \frac{1}{x^5\left(e^{1.432/(Tx)}-1\right)}\,dx$$

que se denomina *Ecuación de Planck* para la energía $E(T)$, donde x es la longitud de onda (en cm) y T es la temperatura (en grados Kelvin) del cuerpo negro. Calculemos el valor de la energía $E(T)$ para T igual a 213K, hasta al menos 10 dígitos significativos exactos. Para ello utilizamos las funciones `gaussadapt` y `quadrature`. En el siguiente Script se calcula la energía $E(T)$.

```
import numpy as np
from scipy.integrate import quadrature
def f_integrando(x,T): return 2.36e-11/(x**5*(np.exp(1.432/(T*x))-1))
T = 213.
f = lambda x: f_integrando(x,T)
a = 3.0e-4
b = 1.4e-3
Q1, err1 = gaussadapt(f,a,b,n=12)
Q2, err2 = quadrature(f,a,b,miniter=12)
print('-----------------------------------------------------')
print('Cuadratura\tEnergía\t\t\tError estimado')
print('-----------------------------------------------------')
print('gaussadapt\t{0:1.16f}\t{1:1.4e}'.format(Q1,err1))
print('quadrature\t{0:1.16f}\t{1:1.4e}'.format(Q2,err2))
print('-----------------------------------------------------')
```

Al ejecutar el Script se obtiene la siguiente tabla.

Cuadratura	$E(T)$	Error estimado
`gaussadapt`	0.0204311376148235	1.5597e-11
`quadrature`	0.0204311376293727	1.4549e-11

Por lo tanto, estimamos que la energía del cuerpo negro es 0.0204311376 a una temperatura de 213K.

Ejemplo 5.10 (Demografía)**.** Consideramos una población de un número P muy grande de individuos. La distribución $N(s)$ de su altura se puede representar mediante una distribución Gaussiana caracterizada por el valor medio $\overline{h}$ de la altura y la desviación estándar σ,

$$N(s) = \frac{P}{\sigma\sqrt{2\pi}} e^{-\left(s-\overline{h}\right)^2/(2\sigma^2)}.$$

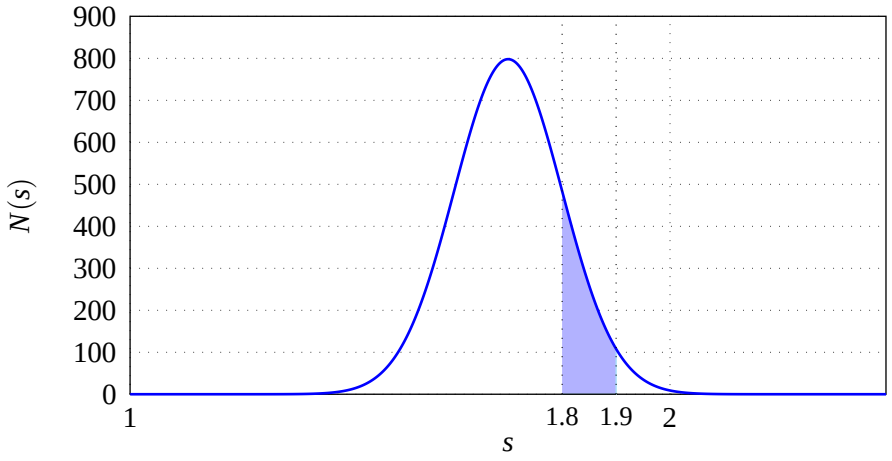


Figura 5.5. Distribución de la altura de una población de $P = 200$ individuos

Entonces,

$$M_{[h,h+\Delta h]} = \int_h^{h+\Delta h} N(s)\, ds \tag{5.24}$$

representa el número de individuos cuya altura está entre h y $h+\Delta h$ (para un Δh positivo). Se proporciona un ejemplo en la Figura 5.5, que corresponde al caso $P = 200$, $\overline{h} = 1.7\,\text{m}$, $\sigma = 0.1\,\text{m}$, y el área de la región sombreada representa el número de individuos cuya altura está en el rango $[1.8, 1.9]\,\text{m}$.

Para calcular el número de individuos cuya altura está entre 1.8 y 1.9 m, necesitamos resolver la integral (5.24) para $h = 1.8$ y $\Delta h = 0.1$. Para esto usamos las funciones `gaussadapt` y `quadrature`. En el siguiente Script se calcula $M_{[1.8,1.9]}$.

```
import numpy as np
from scipy.integrate import quadrature
def f_integrando(x,mu,sigma,P):
    return P*np.exp(-(x-mu)**2/(2*sigma**2))/(sigma*np.sqrt(2*np.pi))
mu = 1.7
sigma = 0.1
P = 200
f = lambda x: f_integrando(x,mu,sigma,P)
a = 1.8
b = 1.9
Q1, err1 = gaussadapt(f,a,b,n=12)
Q2, err2 = quadrature(f,a,b,miniter=12)
print('-------------------------------------------------')
print('Cuadratura\tM[1.8,1.9]\tError estimado')
print('-------------------------------------------------')
print('gaussadapt\t{0:1.6f}\t{1:1.4e}'.format(Q1,err1))
print('quadrature\t{0:1.6f}\t{1:1.4e}'.format(Q2,err2))
print('-------------------------------------------------')
```

Al ejecutar el Script se obtiene la siguiente tabla.

Cuadratura	$M_{[1.8,1.9]}$	Error estimado
`gaussadapt`	27.181024	1.7764e-14
`quadrature`	27.181024	1.4211e-14

Por lo tanto, estimamos que el número de individuos en este rango de altura es 27.1810, que corresponde al 15.39% de todos los individuos.

5.3 Ejercicios

1. Use aproximaciones de diferencias finitas de orden $\mathcal{O}(h^2)$ para calcular $f'(2.36)$ y $f''(2.36)$ a través de la tabla de datos

x	2.36	2.37	2.38	2.39
$f(x)$	0.85866	0.86289	0.86710	0.87129

2. Estime $f'(1)$ y $f''(1)$ utilizando los siguientes datos:

x	0.97	1.00	1.05
$f(x)$	0.85040	0.84147	0.82612

3. Dado los datos

x	$f(x)$
0.84	0.431711
0.92	0.398519
1.00	0.367879
1.08	0.339596
1.16	0.313486

calcule $f''(x)$ con la mayor precisión que usted considere.

4. Use la siguiente tabla de datos para calcular $f'(0.2)$ con la mayor precisión posible.

x	$f(x)$
0	0.000000
0.1	0.078348
0.2	0.138910
0.3	0.192916
0.4	0.244981

5. Emplee el polinomio interpolante para calcular $f'(x)$ y $f''(x)$ en $x = 0$, usando los datos

x	-2.2	-0.3	0.8	1.9
$f(x)$	15.180	10.962	1.920	-2.040

6. Compruebe que la siguiente cuadratura es exacta para polinomios de grado ≤ 4:

$$\int_0^1 f(x)\,dx \approx \frac{1}{90}[7f(0)+32f(1/4)+12f(1/2) +32f(3/4)+7f(1)]$$

7. Calcule un valor aproximado de $\ln 2$ aplicando la fórmula del Problema 6 a:

$$\int_0^1 \frac{1}{t+1}\,dt.$$

Compare su respuesta con el valor correcto y calcule el error.

8. Deduzca una cuadratura para aproximar la integral

$$\int_1^3 f(x)\,dx,$$

en términos de $f(0)$, $f(2)$, $f(4)$. Esta expresión deberá proporcionar resultados exactos para todo polinomio de grado ≤ 2.

9. Determine los valores de los pesos ω_0, ω_1 y ω_2 que hagan que la cuadratura

$$\int_0^2 xf(x)\,dx \approx \omega_0 f(0)+\omega_1 f(1)+\omega_2 f(2),$$

sea exacta para todos los polinomios de grado tan alto como sea posible. ¿Cuál es el grado máximo?

10. La regla del punto medio sobre el intervalo $[x_{i-1}, x_{i+1}]$ se expresa como:

$$\int_{x_{i-1}}^{x_{i+1}} f(x)\,dx \approx (x_{i-1}, x_{i+1})\, f(x_i).$$

Determine la regla del punto medio compuesta sobre el intervalo $[a,b]$, usando un espaciamiento uniforme con $h = (b-a)/n$, tal que $x_i = a + ih$ para $i = 0, 1, 2, \ldots, n$ (con n par).

11. ¿Cuál de las siguientes dos cuadraturas considera usted es mejor?

$$\begin{aligned} \int_0^1 f(x)\,dx &= af(0) + bf(1/2) + cf(1) \\ \int_0^1 f(x)\,dx &= \alpha f(1/4) + \beta f(1/2) + \gamma f(3/4). \end{aligned}$$

12. ¿Existe una expresión de la cuadratura

$$\int_0^1 f(x)\,dx \approx \alpha[f(x_0) + f(x_1)],$$

que sea exacta para todos los polinomios cuadráticos?

13. Use la regla del trapecio compuesta para evaluar la integral

$$\int_0^{\pi/4} \ln(1 + \tan x)\,dx.$$

Explique los resultados. (Computacional)

14. Utilice la regla de Simpson compuesta para calcular la integral

$$\int_0^{10^{-2}} \left(\frac{\operatorname{sen} x}{x}\right) dx,$$

con una precisión de siete decimales. (Computacional)

6 Cálculo de Autovalores y Autovectores

Un problema importante en Álgebra Lineal es el cálculo de los autovalores y autovectores de una matriz. Problemas de autovalores aparecen con mucha frecuencia en distintas áreas de la ciencia y la ingeniería. Por ejemplo, los modos naturales y las frecuencias de vibración de una estructura están determinados por los autovectores y autovalores, respectivamente, de una matriz apropiada. La estabilidad de la estructura está determinada por los autovalores y, por lo tanto, su cálculo es de sumo interés. También los autovalores son útiles para analizar métodos numéricos, por ejemplo, análisis de convergencia de métodos iterativos para resolver sistemas de ecuaciones algebraicas y análisis de estabilidad de métodos para resolver ecuaciones diferenciales.

En este capítulo nos limitaremos al estudio del cálculo de los autovalores y autovectores de matrices reales. Comenzaremos describiendo conceptos básicos y resultados útiles, que nos permitirán describir los métodos iterativos para aproximar los autovalores y autovectores. Seguidamente nos centraremos en el estudio del cálculo de los autovalores de matrices simétricas. En esta parte describiremos el método de las potencias y sus variantes, y el método *QR* y algunas de sus variantes. Finalizamos el capítulo presentando algunas extensiones al caso no simétrico.

6.1 Propiedades Básicas y Algunos Resultados Útiles

La forma estándar de un problema de autovalores es

$$A\mathbf{x} = \lambda\mathbf{x}, \tag{6.1}$$

donde A es una matriz real de orden $n \times n$ y $\lambda \in \mathbb{C}$. Recordemos que el número $\lambda \in \mathbb{C}$ se llama *autovalor* de A y el vector $\mathbf{x}$ es el *autovector* asociado con el autovalor λ, además, el conjunto de los autovalores de A se denomina *espectro* de A, denotado por $\sigma(A)$.

A continuación mostramos resultados básicos que se pueden encontrar en cualquier libro de Álgebra Lineal. Dejaremos que el lector busque las demostraciones. Los primeros son resultados simples y pueden ser demostrados como ejercicio.

Teorema 6.1. *Si λ es un autovalor de una matriz $A \in \mathbb{R}^{n\times n}$ y $\mathbf{v}$ el autovector asociado de λ, entonces $(\alpha\lambda + \beta)$ es un autovalor de la matriz $(\alpha A + \beta I)$ con $\mathbf{v}$ su autovector asociado. Si $\mathbf{v}$ es un autovector de A, entonces $c\mathbf{v}$ también lo es para toda constante $c \neq 0$.*

Teorema 6.2. *Si λ es un autovalor de una matriz $A \in \mathbb{R}^{n\times n}$ no singular y $\mathbf{v}$ es el autovector asociado de λ, entonces λ^{-1} es un autovalor de A^{-1} con $\mathbf{v}$ su autovector asociado.*

Muchas técnicas para el cálculo de autovalores se basan en la transformación de la matriz original en alguna otra estrechamente relacionada, a la cual se le puedan calcular más fácilmente los autovalores. Las principales transformaciones usadas para este fin están dadas en la siguiente definición.

Definición 6.1. Sea $T \in \mathbb{R}^{n\times n}$ una matriz no singular. La transformación

$$B = TAT^{-1}$$

se denomina *transformación de similaridad*. Las matrices A y B se dicen *similares*.

Teorema 6.3. *Sean λ un autovalor de una matriz $A \in \mathbb{R}^{n\times n}$ y $\mathbf{v}$ su autovector asociado. Sea $T \in \mathbb{R}^{n\times n}$ una matriz no singular. Entonces, λ es un autovalor de la matriz $B = TAT^{-1}$, con $T\mathbf{v}$ su autovector asociado.*

Un resultado importante, llamado *Teorema Alternativo*, conecta la resolución de una sistema lineal $A\mathbf{x} = \mathbf{b}$ con el espectro de A.

Teorema 6.4 (Teorema alternativo)**.** *Sean $A \in \mathbb{R}^{n\times n}$ y $\alpha \in \mathbb{R}$. El sistema $(A-\alpha I)\mathbf{x} = \mathbf{y}$ tiene una única solución si, y sólo si α no pertenece al espectro de A.*

El Teorema 6.4 nos dice que si $\lambda \in \sigma(A)$, entonces la matriz $(A-\lambda I)$ es singular; inversamente, si $(A-\lambda I)$ es singular, entonces $\lambda \in \sigma(A)$. Esto implica que el espectro de A es el conjunto de todos los $\lambda \in \mathbb{C}$ tales que satisfacen la *ecuación característica*

$$p_A(\lambda) = \det(A-\lambda I) = 0, \tag{6.2}$$

donde

$$p_A(\lambda) = a_0 + a_1\lambda + a_2\lambda^2 + \cdots + a_n\lambda^n$$

es el *polinomio característico*. Como éste último es un polinomio de grado n con respecto a λ, existen n autovalores de A no necesariamente distintos. Ahora, como A es una matriz real, $p_A(\lambda)$ es un polinomio con coeficientes reales; por lo tanto, los autovalores complejos de A aparecen en pares complejos conjugados. Entonces, el problema de encontrar los autovalores de una matriz se reduce al cálculo de las raíces de un polinomio. En cambio, para hallar los autovectores asociados, es necesario resolver sistemas de ecuaciones lineales.

Ejemplo 6.1. A manera de ejemplo, consideremos la matriz

$$A = \begin{bmatrix} 1 & -1 & 0 \\ -1 & 1 & -1 \\ 0 & -1 & 1 \end{bmatrix}.$$

El polinomio característico de A es

$$p_A(\lambda) = \det(A-\lambda I) = \begin{vmatrix} 1-\lambda & -1 & 0 \\ -1 & 1-\lambda & -1 \\ 0 & -1 & 1-\lambda \end{vmatrix} = -1-\lambda+3\lambda^2-\lambda^3.$$

Los autovalores de A son las raíces de $p_A(\lambda)$, las cuales son: $\lambda_1 = 1$, $\lambda_2 = 1+\sqrt{2}$ y $\lambda_3 = 1-\sqrt{2}$. Ahora, para calcular el autovector asociado con λ_1, sustituimos $\lambda = \lambda_1$ en la Ecuación (6.1), obteniendo

$$\begin{bmatrix} 1 & -1 & 0 \\ -1 & 1 & -1 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \Leftrightarrow \quad \begin{cases} x_1 - x_2 &= x_1 \\ -x_1 + x_2 - x_3 &= x_2 \\ -x_2 + x_3 &= x_3 \end{cases}$$

Este último sistema tiene infinitas soluciones de la forma:

$$\mathbf{x} = \begin{bmatrix} \alpha \\ 0 \\ -\alpha \end{bmatrix} = \alpha \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}.$$

Así el autovector asociado a λ_1 es

$$\mathbf{x}_1 = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}.$$

Por un procedimiento similar se obtienen los otros dos autovectores:

$$\mathbf{x}_2 = \begin{bmatrix} \frac{1}{2} \\ -\frac{\sqrt{2}}{2} \\ \frac{1}{2} \end{bmatrix}, \qquad \mathbf{x}_3 = \begin{bmatrix} \frac{1}{2} \\ \frac{\sqrt{2}}{2} \\ \frac{1}{2} \end{bmatrix}.$$

■

Los siguientes teoremas establecen algunas características de los autovalores y autovectores de matrices simétricas.

Teorema 6.5. *Los autovalores de una matriz real simétrica son todos reales. A cada autovalor le corresponde un autovector real distinto.*

Teorema 6.6. *Si $A \in \mathbb{R}^{n\times n}$ es una matriz simétrica, entonces existe un conjunto de autovectores ortonormales $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$, esto es, $\mathbf{v}_i^T \mathbf{v}_i = 1$ para $i = 1, 2, \ldots, n$, y $\mathbf{v}_i^T \mathbf{v}_j = 0$ para $i \neq j$.*

Como un conjunto de vectores ortogonales son linealmente independientes, entonces del Teorema 6.6 se sigue que todo vector **x** puede expresarse como una combinación lineal de los autovectores de una matriz simétrica. Este simple resultado no es cierto para matrices no simétricas, pero es de gran ayuda para trabajar con matrices simétricas. Observe que la matriz A del Ejemplo 6.1 es simétrica, en consecuencia, todos sus autovalores son reales y sus autovectores asociados son vectores reales ortogonales (se deja como ejercicio para el lector comprobar esta afirmación).

El cálculo de autovalores usando el polinomio característico no es práctico, ya que requiere la aproximación de las raíces de un polinomio, que es computacionalmente costoso cuando el grado del polinomio es grande. Lo recomendable para ello es utilizar un método iterativo. Comenzaremos calculando los autovalores de matrices simétricas, ya que estas matrices aparecen con mucha regularidad en aplicaciones en ingeniería, por ello el cálculo de autovalores de matrices simétricas recibe gran atención. A continuación describiremos el *método de las potencias* y sus variantes: *método de las potencias inverso, método de las potencias inverso con desplazamiento* y el *método del cociente de Rayleigh*. También describiremos el *método QR* y algunas de sus variantes.

6.2 Método de las Potencias y sus Variantes

Existen aplicaciones que requieren del cálculo de ciertos autovalores y autovectores, generalmente se calculan los autovalores más grandes y los más pequeños, en módulo. En tales casos,

se puede utilizar el método de las potencias o una de sus variantes, cuyas implementaciones son relativamente sencillas.

El *cociente de Rayleigh* es una función que juega un rol muy importante en el cálculo de autovalores.

Definición 6.2. Dada $A \in \mathbb{R}^{n\times n}$, el cociente de Rayleigh de $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{x} \neq \mathbf{0}$ es la función $r : \mathbb{R}^n \to \mathbb{R}$ dada por

$$r(\mathbf{x}) = \frac{\mathbf{x}^T A\mathbf{x}}{\mathbf{x}^T\mathbf{x}}.$$

Algunas propiedades del cociente de Rayleigh son las siguentes:

i) Si $\mathbf{x}$ es un autovector de A con λ su autovalor asociado, entonces $r(\mathbf{x}) = \lambda$.

ii) Si A es una matriz simétrica con $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$, entonces

$$\lambda_1 \geq r(\mathbf{x}) \geq \lambda_n,$$

para todo vector $\mathbf{x} \neq 0$; además, $r(\mathbf{v}_i) = \lambda_i$, para todo $i = 1,2,\ldots,n$, donde $\mathbf{v}_i$ es el autovector asociado a λ_i.

Sea $A \in \mathbb{R}^{n\times n}$ una matriz simétrica con autovectores $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ y sus correspondientes autovalores $\lambda_1, \lambda_2, \ldots, \lambda_n$ tales que

$$|\lambda_1| \geq |\lambda_2| \geq \cdots \geq |\lambda_n|,$$

esto es, $A\mathbf{v}_i = \lambda_i\mathbf{v}_i$ para $i = 1,2,\ldots,n$ y λ_1 es el autovalor dominante de A. Deduzcamos la base fundamental del método de las potencias.

Para un vector $\mathbf{x}$, por el Teorema 6.6, éste puede expresarse como una combinación lineal de los autovectores $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$,

$$\mathbf{x} = \sum_{i=1}^{n} c_i\mathbf{v}_i,$$

además, como $\mathbf{v}_i^T\mathbf{v}_i = 1$ y $\mathbf{v}_i^T\mathbf{v}_j = 0$ para $i \neq j$, se tiene

$$c_j = \mathbf{v}_j^T\mathbf{x}, \qquad \text{para } j = 1,2,\ldots,n.$$

Luego,

$$A\mathbf{x} = \sum_{i=1}^{n} c_i A\mathbf{v}_i = \sum_{i=1}^{n} c_i\lambda_i\mathbf{v}_i,$$

de donde se deduce que

$$\begin{aligned} A^k\mathbf{x} &= \sum_{i=1}^{n} c_i\lambda_i^k\mathbf{v}_i \\ &= \lambda_1^k \sum_{i=1}^{n} c_i \left(\frac{\lambda_i}{\lambda_1}\right)^k \mathbf{v}_i, \end{aligned} \tag{6.3}$$

lo cual implica

$$\|A^k\mathbf{x}\|_2 = \lambda_1^k \left(\sum_{i=1}^{n} c_i^2 \left(\frac{\lambda_i}{\lambda_1}\right)^{2k}\right)^{1/2}. \tag{6.4}$$

Como $(\lambda_i/\lambda_1)^k \to 0$ cuando $k \to \infty$, para $i = 2,3,\ldots,n$, entonces por (6.3) y (6.4), podemos escribir:

$$\lim_{k\to\infty} \frac{A^k\mathbf{x}}{\|A^k\mathbf{x}\|_2} = \lim_{k\to\infty} \frac{c_1v_1 + \sum_{i=2}^{n} c_i \left(\frac{\lambda_i}{\lambda_1}\right)^k \mathbf{v}_i}{\left(c_1^2 + \sum_{i=2}^{n} c_i^2 \left(\frac{\lambda_i}{\lambda_1}\right)^{2k}\right)^{1/2}} = \mathbf{v}_1. \tag{6.5}$$

Luego, el valor de λ_1 lo obtenemos usando el cociente de Rayleight de $\mathbf{v}_1$, esto es,

$$\lambda_1 = \frac{\mathbf{v}_1^T A\mathbf{v}_1}{\mathbf{v}_1^T\mathbf{v}_1} = \mathbf{v}_1^T A\mathbf{v}_1.$$

La ecuación (6.5) es la base fundamental del método de las potencias, cuyo pseudocódigo se muestra en el Algoritmo 6.1. La implementación del método de las potencias se muestra en el Programa 6.1, la cual hemos denominado `metpot`.

Algoritmo 6.1 (Método de las potencias)**.**

Entrada: $A \in \mathbb{R}^{n\times n}$ simétrica y $\mathbf{x}_0 \in \mathbb{R}^n$.

1: **for** $k = 1,2,\ldots$ **do**
2: $\quad \mathbf{y}_k = A\mathbf{x}_{k-1}$ ▷ Aplicar A
3: $\quad \mathbf{x}_k = \mathbf{y}_k/\|\mathbf{y}_k\|_2$ ▷ Normalizar
4: $\quad \beta_k = \mathbf{x}_k^T A\mathbf{x}_k$ ▷ Cociente de Rayleigh
5: **end for**

Observe que $\beta_k = \mathbf{x}_k^T A\mathbf{x}_k$ es el cociente de Rayleight evaluado en el vector $\mathbf{x}_k$, ya que $\mathbf{x}_k^T\mathbf{x}_k = \|\mathbf{x}_k\|_2 = 1$. El método de las potencias aproxima el autovalor y el autovector dominantes de A. Generalmente, para el método de las potencias y sus variantes, se utiliza el siguiente criterio de parada:

$$|\beta_k - \beta_{k-1}| < \varepsilon,$$

donde $0 < \varepsilon < 1$ es una tolerancia dada.

Programa 6.1. Método de las potencias

```
def metpot(A, x0, tol=1.0e-6, maxit=1000):
    # A es una matriz simétrica nxn
    A = np.asarray(A,dtype=float)
    np.copy(np.asarray(x0,dtype=float))
    k = 0
    y_k = A.dot(x_k)
    beta_k1 = (x_k.dot(y_k))/(x_k.dot(x_k))
    while True:
        x_k  = y_k/np.linalg.norm(y_k)
        y_k = A.dot(x_k)
        beta_k = x_k.dot(y_k)
        k += 1
        if np.abs(beta_k-beta_k1)<tol:
            return beta_k, x_k, k
        if k>maxit:
            print('Alcanzó el máximo de iteraciones')
            return beta_k, x_k, k
        beta_k1 = beta_k
```

Utilizando la ecuación (6.5) se demuestra fácilmente el siguiente teorema de convergencia del método de las potencias.

Teorema 6.7. *Sean $A \in \mathbb{R}^{n\times n}$ simétrica, $\mathbf{x}_0 \in \mathbb{R}^n$ y $\{\mathbf{x}_k\}$ la sucesión generada por el Algoritmo 6.1. Entonces, si $\mathbf{v}_1^T\mathbf{x}_0 \neq 0$ se cumple:*

$$\lim_{k\to\infty} \beta_k = \lambda_1 \qquad y \qquad \lim_{k\to\infty} \mathbf{x}_k = \mathbf{v}_1.$$

La condición $\mathbf{v}_1^T\mathbf{x}_0 \neq 0$ es necesaria para garantizar la convergencia del Algoritmo 6.1, pero es claro que no puede ser verificada, ya que $\mathbf{v}_1$ no es conocido. Si $\mathbf{x}_0$ se escoge aleatoriamente, entonces la probabilidad de que se satisfaga tal condición es 1.

Ejemplo 6.2. Apliquemos el método de las potencias a la matriz

$$A = \begin{bmatrix} 1 & 2 & 0 \\ 2 & 3 & 5 \\ 0 & 5 & 4 \end{bmatrix},$$

tomando como iterado inicial al vector $\mathbf{x}_0 = (1, \frac{1}{2}, 1)$, para hallar el autovalor y el autovector dominantes de A. Conocemos que los autovalores de A son: $\lambda_1 = 8.76348772$, $\lambda_2 = -2.24048352$ y $\lambda_3 = 1.4769958$, además, el autovector dominante normalizado es

$$\mathbf{v}_1 = \begin{bmatrix} 0.17495651 \\ 0.67913638 \\ 0.71285623 \end{bmatrix}.$$

Mostremos las primeras 5 iteraciones de la función `metpot`.

$k = 0$:

$$\mathbf{x}_0 = \begin{bmatrix} 1, \\ 0.5 \\ 1 \end{bmatrix}, \quad \beta_0 = \frac{\mathbf{x}_0^T A\mathbf{x}_0}{\mathbf{x}_0^T\mathbf{x}_0} = 0.107594937, \quad \mathbf{y}_0 = A\mathbf{x}_0 = \begin{bmatrix} 2 \\ 8.5 \\ 6.5 \end{bmatrix},$$

$k = 1$:

$$\mathbf{x}_1 = \frac{\mathbf{y}_1}{\|\mathbf{y}_1\|_2} = \begin{bmatrix} 0.18372608 \\ 0.78083586 \\ 0.59710978 \end{bmatrix}, \quad \beta_1 = \mathbf{x}_1^T A\mathbf{x}_1 = 8.52531646, \quad \mathbf{y}_1 = A\mathbf{x}_1 = \begin{bmatrix} 2 \\ 8.5 \\ 6.5 \end{bmatrix}$$

$k = 2$:

$$\mathbf{x}_2 = \frac{\mathbf{y}_1}{\|\mathbf{y}_1\|_2} = \begin{bmatrix} 0.20143072 \\ 0.65730024 \\ 0.72621075 \end{bmatrix}, \quad \beta_2 = \mathbf{x}_2^T A\mathbf{x}_2 = 8.74922026, \quad \mathbf{y}_2 = A\mathbf{x}_2 = \begin{bmatrix} 1.5160312 \\ 6.00581593 \\ 6.19134422 \end{bmatrix}$$

$k = 3$:

$$\mathbf{x}_3 = \frac{\mathbf{y}_2}{\|\mathbf{y}_2\|_2} = \begin{bmatrix} 0.1731043 \\ 0.68575934 \\ 0.70694344 \end{bmatrix}, \quad \beta_3 = \mathbf{x}_3^T A\mathbf{x}_3 = 8.76260107, \quad \mathbf{y}_3 = A\mathbf{x}_3 = \begin{bmatrix} 1.54462299 \\ 5.93820382 \\ 6.25657047 \end{bmatrix}$$

$k = 4$:

$$\mathbf{x}_4 = \frac{\mathbf{y}_3}{\|\mathbf{y}_3\|_2} = \begin{bmatrix} 0.17626344 \\ 0.67763347 \\ 0.71396364 \end{bmatrix}, \quad \beta_4 = \mathbf{x}_4^T A\mathbf{x}_4 = 8.76343101, \quad \mathbf{y}_4 = A\mathbf{x}_4 = \begin{bmatrix} 1.53153039 \\ 5.95524549 \\ 6.24402192 \end{bmatrix}$$

$k = 5$:

$$\mathbf{x}_5 = \frac{\mathbf{y}_4}{\|\mathbf{y}_4\|_2} = \begin{bmatrix} 0.17476308 \\ 0.67955364 \\ 0.71250594 \end{bmatrix}, \quad \beta_5 = \mathbf{x}_5^T A\mathbf{x}_5 = 8.76348406, \quad \mathbf{y}_5 = A\mathbf{x}_5 = \begin{bmatrix} 1.53387036 \\ 5.95071679 \\ 6.24779196 \end{bmatrix}$$

Claramente se observa que la sucesión $\{\beta_k\}$ converge al autovalor dominante de A y la sucesión $\{\mathbf{x}_k\}$ converge al autovector dominante normalizado. Ahora, mostremos el uso de la función `metpot` y la aproximación obtenida por el mismo.

```
>>> A = np.array([[1,2,0],[2,3,5],[0,5,4]])
>>> x0 = np.array([1,.5,1])
>>> l1,v1,k = metpot(A, x0, tol=1e-8)
>>> print(l1)
8.76348771799
>>> print(v1)
[ 0.17495549  0.67913811  0.71285482]
>>> print(k)
9
```

Se observa que `metpot` obtuvo el escalar 8.76348771799 como la aproximación del autovalor dominante λ_1 y al vector $(0.17495549, 0.67913811, 0.71285482)^T$ como la aproximación del autovector dominante $\mathbf{v}_1$, en 9 iteraciones.

■

En algunas aplicaciones en ingeniería es necesario hallar, por ejemplo, los tres primeros autovalores de mayor magnitud λ_1, λ_2 y λ_3. Uno puede utilizar el método de las potencias conjuntamente con la técnica de *deflación* (reducción) para hallar tales autovalores. Se procede de la siguiente manera. Una vez hallado el autovalor λ_1 y el autovector $\mathbf{v}_1$, podemos reducir la matriz original por

$$A_1 = A - \lambda_1 \mathbf{v}_1 \mathbf{v}_1^T.$$

Como es fácil demostrar que el espectro de A_1 es $\{\lambda_2, \lambda_3, \ldots, \lambda_n, 0\}$, entonces podemos aplicar el método de las potencias a la matriz A_1 para hallar λ_2 y $\mathbf{v}_2$. Ahora reducimos la matriz A_1,

$$A_2 = A_1 - \lambda_2 \mathbf{v}_2 \mathbf{v}_2^T.$$

Aplicando el método de las potencias a la matriz A_2 se hallan el autovalor λ_3 y el autovector $\mathbf{v}_3$. Veamos un ejemplo.

Ejemplo 6.3. Consideremos la matriz A del Ejemplo 6.2. El autovalor y autovector dominantes de A son: $\lambda_1 = 8.76348772$ y $\mathbf{v}_1 = (0.17495651, 0.67913638, 0.71285623)$. Para crear la matriz $A_1 = A - \lambda_1 \mathbf{v}_1 \mathbf{v}_1^T$ usamos la siguiente función `prod_vv`, que construye la matriz $\mathbf{v}_1 \mathbf{v}_1^T$.

```
def prod_vv(v):
    v = np.asarray(v,dtype=float)
    n = len(v)
    vv = np.empty([n,n])
    for i in range(n):
        vv[i] = v[i]*v
    return vv
```

Apliquemos el método de las potencias a la matriz A_1 con iterado inicial $\mathbf{x}_0 = (0, .5, 0)^T$, para hallar λ_2 y $\mathbf{v}_2$.

```
>>> A = np.array([[1,2,0],[2,3,5],[0,5,4]])
>>> l1 = 8.76348772
>>> v1 = np.array([0.17495651,0.67913638,0.71285623])
>>> A1 = A-l1*prod_vv(v1)
>>> x0 = np.array([0,.5,0])
>>> l2,v2,k = metpot(A1, x0)
>>> print(l2)
-2.24048351337
>>> print(v2)
[-0.43391784  0.70310325 -0.56334814]
>>> print(k)
22
```

Ahora apliquemos el método de las potencias a la matriz $A_2 = A_1 - \lambda_2 \mathbf{v}_2 \mathbf{v}_2^T$ con iterado inicial $\mathbf{x}_0 = (-1, 0, .5)^T$, para hallar λ_3 y $\mathbf{v}_3$.

```
>>> A2 = A1-l2*prod_vv(v2)
>>> x0 = np.array([-1,0,.5])
>>> l3,v3,k = metpot(A2, x0, tol=1e-8)
>>> print(l3)
1.47699580922
>>> print(v3)
[-0.88376755 -0.21081503  0.41774626]
>>> print(k)
2
```

Se observa que efectivamente se obtuvieron aproximaciones de los autovalores λ_2 y λ_3, y de sus autovectores asociados $\mathbf{v}_2$ y $\mathbf{v}_3$.

6.2.1 Método de las Potencias Inverso

Sea $A \in \mathbb{R}^{n \times n}$ una matriz simétrica con autovectores $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ y supongamos que sus correspondientes autovalores $\lambda_1, \lambda_2, \ldots, \lambda_n$ satisfacen:

$$|\lambda_1| \geq |\lambda_2| \geq \cdots \geq |\lambda_n| > 0.$$

Del Teorema 6.2 se sigue que $1/\lambda_i$ es un autovalor de la matriz A^{-1}. El objetivo del *método de las potencias inverso* es encontrar el (único) autovalor más pequeño (en magnitud) λ_n, aplicando el método de las potencias a la matriz A^{-1} para hallar el autovalor y el autovector dominantes de A^{-1}, $1/\lambda_n$ y $\mathbf{v}_n$.

El pseudocódigo del método de las potencias inverso se muestra en el Algoritmo 6.2 y su implementación se aprecia en el Programa 6.2, la cual hemos denominado `metpotinv`.

Algoritmo 6.2 (Método de las potencias inverso)**.**

Entrada: $A \in \mathbb{R}^{n\times n}$ simétrica y $\mathbf{x}_0 \in \mathbb{R}^n$.

1: **for** $k = 1, 2, \ldots$ **do**
2: Resolver $A\mathbf{y}_k = \mathbf{x}_{k-1}$ ▷ Aplicar A^{-1}
3: $\mathbf{x}_k = \mathbf{y}_k / \|\mathbf{y}_k\|_2$ ▷ Normalizar
4: $\beta_k = \mathbf{x}_k^T A \mathbf{x}_k$ ▷ Cociente de Rayleigh
5: **end for**

Programa 6.2. Método de las potencias inverso

```
def metpotinv(A, x0, tol=1.0e-6, maxit=1000):
    # A es una matriz simétrica nxn
    A = np.asarray(A,dtype=float)
    x_k = np.copy(np.asarray(x0,dtype=float))
    k = 0
    beta_k1 = 0
    while True:
        y_k = np.linalg.solve(A,x_k)
        x_k  = y_k/np.linalg.norm(y_k)
        beta_k = x_k.dot(A.dot(x_k))
        k += 1
        if np.abs(beta_k-beta_k1)<tol:
            return beta_k, x_k, k
        if k>maxit:
            print('Alcanzó el máximo de iteraciones')
            return beta_k, x_k, k
        beta_k1 = beta_k
```

Observe que en el Algoritmo 6.2 se resuelve el sistema $A\mathbf{y}_k = \mathbf{x}_{k-1}$ para hallar $\mathbf{y}_k$, en lugar de utilizar la inversa de A. La matriz A del sistema lineal es la misma en cada iteración. En este caso, conviene obtener una factorización de A (LU, Cholesky, QR) antes de empezar el proceso iterativo. Es de destacar que el método de las potencias inverso es una variante del método de las potencias, por ello la convergencia establecida en el Teorema 6.7 aplica con los cambios obvios necesarios al Algoritmo 6.2.

Ejemplo 6.4. Consideremos la matriz A del Ejemplo 6.2. Se tiene que $\lambda_3 = 1.4769958$ y $\mathbf{v}_3 = (-0.88378816, -0.21078162, 0.41771952)$. Apliquemos el método de las potencias inverso con iterado inicial $\mathbf{x}_0 = (-1, 0, 0)$.

```
>>> A = np.array([[1,2,0],[2,3,5],[0,5,4]])
>>> x0 = np.array([-1,0,0])
>>> l3,v3,k = metpotinv(A, x0)
>>> print(l3)
1.47699579945
>>> print(v3)
[-0.88377349  -0.21080539   0.41773856]
>>> print(k)
23
```

Se observa que `metpotinv` obtuvo el escalar 1.47699579945 como la aproximación de λ_3 y al vector $(-0.88377349, -0.21080539, 0.41773856)$ como la aproximación del autovector $\mathbf{v}_3$, en 23 iteraciones.

6.2.2 Método de las Potencias Inverso con Desplazamiento

Pasemos a estudiar el uso de desplazamientos en el método de las potencias inverso para acelerar su convergencia, y además poder aproximar autovalores extremos o intermedios. Sea $\sigma \in \mathbb{R}$ (desplazamiento). Si λ es un autovalor de A con $\mathbf{v}$ su autovector asociado, entonces por el Teorema 6.1 $(\lambda - \sigma)$ es un autovalor de la matriz $(A - \sigma I)$ con el mismo autovector asociado. De lo anterior se infiere que $(\lambda - \sigma)^{-1}$ es un autovalor de $(A - \sigma I)^{-1}$ con $\mathbf{v}$ su autovector asociado. Este hecho permite acelerar la convergencia del método de las potencias inverso. En efecto, si $\sigma \approx \lambda_j$, para algún autovalor λ_j de A, entonces es mucho más grande en módulo que $(\lambda_i - \sigma)^{-1}$, para $i \neq j$; y por tanto si aplicamos el método de las potencias sobre $(A - \sigma I)^{-1}$ observaremos una convergencia muy rápida a $\mathbf{v}_j$ y al autovalor λ_j.

El pseudocódigo del método de las potencias inverso se muestra en el Algoritmo 6.3 y su implementación se aprecia en el Programa 6.3.

Algoritmo 6.3 (Método de las potencias inverso con desplazamiento)**.**

Entrada: $A \in \mathbb{R}^{n\times n}$ simétrica, $\mathbf{x}_0 \in \mathbb{R}^n$ y $\sigma \in \mathbb{R}$.

1: **for** $k = 1, 2, \ldots$ **do**
2: Resolver $(A - \sigma I)\mathbf{y}_k = \mathbf{x}_{k-1}$ ▷ Aplicar $(A - \sigma I)^{-1}$
3: $\mathbf{x}_k = \mathbf{y}_k / \|\mathbf{y}_k\|_2$ ▷ Normalizar
4: $\beta_k = \mathbf{x}_k^T A \mathbf{x}_k$ ▷ Cociente de Rayleigh
5: **end for**

Aprecie que en el Algoritmo 6.3 se resuelve el sistema $(A - \sigma I)\mathbf{y}_k = \mathbf{x}_{k-1}$ para hallar $\mathbf{y}_k$, en lugar de utilizar la inversa de $(A - \sigma I)$. La matriz $(A - \sigma I)$ del sistema lineal es la misma en cada iteración. En este caso, conviene obtener una factorización de $(A - \sigma I)$ (LU, Cholesky, QR) antes de empezar el proceso iterativo. Note también que el método de las potencias inverso con desplazamiento es una variante del método de las potencias, por ello la convergencia establecida en el Teorema 6.7 aplica con los cambios obvios necesarios al Algoritmo 6.3.

Programa 6.3. Método de las potencias inverso con desplazamiento

```
def metpotinv_des(A, x0, sigma, tol=1.0e-6, maxit=1000):
    # A es una matriz simétrica nxn
    A = np.asarray(A,dtype=float)
    x_k = np.copy(np.asarray(x0,dtype=float))
    k = 0
    beta_k1 = 0
    M = A-sigma*np.eye(A.shape[0])
    while True:
        y_k = np.linalg.solve(M,x_k)
        x_k  = y_k/np.linalg.norm(y_k)
        beta_k = x_k.dot(A.dot(x_k))
        k += 1
        if np.abs(beta_k-beta_k1)<tol:
            return beta_k, x_k, k
        if k>maxit:
            print('Alcanzó el máximo de iteraciones')
            return beta_k, x_k, k
        beta_k1 = beta_k
```

Ejemplo 6.5. Consideremos la matriz A del Ejemplo 6.2. Se tiene que $\lambda_3 = 1.4769958$ y $\mathbf{v}_3 = (-0.88378816, -0.21078162, 0.41771952)$. Apliquemos el método de las potencias inverso con desplazamiento, tomando $\mathbf{x}_0 = (-1,0,0)$ iterado inicial y $\sigma = 1$.

```
>>> A = np.array([[1,2,0],[2,3,5],[0,5,4]])
>>> x0 = np.array([-1,0,0])
>>> l3,v3,k = metpotinv(A, x0)
>>> sigma = 1
>>> l3,v3,k = cn.metpotinv_des(A,x0,sigma,tol=1e-8)
>>> print(l3)
>>> 1.476995803601659
>>> print(v3)
[-0.88379033  -0.21077812   0.4177167 ]
>>> print(k)
6
```

Se observa que `metpotinv_des` obtuvo el escalar 1.476995803601659 como la aproximación de λ_3 y al vector $(-0.88379033, -0.21077812, 0.4177167)$ como la aproximación del autovector $\mathbf{v}_3$, en solo 6 iteraciones. Se aprecia una aceleración de la convergencia con respecto a los resultados del Ejemplo 6.4.

■

6.2.3 Método del Cociente de Rayleigh

La introducción del desplazamiento en el método de las potencias inverso introduce efectos positivos. Tomar un desplazamiento que aproxima el autovalor buscado ocasiona un aceleración en la velocidad de convergencia, lo cual motiva actualizar ese desplazamiento en cada iteración. Para ello se usa el cociente de Rayleigh. Esta variante dinámica se conoce como el *método del cociente de Rayleigh*.

Algoritmo 6.4 (Método del cociente de Rayleigh)**.**

Entrada: $A \in \mathbb{R}^{n\times n}$ simétrica y $\mathbf{x}_0 \in \mathbb{R}^n$.

1: Asignar $\sigma_0 = (\mathbf{x}_0^T A\mathbf{x}_0)/(\mathbf{x}_0^T\mathbf{x}_0)$
2: **for** $k = 1,2,\ldots$ **do**
3: Resolver $(A-\sigma_k I)\mathbf{y}_k = \mathbf{x}_{k-1}$ ▷ Aplicar $(A-\sigma_k I)^{-1}$
4: $\mathbf{x}_k = \mathbf{y}_k/\|\mathbf{y}_k\|_2$ ▷ Normalizar
5: $\sigma_k = \mathbf{x}_k^T A\mathbf{x}_k$ ▷ Cociente de Rayleigh
6: **end for**

Observe que en el Algoritmo 6.4 no se calcula la inversa de la matriz $(A-\sigma_k I)$, sino que se resuelve el sistema $(A-\sigma_k I)\mathbf{y}_k = \mathbf{x}_{k-1}$ para hallar $\mathbf{y}_k$. La matriz $(A-\sigma_k I)$ cambia en cada iteración y en consecuencia esta variante del método de las potencias es la más costosa computacionalmente de todas ellas. Sin embargo, vale destacar que al usar como desplazamiento dinámico el propio cociente de Rayleigh evaluado en la aproximación dinámica del autovector, la convergencia del método en el caso simétrico es Q-cúbica (ver [6] y las referencias dadas allí). En la práctica, esto se traduce en muy pocas iteraciones para alcanzar una excelente aproximación del par autovalor y autovector. En el Programa 6.3 se muestra la implementación del método del cociente de Rayleigh.

Programa 6.4. Método del cociente de Rayleigh

```
def met_rayleigh(A, x0, tol=1.0e-6, maxit=1000):
    A = np.asarray(A,dtype=float)
    x_k = np.copy(np.asarray(x0,dtype=float))
    k = 0
    sigma_k1 = 0
    sigma = np.dot(x_k, A.dot(x_k))/np.dot(x_k,x_k)
    while True:
        y_k = np.linalg.solve(A-sigma*np.eye(A.shape[0]),x_k)
        x_k  = y_k/np.linalg.norm(y_k)
        sigma = np.dot(x_k,A.dot(x_k))
        k += 1
        if np.abs(sigma-sigma_k1)<tol:
            return sigma, x_k, k
        if k>maxit:
            print('Alcanzó el máximo de iteraciones')
            return sigma, x_k, k
        sigma_k1 = sigma
```

Ejemplo 6.6. Consideremos la matriz A del Ejemplo 6.2. Se tiene que $\lambda_3 = 1.4769958$ y $\mathbf{v}_3 = (-0.88378816, -0.21078162, 0.41771952)$. Apliquemos el método del cociente de Rayleigh, tomando $\mathbf{x}_0 = (-1,-1,0)$ como iterado inicial.

```
>>> A = np.array([[1,2,0],[2,3,5],[0,5,4]])
>>> x0 = np.array([-1,-1,0])
>>> l3,v3,k = met_rayleigh(A,x0)
>>> print(l3)
1.4769958036944009
>>> print(v3)
[-0.88378816 -0.21078162  0.41771952]
>>> print(k)
5
```

Se observa que `met_rayleigh` obtuvo el escalar 1.4769958036944009 como la aproximación de λ_3 y al vector $(-0.88378816, -0.21078162, 0.41771952)$ como la aproximación del autovector $\mathbf{v}_3$, en solo 5 iteraciones.

■

6.2.4 Extensiones al Caso No Simétrico

En esta sección describiremos en forma breve la extensión del método de las potencias y sus variantes, al caso cuando $A \in \mathbb{R}^{n\times n}$ es no simétrica. En este caso, los autovalores de A pueden ser números complejos y, en general, los autovectores no se pueden escoger ortogonales entre si.

Asumamos que la matriz $A \in \mathbb{R}^{n\times n}$ es no simétrica y posee n autovectores linealmente independientes, $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$, todos ellos en $\mathbb{C}^{n\times n}$, con sus correspondientes autovalores $\lambda_i \in \mathbb{C}$ tales que

$$|\lambda_1| \geq |\lambda_2| \geq \cdots \geq |\lambda_n|.$$

Es decir, $A\mathbf{v}_i = \lambda \mathbf{v}_i$, para $i = 1, 2, \ldots, n$.

El método de las potencias (Algoritmo 6.1), el método de las potencias inverso (Algoritmo 6.2), el método de las potencias inverso con desplazamiento (Algoritmo 6.3) y el método del cociente

de Rayleigh (Algoritmo 6.4) permanecen igual que en el caso simétrico, salvo que el cociente de Rayleigh, $r : \mathbb{C}^n \to \mathbb{C}^n$, se define por:

$$r(\mathbf{x}) = \frac{\mathbf{x}^* A\mathbf{x}}{\mathbf{x}^*\mathbf{x}},$$

donde $\mathbf{x}^*$ representa la traspuesta conjugada del vector $\mathbf{x} \in \mathbb{C}^n$. La función $r(\mathbf{x})$ conserva todas sus propiedades; por ello, en los algoritmos citados arriba solo es necesario cambiar la expresión $\mathbf{x}_k^T A\mathbf{x}_k$ por la expresión $\mathbf{x}_k^* A\mathbf{x}_k$ para obtener, de esta forma, su extensión al caso no simétrico.

Con respecto a los posibles desplazamientos (estáticos o dinámicos), si se escogen como números reales entonces acelerarán la convergencia a autovalores reales. Si se usa aritmética punto-flotante compleja, y se escogen desplazamientos complejos, entonces se puede acelerar la convergencia a cualquier autovalor de A. El Teorema 6.7 sigue siendo válido en el caso no simétrico, ya que en su demostración no se usa la ortogonalidad de los autovectores, sino apenas que ellos forman un conjunto linealmente independiente. Por propiedades del cociente de Rayleigh, la convergencia del método del cociente de Rayleigh a algún autovalor particular, en el caso no simétrico, es Q-cuadrática en lugar de Q-cúbica como en el caso simétrico (ver [6] y las referencias dadas allí).

6.3 Método *QR*

En las secciones anteriores estudiamos el método de las potencias y sus variantes, que permiten calcular el mayor y el menor autovalor, en módulo de una matriz simétrica. En algunas aplicaciones en ingeniería es necesario calcular todos los autovalores y quizás también todos los autovectores. El lector pudiera pensar: esto lo puedo hacer aplicando el método de las potencias o algunas de sus variantes, combinado con la técnica de deflación. Efectivamente, esto se puede hacer, pero es excesivamente costoso. En cambio, podemos usar el *método QR*, que es uno de los algoritmos más usados para calcular autovalores de una matriz, debido a su simplicidad y elegancia.

La idea básica del método *QR* es usar iterativamente la factorización *QR* (para detalles ver Capítulo 2), bajo ciertas condiciones, con el propósito de generar una sucesión de transformaciones de similaridad que converge a la factorización de Schur de la matriz. Antes de presentar el método *QR*, introduzcamos algunos resultados de la factorización de Schur.

Definición 6.3. Una *factorización de Schur* de una matriz $A \in \mathbb{C}^{n\times n}$ es una factorización $A = QTQ^*$, donde Q es una matriz ortogonal o unitaria, T es una matriz triangular superior y Q^* es la matriz traspuesta conjugada de Q.

El *Teorema de Schur* establece que toda matriz compleja cuadrada posee una factorización de Schur.

Teorema 6.8 (Teorema de Schur)**.** *Toda matriz cuadrada $A \in \mathbb{C}^{n\times n}$ tiene una factorización de Schur.*

Bajo ciertas condiciones, una matriz real $A \in \mathbb{R}^{n\times n}$ tiene factorización de Schur. Cuando la matriz A es simétrica, su descomposición de Schur adquiere una forma muy particular, la cual se muestra en el siguiente teorema.

Teorema 6.9. *Si $A \in \mathbb{R}^{n\times n}$ es simétrica, entonces existe una matriz ortogonal $Q \in \mathbb{R}^{n\times n}$ y una matriz diagonal $D \in \mathbb{R}^{n\times n}$ tales que $A = QDQ^T$.*

Aún cuando $A \in \mathbb{R}^{n\times n}$ no simétrica tiene entradas reales, ella pudiera tener autovalores complejos en pares conjugados y la matriz Q tiene entradas complejas. Sin embargo, cuando A es una matriz real siempre se puede construir la *factorización de Schur real*, que se muestra en el Teorema 6.10. Este resultado es de importancia, desde un punto de vista práctico, ya que permite trabajar con aritmética punto flotante real.

Teorema 6.10 (Teorema de Schur real)**.** *Sea $A \in \mathbb{R}^{n\times n}$. Entonces existe una matriz ortogonal $Q \in \mathbb{R}^{n\times n}$ y una matriz $T \in \mathbb{R}^{n\times n}$ triangular superior en bloques tales que*

$$Q^T AQ = T = \begin{bmatrix} T_{11} & T_{12} & \cdots & T_{1k} \\ 0 & T_{22} & \cdots & T_{2k} \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & T_{kk} \end{bmatrix},$$

donde cada T_{ii} o es un número real (autovalor de A) o una matriz 2×2, cuyos autovalores son un par conjugado de autovalores complejos de A.

Del Teorema de Schur (versión compleja o real) se infiere que las matrices A y T son similares. En efecto, como toda matriz $A \in \mathbb{C}^{n\times n}$ tiene una factorización de Schur, esto es, $A = QTQ^*$, con $Q \in \mathbb{C}^{n\times n}$ ortogonal y $T \in \mathbb{C}^{n\times n}$ triangular superior, entonces podemos escribir:

$$T = Q^* AQ = Q^{-1} AQ,$$

en otras palabras, A y T son matrices similares. Por el Teorema 6.3, las matrices A y T poseen los mismos autovalores. Además, las entradas de la diagonal principal de T son sus autovalores.

Regresemos al método QR. En el Algoritmo 6.5 se muestra la versión básica del método QR, la cual parece sumamente simple.

Algoritmo 6.5 (Método QR)**.**

Entrada: $A \in \mathbb{C}^{n\times n}$.

1: $A_0 = A$
2: **for** $k = 1, 2, \ldots$ **do**
3: $\quad Q_k R_k = A_{k-1}$ ▷ Factorización QR de A_{k-1}
4: $\quad A_k = R_k Q_k$ ▷ Recombinación de los factores en orden inverso
5: **end for**

Las matrices A_k generadas por el método QR satisfacen que cada una de ellas es similar a la anterior. Verifiquemos tal afirmación. Como $A_{k-1} = Q_k R_k$ y Q_k es ortogonal, entonces $R_k = Q_k^* A_{k-1}$. Luego,

$$A_k = R_k Q_k = Q_k^* A_{k-1} Q_k, \quad \text{para } k = 1, 2, \ldots$$

es decir, A_k y A_{k-1} son matrices similares, para $k \geq 1$. Bajo ciertas hipótesis se puede establecer que la sucesión $\{A_k\}$ converge a la matriz T de la factorización de Schur de A. En el caso que $A \in \mathbb{R}^{n\times n}$ es una matriz real no simétrica, la aplicación del método QR produce una sucesión de matrices que converge a la factorización de Schur real. Para ser precisos, bajo ciertas hipótesis se

puede establecer que la sucesión $\{A_k\}$ generada por el método *QR* converge a la matriz triangular en bloques T de la factorización de Schur real de A. Cuando A es simétrica, la sucesión $\{A_k\}$ generada por el método *QR* converge a la matriz diagonal D de la factorización de Schur (ver [6] y las referencias dadas allí).

Realicemos un conteo de operaciones punto-flotante de la forma básica del método *QR* para matrices reales. En el Algoritmo 6.5 es necesario calcular una factorización *QR* por cada iteración, pero una factorización *QR* requiere de $\mathbf{O}(n^3)$ operaciones punto-flotante. Por tanto, cada iteración de la forma básica del método *QR* tiene un costo computacional de $\mathbf{O}(n^3)$, que lo hace muy costoso para matrices grandes ($n > 1000$).

Existen variantes del método *QR* que reducen significativamente su costo computacional. Una de estas variantes consiste en transformar primero la matriz $A \in \mathbb{R}^{n\times n}$ en una matriz similar H_0 que posea forma de Hessenberg superior, luego aplicar el método *QR* a H_0. La factorización *QR* vista en el Capítulo 2, se puede extender para obtener una matriz $\widehat{Q}$ ortogonal de manera que

$$H_0 = \widehat{Q}^T A \widehat{Q}, \tag{6.6}$$

donde H_0 es una matriz de Hessenberg superior. Es claro ver que A y H_0 son matrices similares; por lo que poseen los mismos autovalores. La idea principal para obtener H_0 es multiplicar por la izquierda de forma sucesiva la matriz A por transformaciones de Householder apropiadas, seguidas de multiplicaciones por la derecha con las traspuestas de las mismas transformaciones. La matriz $\widehat{Q}$ se construye, de ser necesario, como el producto de $(n-2)$ transformaciones de Householder que denotaremos por $\widehat{Q}_j$, $j = 1,2,\ldots,n-2$.

El proceso de obtener la matriz H_0 se denomina *reducción a la forma de Hessenberg*, que seguidamente ilustramos para el caso de $n = 4$.

$$\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix} \quad A$$

$$\xrightarrow{\widehat{Q}_1} \quad \begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix} \quad \widehat{Q}_1 A$$

$$\xrightarrow{\widehat{Q}_1^T} \quad \begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix} \quad \widehat{Q}_1 A \widehat{Q}_1^T$$

$$\xrightarrow{\widehat{Q}_2} \quad \begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix} \quad \widehat{Q}_2(\widehat{Q}_1 A \widehat{Q}_1^T)$$

$$\xrightarrow{\widehat{Q}_2^T} \quad \begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix} \quad H_0 = \widehat{Q}_2(\widehat{Q}_1 A \widehat{Q}_1^T)\widehat{Q}_2^T$$

Observe que los ceros creados por $\widehat{Q}_j$ no se destruyen al postmultiplicar por $\widehat{Q}_j^T$. Esto se repite en los siguientes pasos del proceso. La matriz final $\widehat{Q}$ se obtiene como

$$\widehat{Q} = \widehat{Q}_{n-2}\widehat{Q}_{n-3}\cdots\widehat{Q}_2\widehat{Q}_1.$$

La matriz $\widehat{Q}$ es el producto de $(n-2)$ matrices transformaciones de Householder, luego ella es ortogonal, además, es inmediato verificar que

$$H_0 = \widehat{Q}^T A \widehat{Q}.$$

Como ejercicio interesante para el lector, se deja modificar la función `hess` para obtener la matriz $\widehat{Q}$.

La implementación de la reducción a la forma de Hessenberg se muestra en el Programa 6.5.

Programa 6.5. Reducción a la Forma de Hessenberg

```
def hess(A):
    H = np.copy(np.asarray(A, dtype=float))
    n = H.shape[0]
    for i in range(n-2):
        v = np.zeros(n)
        v[i+1:] = H[i+1:, i]
        v[i+1] = v[i+1] + signo(v[i+1])*np.linalg.norm(v)
        v /= np.linalg.norm(v)
        Q_aux = np.eye(n) - 2 * prod_vv(v)
        H = Q_aux.dot(H.dot(Q_aux.T))
    return H
```

En el siguiente ejemplo se aplica la reducción a la forma de Hessenberg a dos matrices, una no simétrica y otra simétrica.

Ejemplo 6.7. Consideremos las matrices

$$A = \begin{bmatrix} 3 & 5 & 1 & 3 & 1 \\ 4 & 5 & 4 & 8 & 2 \\ 6 & 3 & 3 & 1 & 3 \\ 9 & 2 & 5 & 0 & 4 \\ 8 & 1 & 0 & 6 & 5 \end{bmatrix} \quad \text{y} \quad B = \begin{bmatrix} 5 & -1 & 1 & 0 & 1 \\ -1 & 4 & 0 & -2 & 0 \\ 1 & 0 & 3 & 0 & -1 \\ 0 & -2 & 0 & 6 & 0 \\ 1 & 0 & -1 & 0 & 8 \end{bmatrix}.$$

Utilicemos la función `hess` del Programa 6.5 para aplicar la reducción a la forma de Hessenberg de cada una de estas matrices. Observe que la matriz A no es simétrica, en cambio, la matriz B si lo es. Primero apliquemos la reducción a la forma de Hessenberg a A.

```
>>> A = np.array([[3,5,1,3,1],[4,5,4,8,2],[6,3,3,1,3],[9,2,5,0,4],
...               [8,1,0,6,5]])
>>> H= cn.hess(A)
>>> print(H)      # Matriz de Hessenberg superior
[[  3.          -4.34607005   3.37599184   1.39936228  -1.93807622]
 [-14.03566885  11.77664975  -1.63046612   0.2045727   -1.1990255 ]
 [  0.          -6.59763555   0.27356309   3.04209207  -4.3045437 ]
 [  0.          -0.           2.78244959   2.50901041   3.40536333]
 [ -0.           0.          -0.          -1.31175497  -1.55922324]]
```

Note que la matriz `H` es efectivamente la matriz de Hessenberg H_0. Ahora apliquemos la reducción a la forma de Hessenberg a B.

```
>>> B = np.array([[5,-1,1,0,1],[-1,4,0,-2,0],[1,0,3,0,-1],
...               [0,-2,0,6,0],[1,0,-1,0,8]])
>>> H = cn.hess(B)
>>> print(H)      # Matriz tridiagonal
[[ 5.          1.73205081  0.          0.          0.        ]
 [ 1.73205081  4.33333333  2.3570226   0.         -0.        ]
 [ 0.          2.3570226   6.44666667  0.48124838 -0.        ]
 [ 0.          0.          0.48124838  7.41170984  0.36637657]
 [ 0.          0.          0.          0.36637657  2.80829016]]
```

Note que la matriz `H` es la matriz de Hessenberg H_0, que tiene una forma muy particular: tridiagonal.

La ventaja de usar esta versión del método *QR* en dos etapas es que la factorización *QR* de una matriz $n \times n$ con forma de Hessenberg requiere sólo $\mathbf{O}(n^2)$ operaciones punto-flotantes, reduciendo el costo por iteración. En el Algoritmo 6.6 se muestra el esquema general del método *QR* en dos etapas.

Algoritmo 6.6 (Método *QR* en dos etapas)**.**

Entrada: $A \in \mathbb{R}^{n\times n}$.

1: Construir $\widehat{Q}$ ortogonal tal que $A_0 = \widehat{Q}^T A \widehat{Q}$ es de Hessenberg. ▷ Etapa 1
2: **for** $k = 1, 2, \ldots$ **do** ▷ Etapa 2
3: $\quad Q_k R_k = A_{k-1}$ ▷ Factorización *QR* de A_{k-1}
4: $\quad A_k = R_k Q_k$ ▷ Recombinación de los factores en orden inverso
5: **end for**

Cuando $A \in \mathbb{R}^{n\times n}$ es una matriz simétrica, la matriz H_0 dada por (6.6) es simétrica y tridiagonal. En este caso, la utilización de la versión del método *QR* en dos etapas es muy ventajosa, ya que la factorización *QR* de una matriz $n \times n$ simétrica y tridiagonal requiere sólo $\mathbf{O}(n)$ operaciones punto-flotantes, reduciendo de forma significativa el costo por iteración. El método *QR* en dos etapas para el caso simétrico se muestra en el Algoritmo 6.7.

Algoritmo 6.7 (Método *QR* en dos etapas (caso simétrico))**.**

Entrada: $A \in \mathbb{R}^{n\times n}$ simétrica.

1: Construir $\widehat{Q}$ ortogonal tal que $A_0 = \widehat{Q}^T A \widehat{Q}$ es Tridiagonal. ▷ Etapa 1
2: **for** $k = 1, 2, \ldots$ **do** ▷ Etapa 2
3: $\quad Q_k R_k = A_{k-1}$ ▷ Factorización *QR* de A_{k-1}
4: $\quad A_k = R_k Q_k$ ▷ Recombinación de los factores en orden inverso
5: **end for**

El criterio de parada natural del método *QR* del Algoritmo6.6 es que la norma del triángulo inferior en bloques de la matriz A_{k+1}, descrito en el Teorema 6.10, sea menor que una cierta tolerancia dada. En la literatura existen otros criterios de parada más especializados. Para el caso simétrico, se puede utilizar el criterio de parada

$$\|diag(A_{k+1}) - diag(A_k)\|_\infty < \varepsilon,$$

donde $0 < \varepsilon < 1$ es una tolerancia dada y $diag(B) = (b_{11}, b_{22}, \ldots, b_{nn})$, es decir, $diag(B)$ es un vector cuyas componentes son los elementos de la diagonal principal de $B \in \mathbb{R}^{n\times n}$. Este criterio de parada se justifica por lo siguiente. Cuando A es simétrica, la sucesión de matrices A_k tiende a la matriz diagonal de la factorización Schur; por lo que se cumple

$$\lim_{k\to\infty} \|diag(A_{k+1}) - diag(A_k)\|_\infty = 0.$$

En otras palabras, la diagonal principal de la matrices A_k tiende a un vector cuyas componentes son los autovalores de A. En el Programa 6.6 se muestra la implementación del Algoritmo 6.7, empleando el criterio de parada dado arriba. Note que la función `metqr` halla solo los autovalores de A.

Se deja como ejercicio para el lector, modificar la función `metqr` para obtener los autovectores.

Programa 6.6. Método *QR* en dos etapas (caso simétrico)

```
def metqr(A, tol=1.0e-6, maxit=1000):
    # A es una matriz simétrica nxn
    H = hess(A)
    n = H.shape[0]
    it = 0
    while True:
        Q_aux, H_aux = qr_h(H)
        H_k = H_aux.dot(Q_aux)
        it += 1
        if np.linalg.norm(np.diag(H_k)-np.diag(H),np.inf) < tol:
            return np.diag(H_k)
        H = H_k
        if it > maxit:
            print('Alcanzó el máximo de iteraciones')
            return np.diag(H)
```

La convergencia del método *QR* en dos etapas puede ser lenta. Una manera de acelerar su convergencia es escoger apropiadamente desplazamientos. Sea $A \in \mathbb{R}^{n\times n}$ simétrica. Supongamos que ya hemos calculado la matriz $A_0 = \widehat{Q}^T A \widehat{Q}$ simétrica y tridiagonal. Las iteraciones del método *QR* son de la forma:

$$\text{Factorizar} \quad A_k - \lambda_k I = Q_k R_k,$$

$$\text{Asignar} \quad A_{k+1} = R_k Q_k + \lambda_k I,$$

donde el escalar λ_k es una aproximación de un autovalor de A_0. Observe que esta forma de iterar implica que $A_{k+1} = Q_k^T A_k Q_k$, por lo que con un proceso inductivo se tiene que la sucesión $\{A_k\}$ son matrices similares entre sí. Esta variante del método *QR* desplaza todos los autovalores de A_k por la misma cantidad λ_k y luego los restablece a su lugar original para construir A_{k+1}.

En el Algoritmo 6.8 se muestra el método *QR* en dos etapas con desplazamiento, cuando la matriz A es simétrica.

Algoritmo 6.8 (Método *QR* en dos etapas con desplazamiento)**.**

Entrada: $A \in \mathbb{R}^{n\times n}$ simétrica.

1: Construir $\widehat{Q}$ ortogonal tal que $A_0 = \widehat{Q}^T A \widehat{Q}$ es Tridiagonal. ▷ Etapa 1
2: Escoger $\lambda_0 \in \mathbb{R}$ ▷ Desplazamiento inicial
3: **for** $k = 1, 2, \ldots$ **do** ▷ Etapa 2
4: $\quad A_k - \lambda_k I = Q_k R_k$ ▷ Factorización *QR* de $A_k - \lambda_k I$
5: $\quad A_{k+1} = R_k Q_k + \lambda_k I$ ▷ Recombinación de los factores en orden inverso
6: $\quad$ Escoger $\lambda_{k+1} \in \mathbb{R}$ ▷ Actualización del desplazamiento
7: **end for**

En general, una buena escogencia de $\lambda_k \in \mathbb{R}$, para cada k, es el elemento de la esquina inferior derecha de la matriz A_k. Salvo en algunos casos esta escogencia no funciona; por ejemplo, si la matriz A_0 es

$$A_0 = \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix}$$

y se toma $\lambda_0 = 3$, se tiene

$$A_0 - 3I = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix},$$

que es una matriz ortogonal; por lo tanto, la factorización QR de $A_0 - 3I$ está dada por $Q_0 = A_0 - 3I$ y $R_0 = I$. Luego, $A_1 = R_0 Q_0 + 3I = A_0$; en consecuencia, el Algoritmo 6.8 deja fija la matriz A_0 en todas las iteraciones.

Dado que la escogencia del elemento de la esquina inferior derecha de la matriz A_k como desplazamiento puede fallar en algunas ocasiones, una escogencia diferente, conocida como el desplazamiento de Wilkinson, se usa con frecuencia en la práctica. El desplazamiento de Wilkinson se define como el autovalor de la submatriz 2×2 de la esquina inferior derecha de A_k más cercano al elemento de la esquina inferior derecha de la misma matriz; y en caso de empate se escoge cualquiera de los dos. El cálculo de este desplazamiento es inmediato, ya que los autovalores de una matriz 2×2 se obtienen con facilidad como las raíces del polinomio característico, que en este caso es un polinomio de grado 2. Denotemos con B la submatriz 2×2 de la esquina inferior derecha de A_k:

$$B = \begin{bmatrix} a_1 & b_1 \\ b_2 & a_2 \end{bmatrix}.$$

Una fórmula numéricamente estable para el desplazamiento de Wilkinson es:

$$\lambda = a_2 - \frac{\text{signo}(\delta) b_1^2}{|\delta| + \sqrt{\delta^2 + b_1^2}},$$

donde $\delta = (a_1 - a_2)/2$ y $\text{signo}(t) = 1$ para $t \geq 0$, y $\text{signo}(t) = 0$ para $t < 0$.

Notemos algo importante en las matrices A_k generadas por el método QR, cuando A es simétrica. La matriz $A_k = (a_{ij}^k)$ puede ser representada en bloques:

$$A_k = \begin{bmatrix} M_k & \mathbf{v} \\ \mathbf{v}^T & a_{nn}^k \end{bmatrix},$$

donde M_k es una matriz simétrica de orden $(n-1) \times (n-1)$ y $\mathbf{v} = (0, \ldots, 0, a_{n-1,n}^k) \in \mathbb{R}^{n-1}$. Además, para k suficientemente grande, se tiene $|a_{n-1,n}^k| < \varepsilon$, con $\varepsilon \in (0,1)$. Es claro ver que el elemento a_{nn}^k es una aproximación de un autovalor de A y los restantes $n-1$ autovalores de A son los mismos autovalores de M_k. De esta forma, cuando $|a_{n-1,n}^k| < \varepsilon$, se puede buscar el siguiente autovalor de A en la matriz M_k, es decir, ahora se puede aplicar el método QR a la matriz M_k hasta encontrar un nuevo autovalor. Este proceso de deflación se repite sucesivamente hasta obtener todos los autovalores de A. En el Programa 6.7 se muestra la implementación de este procedimiento.

Programa 6.7. Método *QR* en dos etapas con desplazamiento y deflación

```
def metqr_des(A, tol=1.0e-10, maxit=1000):
    # A es una matriz simétrica nxn
    H = np.copy(np.asarray(A, dtype=float))
    n = H.shape[0]
    egs = []
    k = 0
    while k < n:
        m = H.shape[0]
        if m==1:
            egs.append(H[0,0])
```

```
            k += 1
        else:
            I = np.eye(m)
            H = hess(H)
            it = 0
            d = np.abs(H[m-2,m-2]) + np.abs(H[m-1,m-1])
            while np.abs(H[m-2,m-1]) > tol*d and k <= maxit:
                lam = WilkinsonShift(H[m-2:,m-2:])
                Q_aux, H_aux = qr_h(H - lam*I)
                H = H_aux.dot(Q_aux) + lam*I
                it += 1
            egs.append(H[m-1,m-1])
            H = H[:m-1,:m-1]
            k += 1
    return np.array(egs)
```

El Programa 6.7 utiliza el desplazamiento de Wilkinson implementado en la siguiente función.

```
def WilkinsonShift(B):
    a1, a2, b1, b2 = B[0,0], B[1,1], B[0,1], B[1,0]
    d = (a1-a2)*.5
    lam = a2 - signo(d)*b1**2/(np.abs(d) + np.sqrt(d**2 + b1**2))
    return lam
```

Este programa también emplea el siguiente criterio de parada (conocido como criterio de Wilkinson) en cada paso de deflación:

$$|h_{m-1,m}| < \text{tol}\,(|h_{m-1,m-1}| + |h_{m,m}|)\,,$$

donde $0 < \text{tol} < 1$ es una tolerancia prefijada y $H = (h_{ij}) \in \mathbb{R}^{m\times m}$ es la submatriz de cada paso de deflación.

Ejemplo 6.8. Hallemos los autovalores de la matriz

$$A = \begin{bmatrix} 5 & -1 & 1 & 0 & 1 \\ -1 & 4 & 0 & -2 & 0 \\ 1 & 0 & 3 & 0 & -1 \\ 0 & -2 & 0 & 6 & 0 \\ 1 & 0 & -1 & 0 & 8 \end{bmatrix}$$

usando el método *QR* con y sin desplazamiento. Mediremos el tiempo de CPU utilizado por cada algoritmo; para ello, empleamos la librería `time`.

```
>>> import time as ti
```

Primero utilicemos *QR* sin desplazamiento.

```
>>> A = np.array([[5,-1,1,0,1],[-1,4,0,-2,0],[1,0,3,0,-1],
...               [0,-2,0,6,0],[1,0,-1,0,8]])
>>> t = ti.clock(); eigs = metqr(A); t = ti.clock()-t
>>> print(eigs)
[8.41631492 7.34157598 5.41893569 2.77937402 2.04379939]
>>> print(t)
0.029466325360303358
```

El algoritmo *QR* sin desplazamiento necesitó de aproximadamente 0.03 segundos para hallar los autovalores de *A*. Ahora utilicemos *QR* con desplazamiento.

```
>>> t = ti.clock(); eigs = metqr_des(A); t = ti.clock()-t
>>> print(eigs)
[2.77937402 2.04379939 7.34157344 5.41893569 8.41631745]
>>> print(t)
0.014558261698027763
```

El algoritmo *QR* con desplazamiento necesitó de aproximadamente 0.015 segundos para hallar los autovalores de *A*.

En el ejemplo anterior observamos que el algoritmo *QR* con desplazamiento posee mejor comportamiento computacional que la versión sin desplazamiento. Veamos como se comporta para una matriz simétrica de mayor tamaño.

Ejemplo 6.9. Sea $A \in \mathbb{R}^{n\times n}$ la matriz de poisson dada en el Ejemplo 2.29. Veamos el comportamiento computacional del algoritmo *QR* con y sin desplazamiento, cuando es aplicado a la matriz de poisson. En la Tabla 6.1 se muestra el tiempo de CPU del algoritmo *QR* sin y con desplazamiento, para matrices de poisson de distintas dimensiones. Esta tabla se generó utilizando el siguiente Script.

```
import time as ti
txt_s = 'QR sin desplazamiento'
txt_c = 'QR con desplazamiento'
for N in [3,5,8,10]:
    A = matrix_poisson(N).toarray()
    t = ti.clock()
    eigs = metqr(A)
    t = ti.clock()-t
    txt = '\t{0:1.4f}'.format(t)
    txt_s = txt_s + txt
    t = ti.clock()
    eigs = metqr_des(A)
    t = ti.clock() - t
    txt = '\t{0:1.4f}'.format(t)
    txt_c = txt_c + txt
print(txt_s)
print(txt_c)
```

Tabla 6.1. Tiempo de CPU del algoritmo *QR* sin y con desplazamiento, para matrices de poisson de diferentes dimensiones.

	Dimensión (n)			
	9	25	64	100
QR sin desplazamiento	0.1202	1.5151	48.5250	410.9947
QR con desplazamiento	0.0081	0.0930	1.6022	6.3382

Note que el tiempo de CPU del algoritmo *QR* con desplazamiento, es significativamente menor que la versión sin desplazamiento.

Existen muchas implementaciones del método *QR* y de sus variantes, para el caso no simétrico. En general, las implementaciones en software comerciales o libres utilizan combinaciones de varios algoritmos para calcular los autovalores y autovectores de una matriz A cualquiera. La librería `Numpy` cuenta con la función `eig` (en el módulo `linalg`), que permite calcular los autovalores y autovectores de una matriz cualquiera. Esta función utiliza rutinas de LAPACK (Linear Algebra Package), que es una librería escrita en Fortran 90 y que proporciona rutinas para resolver sistemas de ecuaciones lineales, problemas de cuadrados mínimos, problemas de valores propios y problemas de valores singulares.

Los parámetros y la salida de la función

```
d,V = eig(A)
```

están dados por:

Parámetros `A`: arreglo(n,n)
: Matriz a la que se le calculará los autovalores y autovectores.

Salida `d`: n-upla
: Los autovalores, cada uno repetido de acuerdo a su multiplicidad. Los autovalores no están necesariamente ordenados. La matriz resultante será de tipo complejo, a menos que la parte imaginaria sea cero, en cuyo caso se convertirá en un tipo real. Cuando `A` es real, los valores propios resultantes serán reales (parte imaginaria 0) u ocurrirán en pares conjugados.

`V`: arreglo(n,n)
: Los autovectores normalizados, de modo que la columna `V[:,i]` es el autovector correspondiente al autovalor `d[i]`.

Mostremos con algunos ejemplos el uso de la función `eig`.

Ejemplo 6.10. Consideremos la matriz

$$A = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{5} & \frac{1}{4} & \frac{1}{3} & \frac{1}{2} & 1 \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{6} & \frac{1}{5} & \frac{1}{4} & \frac{1}{3} & \frac{1}{2} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \end{bmatrix}.$$

Hallemos los autovalores y los autovectores de A usando la función `eig`.

```
>>> A = np.array([[1,1/2,1/3,1/4,1/5],[1/5,1/4,1/3,1/2,1],
... [1/2,1/3,1/4,1/5,1/6],[1/6,1/5,1/4,1/3,1/2],[1/3,1/4,1/5,1/6,1/7]])
>>> d,V = np.linalg.eig(A)
>>> print(d)
[ 1.76171257  0.35985837 -0.16829536 -0.00195419  0.02486908]
>>> print(V)
[[-0.66191798 -0.62745043 -0.25256252 -0.06698828 -0.12225426]
 [-0.47093352  0.64929818  0.83001194  0.47020076  0.32695022]
 [-0.39634514 -0.0944432  -0.34640285 -0.77938287  0.34057692]
 [-0.31417297  0.41885813  0.21180847  0.4057901  -0.83500378]
 [-0.29033099  0.01887293 -0.28711426 -0.04817215  0.2548205 ]]
```

La función `eig` halló efectivamente todos los autovalores y autovectores de A.

Hallemos los autovalores de una matriz no simétrica con autovalores complejos.

Ejemplo 6.11. Consideremos la matriz

$$A = \begin{bmatrix} 2 & 2 & 1 & 1 \\ -3 & -1 & -1 & 0 \\ 0 & 2 & 1 & 3 \\ 1 & -1 & -1 & -2 \end{bmatrix}.$$

Hallemos los autovalores y los autovectores de A usando la función `eig`.

```
>>> A = np.array([[2,2,1,1],[-3,-1,-1,0],[0,2,1,3],[1,-1,-1,2]])
>>> d,V = np.linalg.eig(A)
>>> print(d)
[0.47244117+2.9184116j  0.47244117-2.9184116j  2.50760457+0.j
 0.5475131 +0.j        ]
>>> print(V)
[[ 0.26704354-0.36391971j  0.26704354+0.36391971j -0.37018169+0.j
  -0.00907849+0.j        ]
 [ 0.10091208+0.54145196j  0.10091208-0.54145196j  0.47156597+0.j
  -0.5169614 +0.j        ]
 [ 0.63046193+0.j          0.63046193-0.j         -0.54352186+0.j
   0.82724001+0.j        ]
 [-0.17814331+0.25234784j -0.17814331-0.25234784j -0.58751599+0.j
   0.21986918+0.j        ]]
```

La función `eig` halló efectivamente todos los autovalores y autovectores de A.

6.4 Ejercicios

1. a) Pruebe que 5 es un autovalor de la matriz

$$A = \begin{bmatrix} 6 & 3 & 3 & 1 \\ 0 & 7 & 4 & 5 \\ 0 & 0 & 5 & 4 \\ 0 & 0 & 0 & 8 \end{bmatrix}$$

b) Determine un autovector de A asociado al autovalor 5.

2. ¿Cuáles son los autovalores y sus correspondientes autovectores de la siguiente matriz?

$$A = \begin{bmatrix} 1 & 2 & -4 \\ 0 & 2 & 1 \\ 0 & 0 & 3 \end{bmatrix}$$

3. Suponga que la suma por filas de una matriz A tiene el mismo valor, digamos α. Demuestre que α es un autovalor de A, y halle su autovector asociado.

4. Establezca que si A o B es no singular, entonces $I - AB$ tiene los mismos autovalores que $I - BA$.

5. Sea A una matriz cuadrada $n \times n$. Establezca que si λ es un autovalor de A y P es un polinomio, entonces $P(\lambda)$ es autovalor de $P(A)$.

6. Demuestre que una matriz A $n \times n$ es singular si, y solo si, el cero es uno de sus autovalores.

7. Demuestre que los autovalores de una matriz A simétrica definida positiva son todos positivos.

8. Si λ es un autovalor de una matriz ortogonal Q, demuestre que $|\lambda| = 1$.

9. Caracterice los autovalores de una transformación de Householder,

$$H = I - 2\frac{\mathbf{v}\mathbf{v}^T}{\mathbf{v}^T\mathbf{v}},$$

donde $\mathbf{v}$ es un vector no nulo.

10. Dé un ejemplo de una matriz A 2×2 y un vector inicial $\mathbf{x}_0$ distinto de cero, de modo que el método de las potencias no converja al autovector correspondiente al autovalor dominante de A.

11. Explique la velocidad lenta de convergencia del método de las potencias con las siguientes matrices:

a) $A = \begin{bmatrix} 3 & 2 & 3 \\ 0 & 2.9 & 1 \\ 0 & 0 & 1 \end{bmatrix}$

b) $A = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 10 & 0 \\ 1 & 1 & 9.8 \end{bmatrix}$

Seleccione un desplazamiento apropiado σ, y aplique el método de las potencias inverso con desplazamiento para observar mejoría en la velocidad de convergencia.

12. Aplique 3 iteraciones del método QR con desplazamiento simple (esquina inferior derecha) a cada una de las siguientes matrices y observe la forma de converger, o de no converger, de la subdiagonal principal.

a) $A = \begin{bmatrix} 1 & 2 & 0 \\ 2 & 3 & 4 \\ 0 & 4 & 1 \end{bmatrix}$

b) $A = \begin{bmatrix} 4 & 5 & 6 \\ 1 & 0 & 1 \\ 0 & -2 & 2 \end{bmatrix}$

13. Sea $A \in \mathbb{R}^{3\times 3}$ una matriz simétrica con autovalores $\lambda_1 > \lambda_2 > \lambda_3 > 0$, y sean $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$ los correspondientes autovectores ortonormales de A. Defina el vector $\mathbf{z}_0 = (\mathbf{v}_2 + \mathbf{v}_3)/\|\mathbf{v}_2 + \mathbf{v}_3\|_3$, y considere la sucesión $\{\mathbf{z}_k\}$ generada por:

$$\mathbf{z}_k = \frac{(A - \sigma I)\mathbf{z}_k}{\|(A - \sigma I)\mathbf{z}_k\|_2}.$$

a) ¿A qué vector en $\mathbb{R}^3$ converge la sucesión $\{\mathbf{z}_k\}$ si $\sigma = 0$?

b) ¿A qué vector converge la sucesión $\{\mathbf{z}_k\}$ si $\sigma = 0.2\lambda_3 + 0.8\lambda_2$?

Justifique su respuesta.

14. Usando el método de las potencias con desplazamiento, calcule el autovalor positivo más grande y el autovalor negativo de módulo más grande de

$$A = \begin{bmatrix} 3 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 3 \end{bmatrix}.$$

A es llamada matriz de Wilkinson.

15. Utilice convenientemente el método QR o alguna de sus variantes para hallar los autovalores de la matriz 50×50

$$A = \begin{bmatrix} 2 & 1 & 0 & 0 & \cdots & 0 & 1 \\ 1 & 2 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 2 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 2 & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1 & 2 & 1 \\ 1 & 0 & \cdots & 0 & 0 & 1 & 2 \end{bmatrix}.$$

Nota: Esta es una matriz difícil que tiene muchos pares de autovalores dobles.

16. Escriba una función en Python de nombre `deflacion_metpot`:

```
eigs,V = deflacion_metpot(A, x0, tol=1.0e-6, maxit=1000)
```

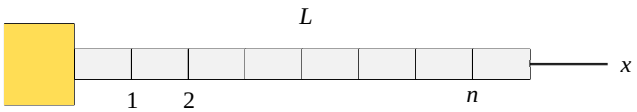
con salida un arreglo `eigs` con todos los autovalores de una matriz simétrica A $n \times n$ y una matriz $V \in \mathbb{R}^{n \times n}$ cuyas columnas sean los autovectores de A. La función `deflacion_metpot` debe combinar la técnica de deflación y el método de las potencias. Los argumentos de entrada tienen los mismos valores por defecto que la función `metpot` del Programa 6.1. Encuentre todos los autovalores y los autovectores de las siguientes matrices A, ejecutando la función `deflacion_metpot`.

a) $A = \begin{bmatrix} 5 & 3 & 0 & 3 \\ 3 & 7 & 8 & 3 \\ 3 & 3 & 4 & 3 \\ 9 & 11 & 8 & 7 \end{bmatrix}$

b) $A = \begin{bmatrix} 2 & 1 & 2 & 0 & 5 \\ 1 & 4 & 0 & 0 & 3 \\ 2 & 0 & 5 & 1 & 0 \\ 0 & 0 & 1 & 1 & 2 \\ 5 & 3 & 0 & 2 & 2 \end{bmatrix}$

Compare sus resultados con los resultados de la función `eig` de la librería `Numpy`.

17. La figura muestra una viga en voladizo con una malla de diferencia finita superpuesta.



Si $u(x,t)$ es el desplazamiento lateral de la viga, la ecuación diferencial del movimiento que gobierna las vibraciones de flexión es

$$u^{(4)} = -\frac{\gamma}{EI}\ddot{u},$$

donde γ es la masa por unidad de longitud y EI es la rigidez de flexión. Las condiciones de borde son $u(0,t) = u'(0,t) = u''(L,t) = u'''(L,t) = 0$. Tomando $u(x,t) = y(x)\,\text{sen}\,\omega t$ el problema se convierte en:

$$y^{(4)} = \frac{\omega^2\gamma}{EI}y, \qquad y(0) = y'(0) = y''(L) = y'''(L) = 0.$$

Las ecuaciones en diferencias finitas correspondientes son:

$$\begin{bmatrix} 7 & -4 & 1 & 0 & 0 & \cdots & 0 \\ -4 & 6 & -4 & 1 & 0 & \cdots & 0 \\ 1 & -4 & 6 & -4 & 1 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 1 & -4 & 6 & -4 & 1 \\ 0 & \cdots & 0 & 1 & -4 & 5 & -2 \\ 0 & \cdots & 0 & 0 & 1 & -2 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-2} \\ y_{n-1} \\ y_n \end{bmatrix} = \lambda \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 0 & 1 & 0 & 0 \\ 0 & \cdots & 0 & 0 & 0 & 1 & 0 \\ 0 & \cdots & 0 & 0 & 0 & 0 & 1/2 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-2} \\ y_{n-1} \\ y_n \end{bmatrix},$$

donde

$$\lambda = \frac{\omega^2\gamma}{EI}\left(\frac{L}{n}\right)^4.$$

Escriba un programa que calcule las dos frecuencias circulares más bajas de la viga y las formas de modo correspondientes (autovectores) utilizando el método de Rayleigh. Ejecutar el programa con $n = 10$. *Nota*: La solución analítica para la frecuencia circular más baja es $\omega_1 = (3.515/L^2)\sqrt{EI/\gamma}$.

7 Solución Numérica de Ecuaciones Diferenciales Ordinarias

La mayoría de los sistemas físicos cambian con el tiempo. Desde un satélite en órbita al enfriamiento de una taza de café, desde un péndulo oscilante a un radio-isótopo en descomposición, desde la reacción de sustancias químicas a la competencia de especies biológicas, un estado de flujo es la norma. Uno de los problemas que motivaron la inven-ción del cálculo diferencial fue caracterizar el movimiento de los cuerpos celestes y los proyectiles terrestres para poder predecir su ubicación futura. Las ecuaciones diferenciales proporcionan un lenguaje matemático para describir el cambio continuo. Así como las leyes de movimiento de Newton, la mayoría de las leyes fundamentales de la ciencia se expresan como ecuaciones diferenciales. Incluso un sistema que no está cambiando a menudo se entiende mejor a describirlo como un estado de equilibrio de una ecuación diferencial relevante. Las ecuaciones diferenciales son modelos matemáticos que describen cómo las variables y sus derivadas (tasas de cambio) con respecto a una o más variables independientes, se afectan mutua y dinámicamente. Sus soluciones nos muestran cómo la(s) variable(s) dependiente(s) cambian por efecto de la(s) variable(s) independiente(s).

Veremos que existen teoremas generales para probar la existencia y unicidad de la solución de una ecuación diferencial. Sin embargo, en general es imposible encontrar una expresión analítica para la solución. En estos casos, los métodos numéricos pueden proporcionar una respuesta, aunque sea limitada. En este capítulo estudiaremos los métodos numéricos más usados en la resolución de ecuaciones diferenciales. Es conocido que la solución de una ecuación diferencial es una función defina en cierto espacio. Por ejemplo, si estamos considerando ecuaciones diferenciales que envuelven funciones reales, entonces la solución de esta ecuación diferencial será, por supuesto, una función real. Los métodos numéricos que estudiaremos no hallan en sí la función, sino que éstos generan iterativamente un conjunto de números o vectores, que aproximan los valores de la función en un conjunto finito de puntos, los cuales representan una discretización del espacio de trabajo.

7.1 Problema de Valor Inicial

La forma general de las *ecuaciones diferenciales de primer orden* es

$$y'(t) = f(t, y(t)), \tag{7.1}$$

donde $y'(t) = dy/dt$ y $f(t,y)$ es una función definida de $\mathbb{R}^2$ en $\mathbb{R}$. Es de nuestro interés encontrar una solución de (7.1) en un intervalo $[a,b]$, dada una función $f(t,y)$. Si la función $f(t,y)$ es lineal en y se dice que la ecuación diferencial ordinaria es lineal, y se dice que es no lineal cuando $f(t,y)$ es no lineal en y. Por ejemplo, $y'(t) = t^3 y(t) + 1$ es lineal, en cambio $y'(t) = te^{y(t)}$ es no lineal.

Bajo ciertas condiciones sobre la función $f(t,y)$, la ecuación (7.1) posee infinitas soluciones, pero sólo existe una solución de (7.1) que pasa por un punto $(t_0, y(t_0))$ en el cual se conoce el valor de la solución. Por ejemplo, la ecuación diferencial $y'(t) = t$ tiene infinitas soluciones de la forma $y(t) = \frac{1}{2}t^2 + c$, pero sólo una de estas soluciones satisface que $y(1) = 1$, a saber: $y(t) = \frac{1}{2}t^2 + \frac{1}{2}$.

Un *problema de valor inicial* (también conocido como problema de Cauchy) consiste en encontrar la solución de (7.1), dadas las condiciones iniciales adecuadas. Particularmente, el problema de valor inicial asociado a la ecuación (7.1) consiste en encontrar una función $y : [a,b] \to \mathbb{R}$ tal que

$$\begin{cases} y'(t) = f(t,y(t)), & t \in [a,b], \\ y(t_0) = y_0, \end{cases} \tag{7.2}$$

donde $t_0 = a$, $y(t_0) = y_0$ es la condición inicial y $f(t,y)$ es una función dada, que es continua con respecto a las dos variables. Si f depende de t solo a través de y, la ecuación diferencial se llama *autónoma*.

Si $f(t,y)$ es continua con respecto a t, entonces la solución de (7.2) satisface

$$y(t) - y_0 = \int_{t_0}^{t} f(\tau, y(\tau))\, d\tau. \tag{7.3}$$

Por lo tanto, si $f(t,y)$ es continua, el problema (7.2) es equivalente a la ecuación integral (7.3). Veremos más adelante cómo aprovechar esta equivalencia en los métodos numéricos.

Existen problemas de valor inicial que no poseen solución única. Para garantizar la existencia y la unicidad de la solución (7.2) es necesario que $f(t,y)$ satisfaga ciertas condiciones, las cuales se establecen en los siguientes tres teoremas. Las demostración de estos teoremas se pueden encontrar en libros especializados de Análisis Numérico (ver, por ejemplo, [30], [14]).

Teorema 7.1. *Si $f(t,y)$ es continua en un rectángulo R centrado en (t_0,y_0),*

$$R = \{(t,y) \in \mathbb{R}^2 : |t - t_0| \leq \alpha, \quad |y - y_0| \leq \beta\}, \tag{7.4}$$

entonces el problema de valor inicial (7.2) *tiene una solución $y(t)$, para todo t tal que*

$$|t - t_0| \leq \min(\alpha, \beta/M),$$

donde $M > 0$ es tal que $|f(t,y)| \leq M$, para todo $(t,y) \in R$.

Teorema 7.2. *Si $f(t,y)$ y $\partial f(t,y)/\partial y$ son continuas en el rectángulo R definido por* (7.4)*, entonces el problema de valor inicial* (7.2) *tiene una solución única en el intervalo*

$$(t_0 - \min(\alpha, \beta/M)\,,\, t_0 + \min(\alpha, \beta/M))\,,$$

donde $M > 0$ es tal que $|f(t,y)| \leq M$, para todo $(t,y) \in R$.

Teorema 7.3. *Supongamos que $f(t,y)$ es continua en la franja $a \leq t \leq b$, $-\infty < y < \infty$. Supongamos también que $f(t,y)$ satisface la condición de Lipschitz en la segunda variable, esto es, existe una constante $\ell > 0$ tal que $|f(t,y_1) - f(t,y_2)| \leq \ell |y_1 - y_2|$, para todo $y_1, y_2 \in R$, donde el rectángulo R está definido por* (7.4)*. Entonces, el problema de valor inicial* (7.2) *tiene una solución única en el intervalo $[a,b]$.*

En el siguiente ejemplo empleamos los teoremas anteriores para demostrar la existencia y unicidad de la solución de un problema de valor inicial.

Ejemplo 7.1. Demostremos que el problema de valor inicial

$$\begin{cases} y'(t) = 2t\,e^{-y}, & t \in [0,b], \\ y(0) = 0, \end{cases} \tag{7.5}$$

tiene una solución única para todo $t \in [0,b]$ y cualquier $b > 0$.

Demostración. Se tiene que $f(t,y) = 2t\,e^{-y}$. Es claro que f es continua en todo $(t,y) \in \mathbb{R}^2$; así por el Teorema 7.1, el problema de valor inicial (7.5) tiene una solución. Ahora, como

$$\frac{\partial f}{\partial y}(t,y) = -2t\,e^{-y},$$

entonces $\partial f/\partial y$ es continua en todo $(t,x) \in \mathbb{R}^2$. Luego, por el Teorema 7.2, podemos asegurar que el problema de valor inicial (7.5) tiene solución única, para todo $t \in [0,b]$. Demostremos este mismo hecho, pero ahora utilizando el Teorema 7.2. Probemos que $f(t,x)$ satisface la condición de Lipschitz en la segunda variable. Sean $0 < a < b$, $\beta > 0$ e $y_1, y_2 \in (-\beta,\beta)$. Sin pérdida de generalidad podemos suponer que $y_1 \leq y_2$. Así,

$$|f(t,y_1) - f(t,y_2)| = 2t\left|e^{-y_1} - e^{-y_2}\right|.$$

Por el Teorema del Valor Medio existe $c \in (x_1,x_2)$ tal que

$$e^{-y_1} - e^{-y_2} = -e^{-c}(y_1 - y_2) \leq e^{-\beta}(y_2 - y_1).$$

Luego,

$$|f(t,y_1) - f(t,y_2)| \leq 2te^{-\beta}|y_1 - y_2| \leq \ell\,|y_1 - y_2|,$$

para todo $a \leq t \leq b$, y $y_1, y_2 \in (-\beta,\beta)$, donde $\ell = 2be^{-\beta} > 0$. En otras palabras, $f(t,y)$ satisface la condición de Lipschitz en la segunda variable, para todo $y \in (-\beta,\beta)$. Como a, b y β se escogieron en forma arbitraria, entonces por el Teorema 7.3, el problema de valor inicial (7.5) tiene solución única, para todo $t \in [0,b]$.

El procedimiento descrito en el ejemplo anterior parece complicado y tedioso, pero es necesario realizarlo para garantizar la existencia y unicidad de la solución. A la mayoría de los problemas de valor inicial no se les puede obtener la solución exacta, salvo contadas excepciones. Por ejemplo, el problema de valor inicial

$$\begin{cases} y'(t) = e^{-y^2}, & t \in [0,1], \\ y(0) = 1, \end{cases}$$

tiene solución única, pero es imposible hallar explícitamente a $y(t)$. Garantizar que existe al menos una solución del problema (7.2), es una condición fundamental para que los métodos numéricos que estudiaremos converjan. Sino se tiene conocimiento de por lo menos la existencia de la solución, no existen garantías teóricas que la aproximación obtenida por los métodos numéricos sea una solución.

7.2 Métodos de un Paso

Vamos a abordar la aproximación numérica del problema (7.2). Fijemos $[a,b]$ como el intervalo de integración y, para un entero $N>0$, los nodos de integración $t_n = a+nh$, con $n = 0,1,2,\ldots,N$ y $h = (b-a)/N$. La cantidad h se llama *longitud de paso* de la discretización. Sea y_n la aproximación en el nodo t_n de la solución exacta $y(t_n)$, esto es, $y(t_n) \approx y_n$. La estrategia de los métodos numéricos que estudiaremos es la siguiente:

- Dada la condición inicial $y_0 = y(t_0)$, calcular con alguna fórmula recursiva las aproximaciones $y_1, y_2, \ldots, y_N$.

Definición 7.1. Un método numérico para la aproximación del problema (7.2) se denomina *método de un paso*, si la fórmula recursiva para hallar y_{n+1} depende solamente de y_n, para todo $n = 0,1,\ldots,N-1$. De lo contrario, el esquema se llama *método multipaso*.

Por ahora, enfocamos nuestra atención en los métodos de un paso. Comencemos con los *métodos basados en expansiones de Taylor* o, simplemente, *métodos de Taylor*.

7.2.1 Métodos de Taylor

Supongamos que $y(t)$ es $(m+1)$ veces diferenciable. Los métodos numéricos basados en expansiones de Taylor, se derivan del desarrollo de Taylor de $y(t+h)$ alrededor de $t-h$, para $h>0$,

$$y(t+h) = y(t) + hy'(t) + \frac{y^2}{2!}y''(t) + \cdots + \frac{h^m}{m!}y^{(m)}(t) + E_m, \tag{7.6}$$

donde

$$E_m = \frac{h^{m+1}}{(n+1)!}\, y^{(m+1)}(t+\delta h), \quad 0<\delta<1.$$

Así, el valor de y en $t+h$ se proxima por

$$y(t+h) \approx y(t) + \sum_{j=1}^{m} \frac{h^j}{j!} y^{(j)}(t), \tag{7.7}$$

donde

$$\begin{aligned}
y'(t) &= f,\\
y''(t) &= f_t + f_y y' = f_t + f_y f,\\
y'''(t) &= f_{tt} + 2f_{ty}y' + f_y y'' + f_{yy}(y')^2\\
&= f_{tt} + 2f_{ty}y' + f_{yy}(f)^2 + f_y(f_t + f_y f),\\
y^{(4)}(t) &= f_{ttt} + 3f_{tty}y' + 3f_{tyy}(y')^2 + 3f_{ty}y'' + f_y y''' + 3f_{yy}y'y'' + f_{yyy}(y')^2\\
&= (f_{ttt} + 3f_{tty}f + 3f_{tyy}f^2 + f_{yyy}f^3) + f_y(f_{tt} + 2f_{ty}f + f_{yy}f^2)\\
&\quad + 3(f_t + f_y f)(f_{ty} + f_{yy}f) + f_y^2(f_t + f_y f),\\
&\ \ \vdots
\end{aligned}$$

y, en general,

$$y^{(j)}(t) = D^{(j-1)} f(t, y(t)), \tag{7.8}$$

donde D es el operador de derivación

$$D = \left(\frac{\partial}{\partial t} + f\frac{\partial}{\partial y}\right).$$

De esta forma, utilizando la ecuación (7.7), un método de Taylor se define en forma recursiva como:

$$y_{n+1} = y_n + \sum_{j=1}^{m} \frac{h^j}{j!} y^{(j)}(t_n), \qquad n = 0,1,\ldots,N-1, \tag{7.9}$$

donde $y^{(j)}(t)$ está dada por (7.8). La ecuación (7.9) se denomina *ecuación en diferencias*.

Al resolver numéricamente un problema de valor inicial aparecen varios errores: *error de truncamiento local* y *error de truncamiento global*.

Definición 7.2. El error de truncamiento local es el error que se comete al pasar de y_n a y_{n+1} debido al truncamiento de la serie que se usa para tal fin.

El método de Taylor (7.9) se obtiene truncando el desarrollo de Taylor luego de m términos, así su error de truncamiento local está dado por

$$E_m = \frac{h^{m+1}}{(n+1)!} y^{(m+1)}(\xi_n), \quad \xi_n \in (t_n, t_{n+1}). \tag{7.10}$$

En este caso se dice que el error de truncamiento local del método de Taylor es de orden h^{m+1} o, equivalentemente, $\mathbf{O}(h^{m+1})$.

Definición 7.3. El error de truncamiento global es la diferencia entre la solución exacta $y(t_n)$ y la aproximación y_n en $t = t_n$, es decir, es igual a $y(t_n) - y_n$.

Considerando las hipótesis del Teorema 7.2 o del Teorema 7.3, se puede establecer que el error de truncamiento global del método de Taylor (7.9) es de orden $\mathbf{O}(h^m)$. Se dice entonces que un método de Taylor es de *orden m*, si su error de truncamiento global es de orden $\mathbf{O}(h^m)$.

Ejemplo 7.2. Consideremos el problema de valor inicial

$$\begin{cases} y'(t) = y - t^2 + 1, & t \in [0,2], \\ y(0) = 0.5, \end{cases}$$

cuya solución exacta es $y(t) = t^2 + 2t + 1 - \frac{1}{2}e^t$. Aproximemos el valor de $y(0.4)$ usando el método de Taylor de orden 4, con $h = 0.2$.
Se tiene que:

$$f(t,y) = y - t^2 + 1, \qquad t_0 = 0, \qquad y_0 = 0.5.$$

La ecuación en diferencias del método de Taylor de orden 4 tiene la forma

$$y_{n+1} = y_n + h f(t_n, y_n) + \frac{h^2}{2} y''(t_n) + \frac{h^3}{3!} y'''(t_n) + \frac{h^4}{4!} y^{(4)}(t_n), \quad n = 0,1,\ldots$$

Ahora, considerando $f(t,y)$ dada, las derivadas $y^{(j)}(t_n)$, para $j = 1,2,3,4$, son:

$$\begin{aligned}
y'(t_n) &= f(t_n,y_n) = y_n - t_n^2 + 1,\\
y''(t_n) &= f_t(t_n,y_n) + f_y(t_n,y_n)y'(t_n) = y_n - t_n^2 - 2t_n + 1,\\
y'''(t_n) &= f_{tt}(t_n,y_n) + f_y(t_n,y_n)y''(t_n) = y_n - t_n^2 - 2t_n - 1,\\
y^{(4)}(t_n) &= f_y(t_n,y_n)y'''(t_n) = y_n - t_n^2 - 2t_n - 1.
\end{aligned}$$

De esta forma, la ecuación en diferencias del método de Taylor de orden 4 para el problema de valor inicial dado, es la siguiente:

$$\begin{aligned}
y_{n+1} =& y_n + h(y_n - t_n^2 + 1) + \frac{h^2}{2}(y_n - t_n^2 - 2t_n + 1)\\
&+ \left(\frac{h^3}{6} + \frac{h^4}{24}\right)(y_n - t_n^2 - 2t_n - 1), \qquad n = 0,1,\ldots
\end{aligned}$$

Se tiene que $t_0 = 0$, $t_1 = 0.2$ y $t_2 = 0.4$; luego debemos hallar y_2 que aproxima al valor $y(0.4)$. Para ello primero debemos hallar y_1, teniendo presente que $y_0 = 0.5$ y $t_0 = 0$:

$$\begin{aligned}
y_1 &= y_0 + h(y_0 - t_0^2 + 1) + \frac{h^2}{2}(y_0 - t_0^2 - 2t_0 + 1) + \left(\frac{h^3}{6} + \frac{h^4}{24}\right)(y_0 - t_0^2 - 2t_0 - 1)\\
&= 0.5 + (0.2)(1.5) + (0.02)(1.5) + (0.0014)(-0.5)\\
&= 0.8293 \quad (\text{Valor exacto } y(0.2) = 0.8292986)
\end{aligned}$$

Hallemos y_2, teniendo presente que $y_1 = 0.8293$ y $t_1 = 0.2$:

$$\begin{aligned}
y_2 &= y_1 + h(y_1 - t_1^2 + 1) + \frac{h^2}{2}(y_1 - t_1^2 - 2t_1 + 1) + \left(\frac{h^3}{6} + \frac{h^4}{24}\right)(y_1 - t_1^2 - 2t_1 - 1)\\
&= 0.8293 + (0.2)(1.7893) + (0.02)(1.3893) + (0.0014)(-0.6107)\\
&= 1.2141 \quad (\text{Valor exacto } y(0.4) = 1.2140877)
\end{aligned}$$

En otras palabras, el método de Taylor de orden 4 obtuvo $y_2 = 1.2141$ como valor aproximado de $y(0.4)$. Se observa que el error cometido por el método en la aproximación hallada es:

$$|y(0.4) - y_2| = 1.23 \times 10^{-5} < h^4 = 1.6 \times 10^{-3}$$

como era de esperarse, ya que el orden del método de Taylor utilizado es $\mathbf{O}(h^4)$ y $h = 0.2$.

■

En general trabajar con métodos de Taylor de órdenes elevados es complicado y algunas veces tedioso, ya que se deben calcular muchas derivadas. En la práctica, se utilizan métodos de Taylor cuyo orden es a lo sumo 4. El método de Taylor de orden 1 se denomina *método de Euler* y es uno de los métodos más famosos y antiguos para resolver problemas de valor inicial.

Método de Euler

La ecuación en diferencias del método de Euler es

$$y_{n+1} = y_n + hf(t_n,y_n), \qquad n = 0,1,\ldots,N-1. \tag{7.11}$$

El método de Euler es un método de Taylor de orden 1, que se obtiene tomando solo los dos primeros sumandos del desarrollo Taylor de $y(t+h)$ alrededor de $t-h$. Veamos ahora la interpretación gráfica de la ecuación de Euler. El método de Euler también puede derivarse considerando la equivalencia del problema (7.2) con la ecuación integral (7.3). Efectivamente, de la ecuación (7.3) se infiere que la solución $y(t)$ (7.2) satisface

$$y(t_{n+1}) = y(t_n) + \int_{t_n}^{t_{n+1}} f(\tau, y(\tau))\, d\tau. \tag{7.12}$$

Así, tomando el área del rectángulo con vértices en los puntos $(t_n, 0)$, $(t_n, f(t_n, y(t_n)))$, $(t_{n+1}, 0)$ y $(t_{n+1}, f(t_{n+1}, y(t_n)))$, como aproximación de la integral de (7.12) (ver Figura 7.1), obtenemos:

$$y(t_{n+1}) \approx y(t_n) + (t_{n+1} - t_n) f(t_n, y(t_n)).$$

Como $h = t_{n+1} - t_n$, entonces tomando y_n en lugar de $y(t_n)$, la ecuación anterior adquiere la forma:

$$y(t_{n+1}) \approx y_n + h f(t_n, y_n),$$

que es la ecuación en diferencias del método de Euler. La integral de la ecuación (7.12) representa el área entre la curva $f(t, y(t))$ y el eje de las abscisas, que se muestra en la Figura 7.1. La ecuación de diferencias de Euler aproxima esta área por el área del rectángulo coloreado de azul. El área entre el rectángulo y la gráfica representa el error de truncamiento local.

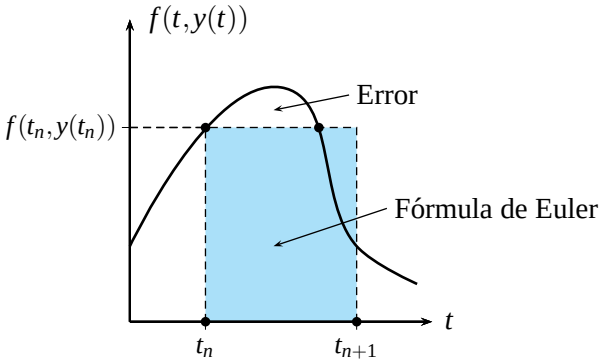


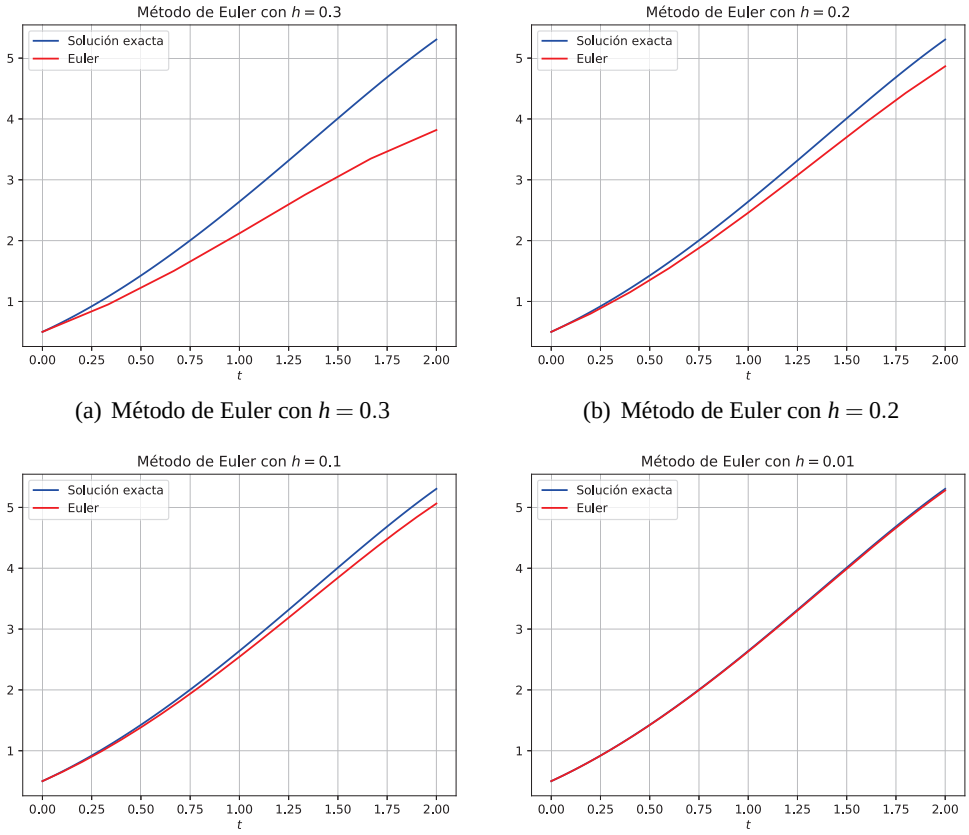
Figura 7.1. Representación gráfica del método de Euler

De esta forma, el error de truncamiento local en la aproximación y_{n+1} será mayor cuanto más grande sea la longitud de paso h; además, el error de truncamiento global será muy grande cuanto mayor sea el número de pasos ya calculados, es decir, cuanto más lejos nos encontremos del punto inicial (t_0, y_0). El método de Euler rara vez se usa en la práctica debido a su ineficiencia computacional. En el Programa 7.1 se muestra la implementación del método de Euler.

Programa 7.1. Método de Euler

```
def euler(t0,y0,f,h,tf):
    N = int((tf-t0)/h)
    t = np.linspace(t0,tf,num=N+1)
    y = np.empty(N+1)
    y[0] = y0
    for n in range(N):
        y[n+1] = y[n] + h*f(t[n],y[n])
    return t,y
```

Ejemplo 7.3. Consideremos el problema de valor inicial del Ejemplo 7.2, cuya solución exacta es $y(t) = t^2 + 2t + 1 - \frac{1}{2}e^t$. Veamos el comportamiento numérico del método de Euler. En la Figura 7.2 se aprecia el comportamiento del método de Euler para las longitudes de paso $h = 0.3, 0.2, 0.1, 0.01$.



(a) Método de Euler con $h = 0.3$

(b) Método de Euler con $h = 0.2$

(c) Método de Euler con $h = 0.1$

(d) Método de Euler con $h = 0.01$

Figura 7.2. Comportamiento del método de Euler para el problema $y'(t) = y - t^2 + 1$, con $y(0) = 0.5$, en el intervalo $t \in [0,2]$.

Las gráfica fueron generadas con el siguiente código

```
t0,y0,tf = 0.,0.5,2.
tk = np.linspace(t0,tf,num=100)
sk = sol(tk)
l = 1
for h in [0.3,0.2,0.1,0.01]:
    tn,yn = euler(t0,y0,f,h,tf)
    plt.figure(l)
    plt.plot(tk,sk,'b-',label='Solución exacta')
    plt.plot(tn,yn,'r-',label='Euler')
    titulo = 'Método de Euler con ' + '$h=' + str(h) + '$'
    plt.title(titulo)
    plt.xlabel('$t$')
    plt.grid()
    plt.legend()
    l +=1
plt.show()
```

Apreciamos que Euler aproxima mejor a la solución exacta a medida que la longitud de paso se hace más pequeña, pero esto requiere de un mayor costo computacional. Para $h = 0.3$, $h = 0.2$, $h = 0.1$ y $h = 0.01$, Euler necesita respectivamente de 6, 10, 20 y 200 nodos t_n, para obtener la aproximación deseada.

7.2.2 Métodos de Runge-Kutta

Los métodos de Taylor requieren de las fórmulas f_t, f_{tt}, etc., para definir la ecuación en diferencias. Este hecho es una gran desventaja de estos métodos. En cambio, los métodos de Runge-Kutta (o, simplemente métodos RK) evaden esta dificultad; sin embargo, imitan a los métodos de Taylor. Generalmente los métodos RK imitan a un método de Taylor de orden m; en este caso, los métodos RK poseen un error de truncamiento global de orden $\mathbf{O}(h^m)$. Mostremos este hecho con los métodos RK de segundo orden o *métodos RK2*.

Consideremos la expansión de Taylor de $y(t+h)$ alrededor de $(t-h)$,

$$y(t+h) = y(t) + hy'(t) + \frac{h^2}{2!}y''(t) + \mathbf{O}(h^3).$$

Como $y'(t) = f(t,y(t))$ e $y''(t) = f_t(t,y(t)) + f_x(t,y(t))f(t,y(t))$, entonces:

$$\begin{aligned} y(t+h) =& y(t) + h\,f(t,y(t)) + \frac{1}{2}h^2(f_t(t,y(t)) + f_x(t,y(t))f(t,y(t))) + O(h^3) \\ =& y(t) + \frac{1}{2}h\,f(t,y(t)) + \frac{1}{2}h(f(t,y(t)) + h\,f_t(t,y(t)) + h\,f_x(t,y(t))f(t,y(t))) + O(h^3) \end{aligned}$$

Utilizando los primeros términos de la serie de Taylor de dos variables de f,

$$f(t+h,y+hf(t,y(t))) = f(t,y(t)) + h\,f_t(t,y(t)) + hf(t,y(t))\,f_x(t,y(t)) + \mathbf{O}(h^2),$$

obtenemos

$$y(t+h) = y(t) + \frac{h}{2}\,f(t,y(t)) + \frac{h}{2}\,f(t+h,y(t)+hf(t,y(t))) + \mathbf{O}(h^3).$$

De esta forma,

$$y(t+h) \approx y(t) + \frac{h}{2}\,f(t,y(t)) + \frac{h}{2}\,f(t+h,y(t)+hf(t,y(t))).$$

Este método se conoce como *método de Heun*, cuya ecuación en diferencias es

$$\begin{aligned} y_{n+1} &= y_n + \frac{h}{2}(K_1+K_2), \qquad n = 0,1,\ldots,N-1, \\ K_1 &= f(t_n,y_n), \\ K_2 &= f(t_n+h,y_n+hK_1). \end{aligned} \tag{7.13}$$

A través de un procedimiento similar al empleado anteriormente, se pueden deducir los siguientes métodos RK2:

$$\begin{aligned} y_{n+1} &= y_n + hK_2, \qquad n = 0,1,\ldots,N-1, \\ K_1 &= f(t_n,y_n), \\ K_2 &= f\left(t_n+\frac{1}{2}h,y_n+\frac{h}{2}K_1\right), \end{aligned} \tag{7.14}$$

$$
\begin{aligned}
y_{n+1} &= y_n + h\left(\frac{1}{3}K_1 + \frac{2}{3}K_2\right), \qquad n = 0, 1, \ldots, N-1, \\
K_1 &= f(t_n, y_n), \\
K_2 &= f\left(t_n + \frac{3}{4}h, y_n + \frac{3}{4}hK_1\right).
\end{aligned} \tag{7.15}
$$

El método dado por (7.14) se denomina *método de Euler modificado* y el método dado por (7.15) se denomina *método de Ralston*. Todos los métodos RK2 son orden $\mathbf{O}(h^2)$ o, simplemente, de segundo orden. La implementación de los métodos de Heun, Euler modificado y Ralston, se muestra en los Programas 7.2, 7.3 y 7.4, respectivamente.

Programa 7.2. Método de Heun

```
def heun(t0,y0,f,h,tf):
    N = int((tf-t0)/h)
    t = np.linspace(t0,tf,num=N+1)
    y = np.empty(N+1)
    y[0] = y0
    for n in range(N):
        k1 = h*f(t[n],y[n])
        k2 = h*f(t[n]+h,y[n]+k1)
        y[n+1] = y[n] + (k1+k2)/2
    return t,y
```

Programa 7.3. Método de Euler modificado

```
def euler_m(t0,y0,f,h,tf):
    N = int((tf-t0)/h)
    t = np.linspace(t0,tf,num=N+1)
    y = np.empty(N+1)
    y[0] = y0
    for n in range(N):
        k1 = h*f(t[n],y[n])
        k2 = h*f(t[n]+h/2,y[n]+k1/2)
        y[n+1] = y[n] + k2
    return t,y
```

Programa 7.4. Método de Ralston

```
def ralston(t0,y0,f,h,tf):
    N = int((tf-t0)/h)
    t = np.linspace(t0,tf,num=N+1)
    y = np.empty(N+1)
    y[0] = y0
    for n in range(N):
        k1 = h*f(t[n],y[n])
        k2 = h*f(t[n]+3*h/4,y[n]+3*k1/4)
        y[n+1] = y[n] + (k1+2*k2)/3
    return t,y
```

En su forma más general, un método RK se puede escribir como:

$$
y_{n+1} = y_n + hF(t_n, y_n, h; f), \qquad n \geq 0 \tag{7.16}
$$

donde F es la *función de incremento* definida de la siguiente manera

$$\begin{aligned} F(t_n, y_n, h; f) &= \sum_{i=1}^{s} b_i K_i, \\ K_i &= f\left(t_n + c_i h, y_n + h \sum_{j=1}^{s} a_{ij} K_j\right) \end{aligned} \tag{7.17}$$

y s denota el número de *estados* del método. Los coeficientes $\{a_{ij}\}$, $\{b_i\}$ y $\{c_i\}$ caracterizan completamente a un método RK y generalmente se disponen en el llamado *tablero de Butcher*

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \cdots & a_{1s} \\ c_2 & a_{21} & a_{22} & \cdots & a_{2s} \\ \vdots & \vdots & & & \vdots \\ c_s & a_{s1} & a_{s2} & \cdots & a_{ss} \\ \hline & b_1 & b_2 & \cdots & b_s \end{array} \quad \text{o} \quad \begin{array}{c|c} \mathbf{c} & A \\ \hline & \mathbf{b}^T \end{array}$$

donde $A = (a_{ij}) \in \mathbb{R}^{s\times s}$, $\mathbf{b} = (b_1, \ldots, b_s) \in \mathbb{R}^s$ y $\mathbf{c} = (c_1, \ldots, c_s) \in \mathbb{R}^s$. Si los coeficientes a_{ij} en A son iguales a cero para $j \geq i$, con $i = 1, \ldots, s$, entonces cada K_i puede calcularse explícitamente en términos de los $i-1$ coeficientes $K_1, \ldots, K_{i-1}$ que ya han sido determinados. En tal caso, el método RK se denomina *explícito*. De lo contrario, éste se denomina *implícito* y es necesario resolver un sistema de ecuaciones no lineales de tamaño s para calcular los coeficientes K_i. Nosotros nos centraremos en los métodos RK explícitos.

A manera de ejemplo, seguidamente mostramos el tablero de Butcher de los métodos de Heun, Euler modificado y Ralston.

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

Tablero de Butcher del método de Heun

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ \hline & 0 & 1 \end{array}$$

Tablero de Butcher del método de Euler modificado

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \frac{3}{4} & \frac{3}{4} & 0 \\ \hline & \frac{1}{3} & \frac{2}{3} \end{array}$$

Tablero de Butcher del método de Ralston

Observe que los métodos de Heun, Euler modificado y de Ralston satisfacen la siguiente condición:

$$c_i = \sum_{j=1}^{s} a_{ij}, \qquad i = 1, \ldots, s.$$

Todos los métodos RK que describiremos satisfacen esta condición.

El siguiente resultado establece una relación entre el orden y el número de estados de los métodos RK explícitos.

Teorema 7.4. *El orden de un método RK explícito de s estados no puede ser mayor que s. Además, no existen métodos RK explícitos de $s \geq 5$ estados con orden s.*

Remitimos al lector a la referencia [3] para la prueba de este resultado. Para órdenes que varían entre 1 y 10, se muestra a continuación el número mínimo de estados (s_{min}) requeridos para obtener un método del orden correspondiente.

Orden	1	2	3	4	5	6	7	8
s_{min}	1	2	3	4	6	7	9	11

Observe que 4 es el número máximo de estados para los cuales el orden del método no es menor que el número de estados. Un ejemplo de un método RK de cuarto orden se proporciona mediante el siguiente método explícito de 4 estados, que usualmente se denomina *método de Runge-Kutta* o *método RK4*.

$$
\begin{aligned}
y_{n+1} &= y_n + \frac{h}{6}(K_1 + 2K_2 + 2K_3 + K_4), \qquad n = 0, 1, \ldots, N-1, \\
K_1 &= f(t_n, y_n), \\
K_2 &= f\left(t_n + \frac{1}{2}h, y_n + \frac{1}{2}hK_1\right), \\
K_3 &= f\left(t_n + \frac{1}{2}h, y_n + \frac{1}{2}hK_2\right), \\
K_4 &= f\left(t_n + h, y_n + hK_3\right).
\end{aligned}
\tag{7.18}
$$

El tablero de Butcher del método RK4 se muestra a continuación.

0	0	0	0	0
$\frac{1}{2}$	$\frac{1}{2}$	0	0	0
$\frac{1}{2}$	0	$\frac{1}{2}$	0	0
1	0	0	1	0
	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{6}$

Tablero de Butcher del método RK4

En el Programa 7.5 se muestra la implementación del método RK4.

Programa 7.5. Método de RK4

```
def rk4(t0,y0,f,h,tf):
    N = int((tf-t0)/h)
    t = np.linspace(t0,tf,num=N+1)
    y = np.empty(N+1)
    y[0] = y0
    for n in range(N):
        k1 = h*f(t[n],y[n])
        k2 = h*f(t[n]+h/2,y[n]+k1/2)
        k3 = h*f(t[n]+h/2,y[n]+k2/2)
        k4 = h*f(t[n]+h,y[n]+k3)
        y[n+1] = y[n] + (k1+2*k2+2*k3+k4)/6
    return t,y
```

Métodos RK Adaptativos

El principal inconveniente del método RK4 es que no se puede estimar fácilmente el error de truncamiento local. Por lo que debemos adivinar la longitud de paso de integración h o determinarla

por prueba y error. Por el contrario, los llamados *métodos adaptativos* pueden evaluar el error de truncamiento en cada paso de integración y ajustar el valor de h en consecuencia (pero a un costo de computación más elevado). En sí, los métodos adaptativos emplean un estimador de errores locales a *priori*, pero estas herramientas son sumamente costosas y su implementación no es fácil. Por lo general, se utilizan estimadores de error local a *posteriori*. El estimador de errores locales a posteriori se puede construir de dos maneras:

i) Usando el mismo método RK, pero con dos pasos diferentes (típicamente $2h$ y h).

ii) Utilizando dos métodos RK de orden diferente, pero con el mismo número de estados.

El primer enfoque consiste en generar una primera aproximación $\widehat{y}_{n+1}$ tomando la longitud de paso $2h$ y luego se utiliza $\widehat{y}_{n+1}$ para hallar y_{n+1}, con la longitud de paso h. Este enfoque produce un aumento considerable en el esfuerzo computacional, debido a las $s-1$ evaluaciones funcionales adicionales necesarias para generar el valor y_{n+1}. Debido a estos inconvenientes, el enfoque dos se usa con mayor frecuencia.

Una alternativa que no requiere evaluaciones funcionales adicionales consiste en usar simultáneamente dos métodos RK diferentes con un mismo número de estados s y de orden p y $p+1$, respectivamente, que comparten el mismo conjunto de valores K_i. Estos métodos están representados sintéticamente por el siguiente tablero de Butcher modificado:

$$\begin{array}{c|c} \mathbf{c} & A \\ \hline & \mathbf{b}^T \\ & \widehat{\mathbf{b}}^T \\ \hline & \mathbf{d}^T \end{array} \tag{7.19}$$

donde el método de orden p es identificado por los coeficientes $\mathbf{c}$, A y $\mathbf{b}$, mientras que el de orden $p+1$ es identificado por $\mathbf{c}$, A y $\widehat{\mathbf{b}}$, y $\mathbf{d} = \mathbf{b} - \widehat{\mathbf{b}}$ representa el vector de diferencia.

La diferencia entre las soluciones aproximadas en t_{n+1} obtenidas por los dos métodos, proporciona una estimación del error de truncamiento local, cuyo orden corresponde al método inferior. Por otro lado, dado que los coeficientes K_i coinciden, esta diferencia viene dada por $h\sum_{i=1}^{s} d_i K_i$, y por lo tanto no requiere evaluaciones funcionales adicionales.

Observe que, si la solución y_{n+1} calculada por el método de orden p se usa para inicializar el método en el nodo t_{n+2}, el método en conjunto tendrá orden p. Si, por el contrario, se emplea la solución calculada por el método de orden $p+1$, el método resultante tiene orden $p+1$.

El *método de Runge-Kutta Fehlberg de cuarto orden* es uno de los esquemas más populares que tiene la forma (7.19) y consiste en un método RK de cuarto orden junto con un método RK de quinto orden. Por este motivo, se lo conoce como el *método RK45*. El tablero de Butcher modificado para este método se muestra a continuación.

0	0	0	0	0	0	0
$\frac{1}{4}$	$\frac{1}{4}$	0	0	0	0	0
$\frac{3}{8}$	$\frac{3}{32}$	$\frac{9}{32}$	0	0	0	0
$\frac{12}{13}$	$\frac{1932}{2197}$	$-\frac{7200}{2197}$	$\frac{7296}{2197}$	0	0	0
1	$\frac{439}{216}$	-8	$\frac{3680}{513}$	$-\frac{845}{4104}$	0	0
$\frac{1}{2}$	$-\frac{8}{27}$	2	$-\frac{3544}{2565}$	$\frac{1859}{4104}$	$-\frac{11}{40}$	0
	$\frac{25}{216}$	0	$\frac{1408}{2565}$	$\frac{2197}{4104}$	$-\frac{1}{5}$	0
	$\frac{16}{135}$	0	$\frac{6656}{12825}$	$\frac{28561}{56430}$	$-\frac{9}{50}$	$\frac{2}{55}$
	$\frac{1}{360}$	0	$-\frac{128}{4275}$	$-\frac{2197}{75240}$	$\frac{1}{50}$	$\frac{2}{55}$

Tablero de Butcher modificado del método RK45

El método RK45 tiene un procedimiento para determinar el tamaño apropiado de la longitud de paso h. La idea es comenzar con una longitud de paso moderada. Cuando se detecte que el error es mayor que una tolerancia ε, se reduce la longitud de paso y se vuelve a calcular el paso actual. Cuando se detecte que el error es menor que ε, se mantiene el paso actual y se aumenta ligeramente la longitud de paso para obtener la siguiente aproximación. En cada paso n del método RK45 se realiza lo siguiente:

- Se calculan los seis valores:

$$
\begin{aligned}
K_1 &= f(t_n, y_n),\\
K_2 &= f\left(t_n + \frac{1}{4}h, y_n + \frac{1}{4}hK_1\right),\\
K_3 &= f\left(t_n + \frac{3}{8}h, y_n + h\left(\frac{3}{32}K_1 + \frac{9}{32}K_2\right)\right),\\
K_4 &= f\left(t_n + \frac{12}{13}h, y_n + h\left(\frac{1932}{2197}K_1 - \frac{7200}{2197}K_2 + \frac{7296}{2197}K_3\right)\right),\\
K_5 &= f\left(t_n + h, y_n + h\left(\frac{439}{216}K_1 - 8K_2 + \frac{3680}{513}K_3 - \frac{845}{4104}K_4\right)\right),\\
K_6 &= f\left(t_n + \frac{1}{2}h, y_n + h\left(-\frac{8}{27}K_1 + 2K_2 - \frac{3544}{2565}K_3 + \frac{1859}{4104}K_4 - \frac{11}{40}K_5\right)\right),
\end{aligned}
$$

- Luego, se calcula una aproximación de $y(t_n)$ usando un método RK de orden 4:

$$w_{n+1} = y_n + h\left(\frac{25}{216}K_1 + \frac{1408}{2565}K_3 + \frac{2197}{4104}K_4 - \frac{1}{5}K_5\right).$$

- Después se calcula otra aproximación de $y(t_n)$ empleando un método RK de orden 5:

$$z_{n+1} = y_n + h\left(\frac{16}{135}K_1 + \frac{6656}{12825}K_3 + \frac{28561}{56430}K_4 - \frac{9}{50}K_5 + \frac{2}{52}K_6\right).$$

- Se calcula el error

$$e_n = |w_{n+1} - z_{n+1}| = h\left(\frac{1}{360}K_1 - \frac{128}{4275}K_3 - \frac{2197}{75240}K_4 + \frac{1}{50}K_5 + \frac{2}{55}K_6\right) \tag{7.20}$$

y el escalar

$$\delta = \left(\frac{\varepsilon h}{2e_n}\right)^{1/4} \approx 0.84\left(\frac{\varepsilon h}{e_n}\right)^{1/4}. \tag{7.21}$$

- Si $e_n \leq \varepsilon$, entonces se toma $y_{n+1} = w_{n+1}$, se calcula

$$h_{aux} = \begin{cases} h_{min}, & \delta h < h_{min}, \\ \delta h, & h_{min} \leq \delta h \leq h_{max}, \\ h_{max}, & \delta h > h_{max}, \end{cases}$$

y se actualiza la longitud de paso $h = h_{aux}$. De lo contrario, si $e_n > \varepsilon$, se recalcula la aproximación actual con la nueva longitud de paso $h = h_{aux}$.

La derivación de la fórmula (7.21) se puede encontrar en libros avanzados de Análisis Numérico. Tenga en cuenta que la cantidad $|w_{n+1} - z_{n+1}|$ es el error de truncamiento local dado por la ecuación (7.20). Para garantizar una precisión computacional razonable sin demasiadas evaluaciones por paso, se imponen un mínimo (h_{min}) y un máximo (h_{max}) en la longitud del paso. Si $\delta h \in [h_{min}, h_{max}]$, se acepta el valor de $h = \delta h$. Si δh es más pequeño que h_{min}, elegimos $h = h_{min}$. Si es mayor que h_{max}, seleccionamos $h = h_{max}$. Es importante resaltar que una longitud de paso fija no es la mejor estrategia, aunque proporcione una tabla de valores de apariencia más agradable. Si se necesitasen valores que no estuvieran en la tabla, se debe utilizar la interpolación polinómica. En el Programa 7.6 se muestra la implementación del método RK45.

Programa 7.6. Método de RK45

```
def rk45(t0,y0,f,h0,tf,tol=1.0e-6,hmin=0.01,hmax=0.1):
    t,y = [],[]
    y.append(y0)
    t.append(t0)
    h,tn,w,n = h0,t0,y0,0
    while tn<tf:
      h = min(h,tf-tn)
      k1 = h*f(tn,w);
      k2 = h*f(tn+h/4,y[n]+k1/4)
      k3 = h*f(tn+3*h/8,y[n]+3*k1/32+9*k2/32)
      k4 = h*f(tn+12*h/13,y[n]+1932*k1/2197-7200*k2/2197+7296*k3/2197)
      k5 = h*f(tn+h,y[n]+439*k1/216-8*k2+3680*k3/513-845*k4/4104)
      k6 = h*f(tn+h/2,y[n]-8*k1/27+2*k2-3544*k3/2565+1859*k4/4104
           -11*k5/40)
      en = np.abs(k1/360-128*k3/4275-2197*k4/75240+k5/50+2*k6/55)
      delta = 0.84*(tol*h/en)**0.25
      if en <= tol:
         tn = tn+h
         t.append(tn)
         w = y[n] + 25*k1/216+1408*k3/2565+2197*k4/4104-k5/5
          y.append(w)
         n +=1
      h = max(hmin,min(delta*h,hmax))
    return np.array(t),np.array(y)
```

Ejemplo 7.4. Comparemos los métodos: Heun, Euler modificado, Ralston, RK4 y RK45, en la resolución del problema de valor inicial

$$\begin{cases} y'(t) = t^2 y(t) + 2t^2, & t \in [0,1], \\ y(0) = 1, \end{cases}$$

cuya solución exacta es $y(t) = 3e^{t^3/3} - 2$. Para los métodos Heun, Euler modificado (Euler), Ralston y RK4 escogimos $h = 0.1$ como longitud de paso. Para el método RK45 escogimos $h = 0.1$ como longitud de paso inicial, una tolerancia $\texttt{tol} = 10^{-6}$, y $h_{min} = 0.01$ y $h_{max} = 0.1$, para las cotas inferior y superior de la longitud de paso. Corrimos las funciones `euler_m`, `heun`, `ralston`, `rk4` y `rk45`, y generamos los resultados que se muestran en las Tablas 7.1 y 7.2. Aquí reportamos las aproximaciones y_n halladas por cada método y el error relativo

$$e_n = \frac{|y(t_n) - y_n|}{|y(t_n)|}, \quad n = 0, 1, \ldots, N.$$

Tabla 7.1. Resumen de las cálculos del Ejemplo 7.4 para los métodos Euler modificado, Heun, Ralston y RK4

	Euler Modificado		Heun		Ralston		RK4	
t_n	y_n^{Euler}	e_{Euler}	y_n^{Heun}	e_{Heun}	$y_n^{Ralston}$	$e_{Ralston}$	y_n^{RK4}	e_{RK4}
0.00	1.000000	0.000000	1.000000	0.000000	1.000000	0.000000	1.00000000	0.00000000
0.10	1.000750	0.000250	1.001500	0.000499	1.001125	0.000125	1.00100016	0.00000001
0.20	1.007505	0.000502	1.009010	0.000991	1.008257	0.000245	1.00801066	0.00000002
0.30	1.026340	0.000762	1.028622	0.001461	1.027480	0.000349	1.02712183	0.00000003
0.40	1.063579	0.001041	1.066698	0.001889	1.065137	0.000422	1.06468750	0.00000004
0.50	1.126113	0.001355	1.130179	0.002251	1.128142	0.000445	1.12764066	0.00000005
0.60	1.221860	0.001721	1.227058	0.002526	1.224453	0.000398	1.22396596	0.00000006
0.70	1.360434	0.002159	1.367054	0.002697	1.363733	0.000261	1.36337721	0.00000007
0.80	1.554089	0.002693	1.562572	0.002751	1.558314	0.000018	1.55828580	0.00000010
0.90	1.819089	0.003351	1.830093	0.002678	1.824564	0.000351	1.82520559	0.00000016
1.00	2.177721	0.004169	2.192228	0.002465	2.184934	0.000870	2.18683666	0.00000028

Tabla 7.2. Resumen de las cálculos del Ejemplo 7.4 para el método RK45

	RK45	
t_n	y_n^{RK45}	e_{RK45}
0.00	1.00000000	0.00000000
0.10	1.00100016	0.00000000
0.20	1.00801067	0.00000001
0.30	1.02712185	0.00000002
0.40	1.06468752	0.00000003
0.50	1.12764067	0.00000004
0.60	1.22396597	0.00000005
0.70	1.36337722	0.00000007
0.80	1.55828583	0.00000008
0.90	1.82520571	0.00000009
0.99	2.16338369	0.00000008
1.00	2.18683709	0.00000008

En estas tablas se proporciona un resumen de todos los cálculos, donde se percibe que el método RK45 posee el menor % de error relativo global.

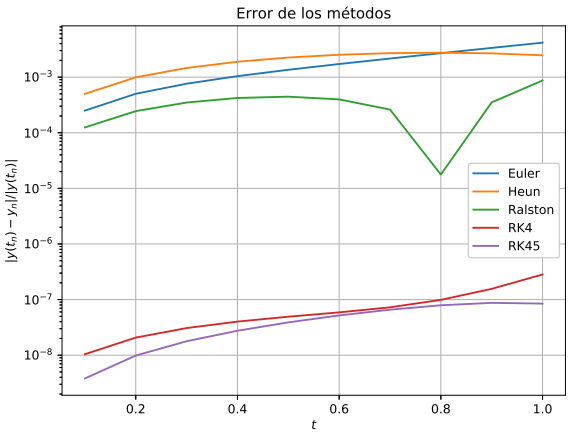


Figura 7.3. Tasa de convergencia de los métodos Heun, Euler modificado (Euler), Ralston, RK4 y RK45 para el problema de valor inicial del Ejemplo 7.4.

Como se esperaba, comenzando con el método de Euler modificado, que es de hecho un método RK2, la precisión mejora con el orden del método RK45. El comportamiento del error de los métodos se puede apreciar gráficamente en la Figura 7.3.

7.3 Métodos Multipaso

En esta sección describimos los métodos multipaso, es decir, métodos que pueden considerar valores previos a $y(t)$, por ejemplo $y(t-h)$, o valores posteriores, por ejemplo $y(t+2h)$, para aproximar el valor de $y(t+h)$. Recordemos que deseamos resolver el problema de valor inicial (7.2), en particular, deseamos conocer el valor de $y(t)$ en $t = t_n$, donde $t_n = t_0 + nh$ y $y_n \approx y(t_n)$, para $i = 0, 1, \ldots, N$.

Una técnica ampliamente usada para derivar los métodos multipaso es aplicar cuadraturas particulares para aproximar la integral de la ecuación (7.3). Se procede de la siguiente manera. La solución de (7.2) satisface

$$\int_{t_n}^{t_{n+1}} y'(\tau)\,d\tau = y(t_{n+1}) - y(t_n),$$

luego,

$$y(t_{n+1}) = y(t_n) + \int_{t_n}^{t_{n+1}} f(\tau, y(\tau))\,d\tau.$$

De esta forma, podemos utilizar una cuadratura para aproximar el valor de la integral de la ecuación anterior, es decir,

$$\int_{t_n}^{t_{n+1}} f(\tau, y(\tau))\,d\tau \approx \sum c_j f(t_j, y(t_j)). \tag{7.22}$$

Por lo tanto, en su forma más general, la ecuación en diferencias de un método multipaso se puede expresar como:

$$y(t_{n+1}) \approx y(t_n) + \sum c_j f(t_j, y(t_j)), \tag{7.23}$$

donde los coeficientes c_j y los nodos t_j dependen de los valores utilizados en la cuadratura. En las subsiguientes secciones describiremos las *fórmulas de Adams-Bashforth* y *Adams-Moulton*, que se generan a partir de cuadraturas específicas.

7.3.1 Fórmulas de Adams-Bashforth

Utilizando cuadraturas particulares (por ejemplo, la regla del trapecio, Simpson o alguna otra cuadratura que se obtenga por la interpolación de la función $f(t,y(t))$ en nodos igualmente espaciados), la fórmula (7.23) adquiere la forma:

$$y_{n+1} = y_n + c_0 f_n + c_1 f_{n-1} + c_2 f_{n-2} + \cdots + c_k f_{n-k}, \quad n \geq k. \tag{7.24}$$

donde f_n denota $f(t_n,y_n)$. Una expresión de este tipo se conoce como *fórmula de Adams-Bashforth de* $(k+1)$ *pasos* o simplemente *método ABk* $+1$. Por ejemplo, la expresión

$$y_{n+1} = y_n + \frac{h}{720}\left[1901 f_n - 2774 f_{n-1} + 2616 f_{n-2} - 1274 f_{n-3} + 251 f_{n-4}\right],$$

para $n \geq 4$, se denomina *fórmula de Adams-Bashforth de 5 pasos* o método AB5. El lector pudiera estar preguntándose: ¿cómo se determinaron los coeficientes c_j? Describamos un procedimiento para calcular tales coeficientes, el cual consiste en aplicar el método de los coeficientes indeterminados a la ecuación (7.22) (ver Capítulo 5). Como la fórmula de Adams-Bashforth que deseamos construir es de 5 pasos, entonces la cuadratura que debe utilizarse para obtener la ecuación (7.24) es de la forma

$$\int_{t_{n-4}}^{t_{n+1}} f(t,y(t))\,dt \approx c_0 f_n + c_1 f_{n-1} + c_2 f_{n-2} + c_3 f_{n-3} + c_4 f_{n-4},$$

además, debe ser exacta para polinomios de grado menor o igual 4. Sin pérdida de generalidad podemos suponer que $t_n = 0$ y $h = 1$. De esta forma, aplicando el método de los coeficientes indeterminados obtenemos el sistema de ecuaciones lineales

$$\begin{cases} c_0 + c_1 + c_2 + c_3 + c_4 &= 1 \\ -c_1 - 2c_2 - 3c_3 - 4c_4 &= \frac{1}{2} \\ 2c_2 + 6c_3 + 12c_4 &= \frac{5}{6} \\ -6c_3 - 24c_4 &= \frac{9}{4} \\ 24c_4 &= \frac{251}{30} \end{cases}$$

cuya solución es:

$$c_0 = \frac{1901}{720}, \quad c_1 = -\frac{2774}{720}, \quad c_2 = \frac{2616}{720}, \quad c_3 = -\frac{1274}{720}, \quad c_4 = \frac{251}{720}.$$

Por este mismo procedimiento se pueden generar otros métodos AB, que a continuación listamos.

- *Adams-Bashforth de 2 pasos o AB2*

$$y_{n+1} = y_n + \frac{h}{2}\left[3 f_n - f_{n-1}\right], \quad n \geq 1.$$

- *Adams-Bashforth de 3 pasos o AB3*

$$y_{n+1} = y_n + \frac{h}{12}\left[23 f_n - 16 f_{n-1} + 5 f_{n-2}\right], \quad n \geq 2.$$

- *Adams-Bashforth de 4 pasos o AB4*

$$y_{n+1} = y_n + \frac{h}{24}\left[55 f_n - 59 f_{n-1} + 37 f_{n-2} - 9 f_{n-3}\right], \quad n \geq 3.$$

Es muy importante resaltar que para aplicar un método AB de $(k+1)$ pasos, se deben conocer $k+1$ valores de arranque, $y_0, y_1, \ldots, y_k$, con los que se puede calcular y_{k+1} y de allí calcular los siguientes valores. Por ello, los métodos AB no se emplean en forma aislada. Generalmente se usa un método de un paso para hallar los valores $y_1, \ldots, y_k$ y después se aplica un método AB.

7.3.2 Fórmulas de Adams-Moulton

Supongamos que la cuadratura que se emplea en la fórmula (7.23) considera el término f_{n+1}, en otras palabras, tiene la forma

$$y_{n+1} = y_n + c_0 f_{n+1} + c_1 f_n + c_2 f_{n-1} + \cdots + c_k f_{n-k+1}, \quad n \geq k+1. \tag{7.25}$$

Una expresión de este tipo se conoce como *fórmula de Adams-Moulton de k pasos* o simplemente *método AMk*. Utilizando el método de los coeficientes indeterminados conjuntamente con la ecuación (7.25), se pueden generar distintos métodos AM. Algunos de ellos los listamos a continuación.

- *Adams-Moulton de 2 pasos o AM2*

$$y_{n+1} = y_n + \frac{h}{12}\left[f_{n+1} + 8f_n - f_{n-1}\right], \quad n \geq 1.$$

- *Adams-Moulton de 3 pasos o AM3*

$$y_{n+1} = y_n + \frac{h}{24}\left[9f_{n+1} + 19f_n - 5f_{n-1} + f_{n-2}\right], \quad n \geq 2.$$

- *Adams-Moulton de 4 pasos o AM4*

$$y_{n+1} = y_n + \frac{h}{720}\left[251f_{n+1} + 646f_n - 264f_{n-1} + 106f_{n-2} - 19f_{n-3}\right], \quad n \geq 3.$$

Al resolver un problema de valor inicial (7.2), en cada nodo t_n los métodos implícitos necesitan resolver una ecuación no lineal para hallar y_{n+1} y esto requiere de un gran costo computacional (ver Capítulo 4). Por ello, los métodos AM no se emplean solos, sino que usualmente se usan conjuntamente con un método AB, con lo cual forman un tipo especial de método llamado *predictor-corrector*, que veremos más adelante.

7.3.3 Orden de un Método Multipaso

En general, la ecuación en diferencias de un método de k pasos se puede escribir como:

$$a_k y_n + a_{k-1} y_{n-1} + \cdots + a_0 y_{n-k} = h\left[b_k f_n + b_{k-1} f_{n-1} + \cdots + b_0 f_{n-k}\right], \quad n \geq k. \tag{7.26}$$

donde los coeficientes a_j y b_j son números reales dados, además, se asume que $a_k \neq 0$. Por ejemplo, el método AB5 se puede expresar como:

$$a_5 y_n + a_4 y_{n-1} = h\left[b_4 f_{n-1} + b_3 f_{n-2} + b_2 f_{n-3} + b_1 f_{i-4} + b_0 f_{n-5}\right],$$

donde $k = 5$, y $a_5 = 1$, $a_4 = -1$, $a_3 = 0$, $a_2 = 0$, $a_1 = 0$, $a_0 = 0$, $b_5 = 0$, $b_4 = \frac{1901}{720}$, $b_3 = -\frac{2774}{720}$, $b_2 = \frac{2616}{720}$, $b_1 = -\frac{1274}{720}$, $b_0 = \frac{251}{720}$.

Definición 7.4. Si $b_k = 0$, se dice que el método multipaso (7.26) es *explícito*. Por el contrario, si $b_k \neq 0$, se dice que el método multipaso (7.26) es *implícito*.

Es de notar que todas las fórmulas AB son métodos explícitos y todas las fórmulas AM son métodos implícitos. Pasemos ahora a estudiar el orden de error de los métodos multipaso. Para un entero positivo k (número de pasos del método multipaso), sea L la funcional lineal definida como

$$L(g) = \sum_{i=0}^{k} [a_i g(ih) - hb_i g'(ih)], \tag{7.27}$$

para toda función $g : \mathbb{R} \to \mathbb{R}$ diferenciable. En sí, L le asigna a cada función g un número real. Ahora, sustituyendo las series de Taylor

$$y(ih) = \sum_{j=0}^{\infty} \frac{(ih)^j}{j!} y^{(j)}(0) \qquad \text{e} \qquad y'(ih) = \sum_{j=0}^{\infty} \frac{(ih)^j}{j!} y^{(j+1)}(0)$$

en (7.27), obtenemos

$$L(y) = d_0 y(0) + d_1 h y'(0) + d_2 h^2 y''(0) + \cdots, \tag{7.28}$$

donde

$$\begin{aligned} d_0 &= \sum_{i=0}^{k} a_i, \\ d_1 &= \sum_{i=0}^{k} (ia_i - b_i), \\ &\vdots \\ d_j &= \sum_{i=0}^{k} \left(\frac{i^j}{j!} a_i - \frac{i^{j-1}}{(j-1)!} b_i \right), \qquad j = 1, 2, \ldots. \end{aligned} \tag{7.29}$$

En el siguiente teorema se establece la relación entre el orden del error de truncamiento local de un método multipaso y la funcional lineal L dada por (7.28). La demostración de este teorema se puede encontrar en un libro especializado de Análisis Numérico.

Teorema 7.5. *Las siguientes tres propiedades del método multipaso* (7.26) *son equivalentes:*

i) $d_0 = d_1 = \cdots = d_m = 0$, *donde* d_j *está dado por* (7.29).

ii) $L(p) = 0$ *para todo polinomio* p *de grado menor o igual a* m, *donde* L *está dada por* (7.27).

iii) $L(y)$ *es de orden* $\mathbf{O}(h^{m+1})$ *para toda función* $y(t)$ *que sea* $m+1$*-veces continuamente diferenciable.*

Definición 7.5. El orden del error de truncamiento local del método multipaso (7.26) de k pasos, es el único número natural m tal que $d_0 = d_1 = \cdots = d_m = 0 \neq d_{m+1}$, donde d_j está dado por (7.29).

A manera de ejemplo, veamos que el orden del error de truncamiento local de los métodos AB5 y AM4 es $\mathbf{O}(h^5)$. Iniciemos con el método AB5. Como vimos anteriormente, el método AB5 se

puede expresar de la forma (7.26), donde $k=5$, y $a_5=1$, $a_4=-1$, $a_3=0$, $a_2=0$, $a_1=0$, $a_0=0$, $b_5=0$, $b_4=\frac{1901}{720}$, $b_3=-\frac{2774}{720}$, $b_2=\frac{2616}{720}$, $b_1=-\frac{1274}{720}$, $b_0=\frac{251}{720}$. Así, empleando (7.29) obtenemos:

$$
\begin{aligned}
d_0 &= \sum_{i=0}^{5} a_i = 0,\\
d_1 &= \sum_{i=0}^{5} i a_i - \sum_{i=0}^{5} b_i = 1-1=0,\\
d_2 &= \sum_{i=0}^{5} \frac{i^2}{2!} a_i - \sum_{i=0}^{5} i b_i = \frac{9}{2} - \frac{3240}{720} = 0,\\
d_3 &= \sum_{i=0}^{5} \frac{i^3}{3!} a_i - \sum_{i=0}^{5} \frac{i^2}{2!} b_i = \frac{61}{6} - \frac{7320}{720} = 0,\\
d_4 &= \sum_{i=0}^{5} \frac{i^4}{4!} a_i - \sum_{i=0}^{5} \frac{i^3}{3!} b_i = \frac{369}{24} - \frac{11070}{720} = 0,\\
d_5 &= \sum_{i=0}^{5} \frac{i^5}{5!} a_i - \sum_{i=0}^{5} \frac{i^4}{4!} b_i = \frac{2101}{120} - \frac{12606}{720} = 0,\\
d_6 &= \sum_{i=0}^{5} \frac{i^6}{6!} a_i - \sum_{i=0}^{5} \frac{i^5}{5!} b_i = \frac{11529}{720} - \frac{22583}{1440} = \frac{475}{720}.
\end{aligned}
$$

Como $d_0=d_1=\cdots=d_5=0$ y $d_6\neq 0$, entonces por el Teorema 7.5 el orden del error de truncamiento local del método AB5 es $\mathbf{O}(h^5)$ y por ende el global es $\mathbf{O}(h^4)$.

Demostremos ahora que el orden del error de truncamiento local del método AM4 es $\mathbf{O}(h^5)$. El método de AM4 se puede expresar de la forma (7.26), donde $k=4$, y $a_4=1$, $a_3=-1$, $a_2=0$, $a_1=0$, $a_0=0$, $b_4=\frac{251}{720}$, $b_3=\frac{646}{720}$, $b_2=-\frac{264}{720}$, $b_1=\frac{106}{720}$, $b_0=-\frac{19}{720}$. Así, empleando (7.29) obtenemos:

$$
\begin{aligned}
d_0 &= \sum_{i=0}^{4} a_i = 0,\\
d_1 &= \sum_{i=0}^{4} i a_i - \sum_{i=0}^{4} b_i = 1-1=0,\\
d_2 &= \sum_{i=0}^{4} \frac{i^2}{2!} a_i - \sum_{i=0}^{4} i b_i = \frac{7}{2} - \frac{2520}{720} = 0,\\
d_3 &= \sum_{i=0}^{4} \frac{i^3}{3!} a_i - \sum_{i=0}^{4} \frac{i^2}{2!} b_i = \frac{37}{6} - \frac{4440}{720} = 0,\\
d_4 &= \sum_{i=0}^{4} \frac{i^4}{4!} a_i - \sum_{i=0}^{4} \frac{i^3}{3!} b_i = \frac{175}{24} - \frac{5250}{720} = 0,\\
d_5 &= \sum_{i=0}^{4} \frac{i^5}{5!} a_i - \sum_{i=0}^{4} \frac{i^4}{4!} b_i = \frac{781}{120} - \frac{4686}{720} = 0,\\
d_6 &= \sum_{i=0}^{4} \frac{i^6}{6!} a_i - \sum_{i=0}^{4} \frac{i^5}{5!} b_i = \frac{3367}{720} - \frac{405660}{86400} = -\frac{1620}{720}.
\end{aligned}
$$

Como $d_0=d_1=\cdots=d_5=0$ y $d_6\neq 0$, entonces por el Teorema 7.5 el orden error de truncamiento local del método de AM4 es $\mathbf{O}(h^5)$ y por ende el global es $\mathbf{O}(h^4)$.

7.4 Métodos Predictor-Corrector

Los métodos predictor-corrector son una clase de algoritmos que emplean una combinación de un método explícito y uno implícito para resolver el problema de valor inicial (7.2). Primero, el método explícito se usa para predecir el valor de y_{n+1}, denotando con $y_{n+1}^{(0)}$ el valor predicho. El valor $y_{n+1}^{(0)}$ se usa luego en el lado derecho de un método implícito para obtener un nuevo valor más preciso para y_{n+1} en el lado izquierdo. Este procedimiento se realiza de la siguiente manera iterativa:

Paso 1. Se predice un valor de y_{n+1}, denotado por $y_{n+1}^{(0)}$, usando un método explícito.

Paso 2. Se corrige el valor $y_{n+1}^{(0)}$ en forma iterativa $y_{n+1}^{(1)}, y_{n+1}^{(2)}, \ldots, y_{n+1}^{(k)}$, a través de un método implícito, hasta que dos iteraciones sucesivas cumplan con el siguiente criterio

$$\frac{\left| y_{n+1}^{(k+1)} - y_{n+1}^{(k)} \right|}{\left| y_{n+1}^{(k)} \right|} < \varepsilon, \tag{7.30}$$

donde ε es una tolerancia prestablecida.

Paso 3. Si se cumple el criterio (7.30), se toma $y_{n+1} = y_{n+1}^{(k+1)}$ como aproximación, se incrementa n en 1 y se va al Paso 1.

Cuando se utiliza un método explícito de orden p como predictor y un método implícito de orden q como corrector, entonces el orden del predictor-corrector es q. Para más detalles acerca del orden de un método predictor-corrector se recomienda el libro [25]. A continuación describiremos algunos métodos predictor-corrector de uso frecuente.

Método Predictor-Corrector de Heun

En el método predictor-corrector de Heun, también conocido como *método Euler-trapecio*, se usa el método de Euler como predictor y luego el método de Heun como corrector:

$$\text{Predictor:} \quad y_{n+1}^{(0)} = y_n + h f_n, \quad n = 0, 1, \ldots, N-1$$

$$\text{Corrector:} \quad y_{n+1}^{(k)} = y_n + \frac{h}{2}\left[f(t_{n+1}, y_{n+1}^{(k)}) + f_n \right], \quad k = 1, 2, \ldots$$

En el Programa 7.7 se muestra la implementación del método predictor-corrector de Heun.

Programa 7.7. Método Predictor-Corrector de Heun

```
def pc_heun(t0,y0,f,h,tf,tol=1.0e-6):
    N = int((tf-t0)/h)
    t = np.linspace(t0,tf,num=N+1)
    y = np.empty(N+1)
    y[0] = y0
    fn = []
    h2 = h/2
    for n in range(N):
        fn.append(f(t[n],y[n]))
```

```
        yn1_k = y[n] + h*fn[n]
        while True:
            fn1_k = f(t[n+1],yn1_k)
            w = y[n] + h2*(fn1_k + fn[n])
            if np.abs(w-yn1_k)/np.abs(yn1_k) < tol:
                break
            yn1_k = w
        y[n+1] = w
    return t,y
```

Método Predictor-Corrector de Milne

El método predictor-corrector de Milne utiliza fórmulas basadas en la regla de integración de Simpson, para obtener tanto el predictor como el corrector:

$$\text{Predictor:} \quad y_{n+1}^{(0)} = y_{n-3} + \frac{4}{3}h\left[2f_n - f_{n-1} + 2f_{n-2}\right], \quad n = 3,4,\ldots,N-1$$

$$\text{Corrector:} \quad y_{n+1}^{(k)} = y_{n-1} + \frac{h}{3}\left[f(t_{n+1}, y_{n+1}^{(k)}) + 4f_n + f_{n-1}\right], \quad k = 1,2,\ldots$$

Observe que el método de Milne requiere conocer los valores y_1, y_2, y_3. Para calcular estos valores normalmente se utiliza un método de un paso de orden 4, por ejemplo, el método RK4. En el Programa 7.8 se muestra la implementación del método predictor-corrector de Milne. Aquí usamos la función `rk4` dada en el Programa 7.5.

Programa 7.8. Método Predictor-Corrector de Milne

```
def milne(t0,y0,f,h,tf,tol=1.0e-6):
    N = int((tf-t0)/h)
    t = np.linspace(t0,tf,num=N+1)
    y = np.empty(N+1)
    t_aux, y_aux = rk4(t0,y0,f,h,t0+3*h)
    t[0:4], y[0:4] = t_aux, y_aux
    fn = [f(t[0],y[0]),f(t[1],y[1]),f(t[2],y[2])]
    for n in range(3,N):
        fn.append(f(t[n],y[n]))
        yn1_k = y[n-3] + 4*h*(2*fn[n]-fn[n-1]+2*fn[n-2])/3
        while True:
            fn1_k = f(t[n+1],yn1_k)
            w = y[n-1] + h(fn1_k+4*fn[n]+fn[n-1])/3
            if np.abs(w-yn1_k)/np.abs(yn1_k) < tol:
                break
            yn1_k = w
        y[n+1] = w
    return t,y
```

Métodos Adams–Bashforth–Moulton

Los llamados métodos Adams–Bashforth–Moulton (ABM) son esquemas predictor-corrector que se crean combinando un método AB de un orden particular con un método AM del mismo orden como predictor; por ejemplo, las fórmulas de cuarto orden de estos dos métodos (AB5 y AM4) se pueden combinar para crear el predictor-corrector ABM de cuarto orden, comúnmente

llamado ABM4:

$$\text{Predictor:} \quad y_{n+1}^{(0)} = y_n + \frac{h}{24}\left[55f_n - 59f_{n-1} + 37f_{n-2} - 9f_{n-3}\right], \quad n = 3,4,\ldots,N-1$$

$$\text{Corrector:} \quad y_{n+1}^{(k)} = y_n + \frac{h}{24}\left[9f(t_{n+1}, y_{n+1}^{(k)}) + 19f_n - 5f_{n-1} + f_{n-2}\right], \quad k = 1,2,\ldots$$

Note que el método ABM4 requiere conocer los valores y_1, y_2, y_3. Para calcular estos valores se usa normalmente el método RK4. En el Programa 7.9 se muestra la implementación del método predictor-corrector de Milne. Aquí usamos la función `rk4` dada en el Programa 7.5.

Programa 7.9. Método ABM4

```
def abm4(t0,y0,f,h,tf,tol=1.0e-6):
    N = int((tf-t0)/h)
    t = np.linspace(t0,tf,num=N+1)
    y = np.empty(N+1)
    t_aux, y_aux = rk4(t0,y0,f,h,t0+3*h)
    t[0:4], y[0:4] = t_aux, y_aux
    fn = [f(t[0],y[0]),f(t[1],y[1]),f(t[2],y[2])]
    h4 = h/24
    for n in range(3,N):
        fn.append(f(t[n],y[n]))
        yn1_k = y[n] + h4*(55*fn[n]-59*fn[n-1]+37*fn[n-2]-9*fn[n-3])
        while True:
            fn1_k = f(t[n+1],yn1_k)
            w = y[n] + h4*(9*fn1_k+19*fn[n]-5*fn[n-1]+fn[n-2])
            if np.abs(w-yn1_k)/np.abs(yn1_k) < tol:
                break
            yn1_k = w
        y[n+1] = w
    return t,y
```

Ejemplo 7.5. Comparemos los métodos predictor-corrector: Milne, Heun y ABM4, en la resolución del problema de valor inicial del Ejemplo 7.4. Escojamos $h = 0.1$ como longitud de paso y `tol` $= 10^{-6}$ como tolerancia. En la Tabla 7.3 se proporciona un resumen de todos los cálculos, donde se ve fácilmente que el % de error relativo global para el método de Milne es significativamente más bajo que todos los métodos utilizados. Como se esperaba, comenzando con el método de predictor-corrector de Heun, que es de hecho un método de orden 2, la precisión mejora con el orden del método de Milne. Cabe destacar que los valores y_1, y_2, y_3 de los métodos de Milne y ABM4 son los mismos, ya que fueron generados usando el método RK4. El comportamiento del error relativo local de los métodos se puede apreciar gráficamente en la Figura 7.3.

Tabla 7.3. Resumen de las cálculos del Ejemplo 7.4 para los métodos predictor-corrector: Milne, Heun y ABM4

	PC-Heun		Milne		ABM4	
t_n	$y_n^{PC-Heun}$	$e_{PC-Heun}$	y_n^{Milne}	e_{Milne}	y_n^{ABM4}	e_{ABM4}
0.0	1.00000000	0.00000000	1.00000000	0.00000000	1.00000000	0.00000000
0.1	1.00150075	0.00050008	1.00100016	0.00000001	1.00100016	0.00000001
0.2	1.00901954	0.00100085	1.00801066	0.00000002	1.00801066	0.00000002
0.3	1.02866658	0.00150392	1.02712183	0.00000003	1.02712183	0.00000003
0.4	1.06683021	0.00201248	1.06469182	0.00000401	1.06469662	0.00000852
0.5	1.13049606	0.00253214	1.12764701	0.00000558	1.12766344	0.00002015
0.6	1.22772633	0.00307222	1.22397950	0.00001100	1.22400874	0.00003489
0.7	1.36834997	0.00364731	1.36339724	0.00001461	1.36344959	0.00005301
0.8	1.56495301	0.00427845	1.55832003	0.00002187	1.55840299	0.00007511
0.9	1.83432152	0.00499431	1.82525799	0.00002855	1.82539249	0.00010224
1.0	2.19959108	0.00583208	2.18692278	0.00003910	2.18713496	0.00013613

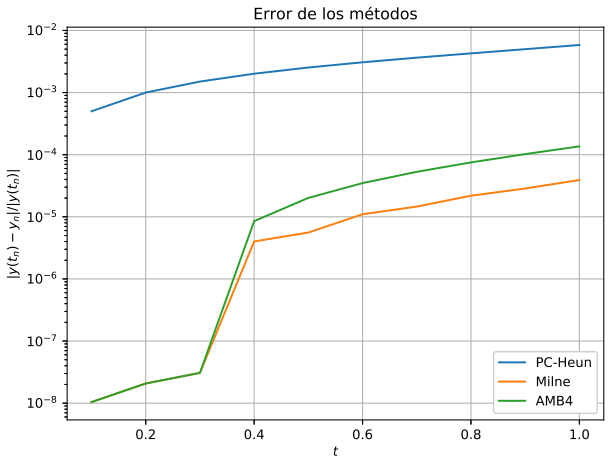


Figura 7.4. Tasa de convergencia de los métodos predictor-corrector: Milne, Heun y ABM4 para el problema de valor inicial del Ejemplo 7.4.

7.5 Convergencia, Consistencia y Estabilidad

Hasta ahora, nos hemos concentrado en los aspectos constructivos de los métodos. Desafortunadamente, saber construir y aplicar un método no garantiza éxito, por lo que ahora debemos examinar en qué medida los métodos son útiles en la práctica. En esa dirección, lo primero que debemos revisar es el orden del error del método, de lo cual en las secciones anteriores hemos discutido algo al respecto. Para complementar lo dado, estudiaremos la convergencia, consistencia y estabilidad de los métodos.

Asumamos que el problema de valor inicial (7.2) tiene solución única, es decir, cuando la función $f(t,y)$ satisface las hipótesis de los Teoremas 7.2 o 7.3; específicamente, asumamos que la función $f(t,y)$ es continua y satisface la condición de Lipschitz en la segunda variable. Denotemos con $y_h(t)$ la solución aproximada del problema de valor inicial (7.2), que se obtiene usando una longitud de paso h.

Definición 7.6. Sean $h > 0$ y $t \in \mathbb{R}$ fijo. Se dice que un método numérico para el problema de valor inicial (7.2) es *convergente*, si

$$\lim_{h\to 0} y_h(t) = y(t),$$

para todo $t \in [a,b]$, siempre y cuando los valores de arranque satisfacen la misma ecuación, a saber:

$$\lim_{h\to 0} y_h(t_0 + nh) = y_0, \quad (0 \leq n < N).$$

Si existe una constante $c > 0$ tal que

$$\max_{t\in[a,b]} |y(t) - y_h(t)| \leq c\,h^p,$$

para todo h, entonces p es el orden de convergencia del método.

Definición 7.7. Se dice que un método numérico para el problema de valor inicial (7.2) es *estable*, si pequeñas perturbaciones de la ecuación en diferencias o de las condiciones iniciales afectan poco a la solución aproximada $y_h(t)$. Asimismo, se dice que un método numérico es *consistente*, si el error local tiende a cero cuando la longitud de paso h tiende a cero.

Comencemos estudiando la convergencia, estabilidad y consistencia de los métodos de un paso. Los métodos de un paso que hemos descrito se pueden expresar en forma general como

$$y_{n+1} = y_n + h\varphi(t_n, y_n, h), \tag{7.31}$$

donde φ se denomina *función incremento*. Se asume que φ es continua y diferenciable. En el siguiente teorema se establece la relación entre convergencia y consistencia de un método de un paso.

Teorema 7.6. *Sea un método numérico de la forma* (7.31) *tal que la función de incremento* φ *satisface la condición de Lipschitz en la segunda variable. Entonces, el método numérico es convergente si, y sólo si es consistente.*

Se puede establecer que los métodos de un paso consistentes son estables. De esta forma, el Teorema 7.6 nos indica que la consistencia de un método de un paso es una condición necesaria y suficiente para la convergencia, pues estos métodos son siempre estables. Asimismo, de este teorema también se infiere que el orden de convergencia de un método de un paso está dado por el orden del error de truncamiento global. Así, por ejemplo, el orden de convergencia de Euler es $\mathbf{O}(h)$ o el de RK4 es $\mathbf{O}(h^4)$.

Pasemos ahora a estudiar la convergencia, consistencia y estabilidad de un método multipaso. En general el análisis de un método multipaso es más complicado. Asumamos que el método multipaso está definido por (7.26), además, consideremos los polinomios $p(z)$ y $q(z)$ definidos como:

$$p(z) = a_k z^k + a_{k-1} z^{k-1} + \cdots + a_0, \tag{7.32}$$

$$q(z) = b_k z^k + b_{k-1} z^{k-1} + \cdots + b_0. \tag{7.33}$$

Definición 7.8. El método multipaso (7.26) se dice *estable* si las raíces del polinomio p dado en (7.32) se encuentran en el disco $\{z \in \mathbb{C} : |z| \leq 1\}$ y si cada raíz de módulo 1 es simple. Además, el método se dice *consistente* si $p(1) = 0$ y $p'(1) = q(1)$, donde q es el polinomio definido en (7.33).

Para el estudio de los conceptos clásicos de estabilidad y consistencia y su conexión con los polinomios p y q, así como también la demostración del siguiente teorema, se recomienda al lector revisar el libro [17].

Teorema 7.7. *Para que el método multipaso* (7.26) *sea convergente es necesario y suficiente que sea estable y consistente.*

Ejemplo 7.6. A manera de ejemplo utilicemos el Teoream7.7 para demostrar que el método AB5 es convergente. Se tiene que los polinomios p y q asociados al método AB5 están dados por:

$$p(z) = z^5 - z^4 = z^4(z-1),$$
$$q(z) = \frac{1901}{720}z^4 - \frac{2774}{720}z^3 + \frac{2616}{720}z^2 - \frac{1274}{720}z + \frac{251}{720}.$$

Luego, las raíces de p son $z = 0$ y $z = 1$. Por lo tanto, el método AB5 es estable. Por otra parte, como $p'(z) = 5z^4 - 4z^3$, entonces

$$p'(1) = 1 = q(1) = \frac{1901 - 2774 + 2616 - 1274 + 251}{720}.$$

Así, el método AB5 también es consistente. Por el Teorema 7.6, el método AB5 es convergente.

La convergencia es una propiedad esencial para un algoritmo. Los resultados de convergencia, tales como el Teorema 7.6 o el Teorema 7.7 , aseguran que uno puede calcular la solución de un problema de valor inicial (7.2) para cualquier precisión deseada simplemente haciendo h lo suficientemente pequeño. Como estos teoremas se aplican en aritmética exacta, ellos no hacen referencia al error de redondeo. Aunque el método sea estable y convergente es posible que, para determinados valores de la longitud de paso, el error de truncamiento global sea demasiado grande para que la solución obtenida por el método sea aceptable, y se requiera tomar una longitud de paso más pequeña, originando con ello un aumento considerable en el costo computacional. Este hecho ocasiona que se deba introducir el concepto de *estabilidad absoluta*.

Intuitivamente, diremos que un método numérico es absolutamente estable si, para una longitud de paso h fija, y_n permanece acotado cuando $t_n \to +\infty$. Esta propiedad, por lo tanto, se ocupa del comportamiento asintótico de y_n, a diferencia de un método estable para el cual, para un intervalo de integración fijo, y_n permanece acotado cuando $h \to 0$. Para una definición precisa, consideremos el siguiente problema de valor inicial, denominado *problema de prueba*,

$$\begin{cases} y'(t) = \lambda y(t), & t > 0, \\ y(0) = 1, \end{cases} \tag{7.34}$$

con $\lambda \in \mathbb{C}$, cuya solución es $y(t) = e^{\lambda t}$. Note que $\lim_{t \to +\infty} |y(t)| = 0$ si $\mathrm{Re}(\lambda) < 0$.

Definición 7.9. Un método numérico para el problema inicial (7.34) es *absolutamente estable* si

$$|y_n| \to 0 \qquad \text{cuando} \qquad t_n \to +\infty. \tag{7.35}$$

Sea h la longitud de paso. La solución numérica y_n de (7.34) obviamente depende de h y λ. La *región de absoluta estabilidad* de un método numérico es el subconjunto de números complejos

$$\mathscr{A} = \{z = \lambda h \in \mathbb{C} : (7.35) \text{ se cumple}\}.$$

Por lo tanto, $\mathscr{A}$ es el conjunto de valores del producto λh para el cual el método numérico proporciona soluciones que decaen a cero cuando t_n tiende a infinito.

Veamos si los métodos de un paso presentados previamente son absolutamente estables.

1. *Método de Euler:* aplicando (7.11) al problema (7.34) y considerando que $f(t,y) = \lambda y$, obtenemos

$$\begin{aligned} y_{n+1} &= y_n + \lambda h y_n \\ &= (1+\lambda h) y_n, \quad \text{con } y_0 = 1. \end{aligned}$$

Procediendo recursivamente sobre n se tiene

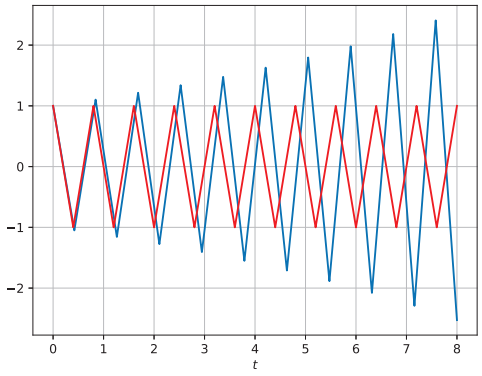
$$y_n = (1+\lambda h)^n, \quad n \geq 0.$$

De esta forma, la condición (7.35) se satisface si, y sólo si $|1+\lambda h| < 1$, esto es, si λh pertenece al círculo de radio uno y centro $(-1,0)$ (ver Figura 7.6). Esto equivale a requerir que

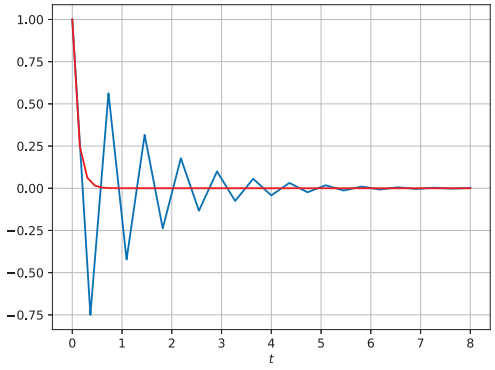
$$\lambda h \in \mathbb{C}^- \quad \text{y} \quad 0 < h < -\frac{2\text{Re}(\lambda)}{|\lambda|^2}, \tag{7.36}$$

donde $\mathbb{C}^- = \{z \in \mathbb{C} : \text{Re}(z) > 0\}$.

Ejemplo 7.7. Para el problema de valor inicial $y(t) = -5y(t)$ para $t > 0$ y $y(0) = 1$, la condición (7.36) implica que $0 < h < 2/5$. La Figura 7.5 muestra el comportamiento de la solución calculada por el método de Euler, para dos valores de h que no cumplen esta condición, mientras que a la derecha mostramos las soluciones para dos valores de h que sí lo hacen. Observe que en este segundo caso las oscilaciones también están presentes, pero se atenúan cuando t crece.



(a) Soluciones calculadas con $h = 0.41 > 2/5$ (azul) y $h = 2/5$ (rojo)



(b) Soluciones calculadas para $h = 0.35$ (azul) y la otra para $h = 0.15$ (rojo).

Figura 7.5. Comportamiento de la solución calculada por el método de Euler. Observe que en el caso $h = 2/5$ las oscilaciones permanecen sin modificaciones a medida que t crece.

2. *Método de Euler modificado:* aplicando (7.14) al problema (7.34) y procediendo como antes, tenemos

$$y_n = \left(1 + \lambda h + \frac{(\lambda h)^2}{2}\right)^n, \quad n \geq 0.$$

Así, la condición (7.35) se satisface si, y sólo si

$$\left|1 + \lambda h + \frac{(\lambda h)^2}{2}\right| < 1.$$

Como se muestra en la Figura 7.6, la región de estabilidad absoluta del método de Euler modificado es mayor que la del método de Euler. Sin embargo, su restricción al eje real es la misma.

3. *Método de Heun:* aplicando (7.13) al problema (7.34) y procediendo recursivamente sobre n, obtenemos:

$$y_n = \left(1 + \lambda h + \frac{(\lambda h)^2}{2}\right)^n, \quad n \geq 0.$$

Así, la condición (7.35) se satisface si, y sólo si

$$\left|1 + \lambda h + \frac{(\lambda h)^2}{2}\right| < 1,$$

es decir, los métodos de Heun y Euler modificado tienen la misma región de estabilidad (ver Figura 7.6).

4. *Método de Ralston:* aplicando (7.15) al problema (7.34) y procediendo recursivamente sobre n, obtenemos:

$$y_n = \left(1 + \frac{2(\lambda h)}{3} + \frac{(\lambda h)^2}{4}\right)^n, \quad n \geq 0.$$

Así, la condición (7.35) se satisface si, y sólo si

$$\left|1 + \frac{2(\lambda h)}{3} + \frac{(\lambda h)^2}{4}\right| < 1,$$

Como se muestra en la Figura 7.6, la región de estabilidad absoluta del método de Ralston es mayor que la del método de Heun.

5. *Método RK4:* aplicando (7.5) al problema (7.34) y procediendo recursivamente sobre n, obtenemos:

$$y_n = \left(1 + (\lambda h) + \frac{(\lambda h)^2}{2} + \frac{(\lambda h)^3}{6} + \frac{(\lambda h)^4}{24}\right)^n, \quad n \geq 0.$$

Así, la condición (7.35) se satisface si, y sólo si

$$\left|1 + (\lambda h) + \frac{(\lambda h)^2}{2} + \frac{(\lambda h)^3}{6} + \frac{(\lambda h)^4}{24}\right| < 1,$$

En la Figura 7.6 se muestra la región de estabilidad absoluta del método RK4.

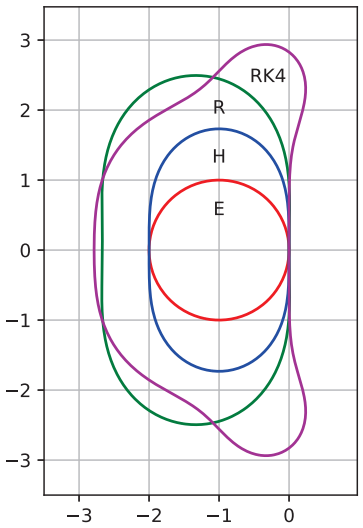


Figura 7.6. Regiones de estabilidad absoluta para los métodos Euler (E), Heun (H), Ralston (R) y para el método RK4.

Definición 7.10. Diremos que un método es *A*-estable si, para $\mathrm{Re}(\lambda) < 0$, la condición (7.35) se cumple para todo valor de h; esto es, $\mathscr{A} \cap \mathbb{C}^- = \mathbb{C}^-$. Diremos que un método es *condicionalmente estable* si $\mathscr{A} \cap \mathbb{C}^- \subset \mathbb{C}^-$.

Se aprecia que ninguno de los métodos de un paso descritos son *A*-estables, pero si son condicionalmente estables. Es de notar que en las aplicaciones es muy importante conocer la región de estabilidad del método empleado, ya que con ello se puede determinar la longitud de paso h que garantice la consistencia, estabilidad y convergencia. Más adelante veremos algunos casos. Para un estudio detallado de la estabilidad de los métodos numéricos para el problema de valor inicial (7.2) se recomienda al lector revisar los libros [11] y [25].

7.6 Sistemas de Ecuaciones Diferenciales

En la práctica, muchas de las aplicaciones de ecuaciones diferenciales ordinarias envuelven un sistema de ecuaciones diferenciales ordinarias de primer orden. Por ello, centraremos nuestra atención en resolver el siguiente sistema de ecuaciones diferenciales ordinarias de primer orden cuyas incógnitas son $y_1(t),\ldots,y_1(t)$:

$$\begin{cases} y_1'(t) = f_1(t,y_1,\ldots,y_m), \\ \vdots \\ y_m'(t) = f_m(t,y_1,\ldots,y_m), \end{cases} \tag{7.37}$$

donde $t \in [a,b]$, con condiciones iniciales

$$y_1(a) = y_{0,1},\ldots,y_m(a) = y_{0,m}.$$

Afortunadamente, uno puede adaptar los métodos numéricos previamente descritos para resolver un sistema de ecuaciones diferenciales ordinarias de primer orden. Simplemente aplicando el método a cada ecuación $y_j'(t) = f_j(t,y_1,\ldots,y_m)$, para $j = 1,\ldots,m$. Esto último es equivalente a aplicar el método a la forma vectorial del sistema. La forma vectorial del sistema (7.37) está dada por

$$\begin{cases} \mathbf{y}'(t) = \mathbf{f}(t,\mathbf{y}(t)), \quad t \in [a,b], \\ \mathbf{y}(a) = (y_{0,1},\ldots,y_{0,m}), \end{cases} \tag{7.38}$$

donde

$$\mathbf{y}(t) = (y_1(t),\ldots,y_m(t)), \qquad \mathbf{y}'(t) = \left(y_1'(t),\ldots,y_m'(t)\right)$$

y la función $\mathbf{f} : \mathbb{R}\times\mathbb{R}^m \to \mathbb{R}^m$ está definida como

$$\mathbf{f}(t,\mathbf{y}(t)) = (f_1(t,y_1,\ldots,y_m),\ldots,f_m(t,y_1,\ldots,y_m)).$$

Así, la forma vectorial del método de Euler está dada por:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{f}(t,\mathbf{y}(t)), \quad n \geq 0$$

con $\mathbf{y}_0 = \mathbf{y}(t_0)$. La extensión de los otros métodos desarrollados previamente para el caso de una sola ecuación diferencial al caso vectorial es sencilla. Por ejemplo, la forma vectorial del método RK4 es

$$\begin{aligned} \mathbf{y}_{n+1} &= \mathbf{y}_n + \frac{h}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_4 + \mathbf{k}_4), \qquad n = 0,1,\ldots,N-1, \\ \mathbf{k}_1 &= \mathbf{f}(t_n,\mathbf{y}_n), \\ \mathbf{k}_2 &= \mathbf{f}\left(t_n + \frac{1}{2}h, \mathbf{y}_n + \frac{1}{2}h\mathbf{k}_1\right), \\ \mathbf{k}_3 &= \mathbf{f}\left(t_n + \frac{1}{2}h, \mathbf{y}_n + \frac{1}{2}h\mathbf{k}_2\right), \\ \mathbf{k}_4 &= \mathbf{f}\left(t_n + h, \mathbf{y}_n + h\mathbf{k}_3\right), \end{aligned}$$

con $\mathbf{y}_0 = \mathbf{y}(t_0)$. En los Programas 7.10 y 7.11 se muestra la implementación de la forma vectorial de los métodos de Euler y RK4, respectivamente.

Programa 7.10. Forma vectorial del método de Euler

```
def euler_v(t0,y0,f,h,tf):
    N = int((tf-t0)/h)
    t = np.linspace(t0,tf,num=N+1)
    y0 = np.asarray(y0,dtype=float)
    y = []
    y.append(y0)
    for n in range(N):
        yn1 = y[n] + h*f(t[n],y[n])
        y.append(yn1)
    return t,np.array(y)
```

Programa 7.11. Forma vectorial del método RK4

```
def rk4_v(t0,y0,f,h,tf):
    N = int((tf-t0)/h)
    t = np.linspace(t0,tf,num=N+1)
    y0 = np.asarray(y0,dtype=float)
    y = []
    y.append(y0)
    for n in range(N):
        k1 = h*f(t[n],y[n])
        k2 = h*f(t[n]+h/2,y[n]+k1/2)
        k3 = h*f(t[n]+h/2,y[n]+k2/2)
        k4 = h*f(t[n]+h,y[n]+k3)
        yn1 = y[n] + (k1+2*k2+2*k3+k4)/6
        y.append(yn1)
    return t,np.array(y)
```

Por otra parte, el teorema de existencia y unicidad de la solución del sistema (7.38) se obtiene extendiendo el Teorema 7.2. Tal extensión se muestra a continuación.

Teorema 7.8. *Si* $\mathbf{f}(t,\mathbf{y})$ *y todas sus derivadas parciales de primer orden con respecto a todos los argumentos son continuas en la región* $R \subset \mathbb{R}\times\mathbb{R}^m$*, definida por*

$$R = \{(x,\mathbf{y}) \in \mathbb{R}\times\mathbb{R}^m : |t-t_0| \leq \alpha, \quad \|\mathbf{y}-\mathbf{y}_0\| \leq \beta\},$$

entonces el sistema de ecuaciones diferenciales (7.38) *tiene una solución única.*

Ejemplo 7.8. Apliquemos la forma vectorial del método RK4 para resolver las *ecuaciones de Lotka-Volterra* o también conocidas *ecuaciones de depredador-presa*,

$$\begin{cases} y_1'(t) = \alpha y_1(t) - \beta y_1(t) y_2(t), \\ y_2'(t) = -\gamma y_2(t) + \delta y_1(t) y_2(t), \end{cases}$$

para $t \in [0,T]$ y condiciones iniciales $y_1(0) = y_{0,1}, y_2(0) = y_{0,2}$. Las funciones y_1 y y_2 representan el número de presas y depredadores, respectivamente. Los parámetros α, β, δ y γ son números reales positivos que describen la interacción de las dos especies. Para el ejemplo numérico, consideremos los valores $\alpha = \beta = \delta = \gamma = 1$ y el intervalo de integración $[0,10]$. Con el propósito de usar el Programa 7.11 para un sistema de ecuaciones diferenciales ordinarias, debemos crear una función

Se deja como ejercicio para el lector, probar la existencia y unicidad de la solución de las ecuaciones de Lotka-Volterra.

`f` que contenga las componentes de la función vectorial $\mathbf{f}(t,\mathbf{y}(t))$. En el caso de las ecuaciones de Lotka-Volterra, primero creamos la función `f_LV`,

```
def f_LV(t,y,args):
    alfa,beta,delta,gamma = args
    return np.array([alfa*y[0]-beta*y[0]*y[1],-gamma*y[1]+delta*y[0]*y[1]])
```

donde `args` es una tupla que contiene los valores de los parámetros α,β,δ y γ. Seguidamente cargamos los valores de los parámetros

```
>>> args = 1., 1., 1., 1.
```

Por último, creamos la función `f` de forma anónima,

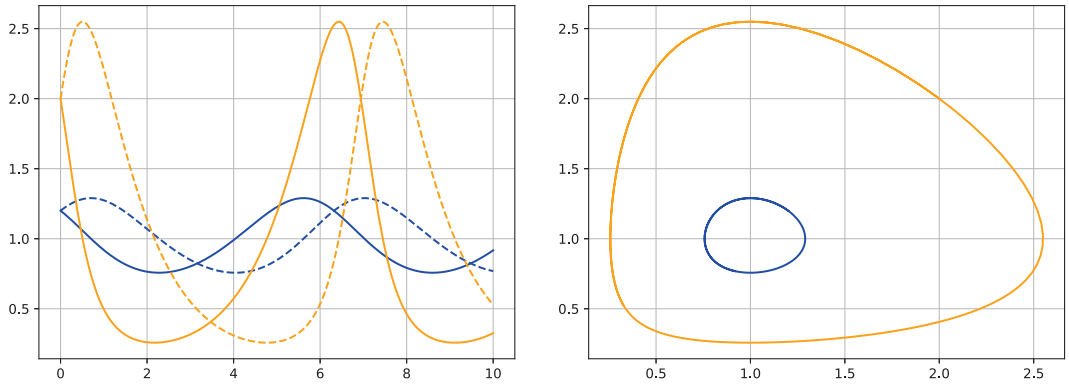
```
>>> f = lambda t,y: f_LV(t,y,args)
```

Ahora ejecutamos el Programa 7.11 con las siguientes instrucciones.

```
>>> tn,yn1 = rk4_v(0,[1.2,1.2],f,0.05,10)
>>> tn,yn2 = rk4_v(0,[2.,2.],f,0.05,10)
```

Aquí, `yn1` y `yn2` son arreglos bidimensionales que contienen las aproximaciones de y_1 y y_2, en cada nodo t_n con una longitud de paso $h = 0.05$, para las condiciones iniciales $y_1(0) = y_2(0) = 1.2$ y $y_1(0) = y_2(0) = 2$, respectivamente.

El gráfico de la Figura 7.7, a la izquierda, representa la evolución temporal de las dos componentes de la solución. El segundo gráfico de la Figura 7.7, a la derecha, muestra la trayectoria que emana del valor inicial en el llamado *plano de fase*, es decir, el plano cartesiano cuyos ejes de coordenadas son y_1 e y_2.



(a) Representación de y_1 e y_2, la línea continua se refiere a y_1, la línea punteada a y_2.

(b) Trayectorias correspondientes en el plano de fase.

Figura 7.7. Soluciones numéricas del sistema de Lotka-Volterra en el intervalo de tiempo $[0,10]$. Se consideran dos condiciones iniciales diferentes: $y_1(0) = y_2(0) = 1.2$ (líneas azules) y $y_1(0) = y_2(0) = 2$ (líneas naranjas).

Simulación de la Salida de un Sistema Dinámico Lineal

Una aplicación importante de la resolución numérica de sistemas de ecuaciones diferenciales ordinarias, es la simulación de la salida de un sistema dinámico lineal. Consideremos un sistema lineal e invariante en tiempo descrito por las ecuaciones de estado:

$$\begin{aligned} \mathbf{x}'(t) &= A\mathbf{x}(t) + B\mathbf{u}(t) \\ \mathbf{y}(t) &= C\mathbf{x}(t) + D\mathbf{u}(t), \end{aligned} \tag{7.39}$$

donde $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{u} \in \mathbb{R}^m$ y $\mathbf{y} \in \mathbb{R}^p$ son los vectores de estados, entradas y salidas, respectivamente, $B \in \mathbb{R}^{n\times m}$, $C \in \mathbb{R}^{p\times n}$, $D \in \mathbb{R}^{p\times m}$ y $A = (a_{ij})$ es una matriz real de orden n estable, esto es, los autovalores de A poseen parte real negativa.

Se requiere aproximar el vector de salidas $\mathbf{y}(t)$ en el intervalo $[t_0, T_{max}]$, con un vector de entradas $\mathbf{u}(t)$ dado y

$$\mathbf{x}(t_0) = (x_1(t_0), x_2(t_0), \cdots, x_n(t_0)) \tag{7.40}$$

el vector de estados inicial del sistema. Para tal requerimiento, se pueden utilizar la versión vectorial de los métodos numéricos para ecuaciones diferenciales ordinarias, en la resolución del sistema de ecuaciones diferenciales ordinarias dado por

$$\begin{cases} \mathbf{x}'(t) = \mathbf{f}(t, \mathbf{x}(t)), \quad t \in [t_0, T_{max}], \\ \mathbf{x}(t_0) = (x_1(t_0), x_2(t_0), \cdots, x_n(t_0)), \end{cases} \tag{7.41}$$

donde

$$\mathbf{f}(t, \mathbf{x}(t)) = A\mathbf{x}(t) + B\mathbf{u}(t).$$

Ahora bien, las formas vectoriales de los métodos numéricos aplicados al sistema de ecuaciones diferenciales ordinarias (7.41) generan respectivamente aproximaciones, $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_N \in \mathbb{R}^n$, de los vectores de estado $\mathbf{x}(t_0), \mathbf{x}(t_1), \ldots, \mathbf{x}(t_N)$ en cada uno de los nodos $t_0, t_1, \ldots, t_N$. De esta forma, las aproximaciones del vector de salidas $\mathbf{y}(t)$ se obtienen como:

$$\mathbf{y}_k = C\mathbf{x}_k + D\mathbf{u}(t_k), \qquad k = 0, 1, \ldots, N,$$

donde $\mathbf{x}_0 = \mathbf{x}(t_0)$ y $\mathbf{x}(t_k) \approx \mathbf{x}_k$, para $k = 1, 2, \ldots, N$.

Ejemplo 7.9 (Dinámica de reservorios de agua [4])**.** Consideremos el sistema de reservorios de agua que se muestra en la Figura 7.8, donde $r_1(t)$, $r_2(t)$ son las entradas de lluvia, $x_1(t)$, $x_2(t)$, $x_3(t)$ son los estados de almacenamiento de superficie en las ubicaciones 1-3, respectivamente, mientras que el estado de almacenamiento de agua subterránea (incluida la infiltración) es $x_4(t)$. La constante k es el flujo de agua superficial, mientras que las constantes l_1 y l_2 son el flujo de infiltración. La expresión $l_3(x_4 - x_3)$ significa el intercambio entre la corriente y el agua subterránea. Las salidas y_1, y_2 son el flujo de salida y la contribución del agua subterránea al flujo de la corriente, respectivamente; y las cantidades $g_{11}u_1$ y $g_{22}u_2$ denotan la liberación de agua.

Las ecuaciones de continuidad para este sistema son:

$$\begin{aligned} x_1'(t) &= -l_1 x_1(t) + r_1(t) - g_{11}u_1(t), \\ x_2'(t) &= -l_2 x_2(t) + r_2(t) - g_{22}u_2(t), \\ x_3'(t) &= l_3(x_4(t) - x_3(t)) - k x_3(t) + g_{11}u_1(t) + g_{22}u_2(t), \\ x_4'(t) &= l_1 x_1(t) + l_2 x_2(t) - l_3(x_4(t) - x_3(t)). \end{aligned}$$

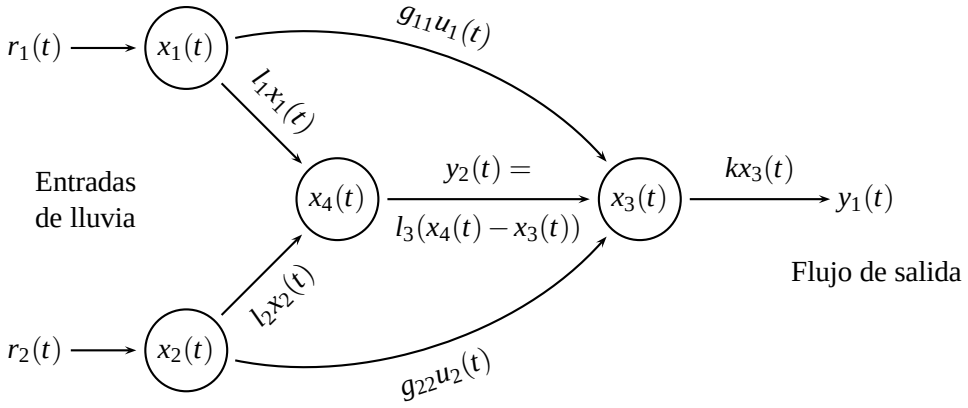


Figura 7.8. Red de reservorios de agua

Las salidas son:

$$\begin{aligned} y_1(t) &= kx_3(t), \\ y_2(t) &= l_3(x_4(t) - x_3(t)). \end{aligned}$$

En forma vectorial, las ecuaciones de estado son:

$$\begin{aligned} \mathbf{x}'(t) &= A\mathbf{x}(t) + B\mathbf{u}(t) + \mathbf{r}(t) \\ \mathbf{y}(t) &= C\mathbf{x}(t), \end{aligned}$$

donde

$$A = \begin{bmatrix} -l_1 & 0 & 0 & 0 \\ 0 & -l_2 & 0 & 0 \\ 0 & 0 & -(k+l_3)) & l_3 \\ l_1 & l_2 & l_3 & -l_3 \end{bmatrix}, \quad B = \begin{bmatrix} -g_{11} & 0 \\ 0 & -g_{22} \\ g_{11} & g_{22} \\ 0 & 0 \end{bmatrix}, \quad \mathbf{r}(t) = \begin{bmatrix} r_1(t) \\ r_2(t) \\ 0 \\ 0 \end{bmatrix},$$

$$C = \begin{bmatrix} 0 & 0 & k & 0 \\ 0 & 0 & -l_3 & l_3 \end{bmatrix}.$$

Para aproximar el vector de salidas $\mathbf{y}(t) = (y_1(t), y_2(t))$ en el intervalo $[0, T_{max}]$, se resuelve numéricamente el sistema de ecuaciones diferenciales ordinarias:

$$\begin{cases} \mathbf{x}'(t) = A\mathbf{x}(t) + B\mathbf{u}(t) + \mathbf{r}(t), \quad t \in [0, T_{max}], \\ \mathbf{x}(0) = (x_1(0), x_2(0), \cdots, x_n(0)). \end{cases} \tag{7.42}$$

Simulemos las salidas del sistema resolviendo el problema (7.42), empleando la forma vectorial del método RK4 en el intervalo $[0, 60]$, con longitud de paso $h = 0.05$. Supongamos, en unidades físicas consistentes, que los valores de los parámetros del sistema son: $l_1 = l_2 = 0.1$, $l_3 = 0.3$, $k = 1$, $g_{11} = 0.05$ y $g_{22} = 0.15$. Las funciones de entrada de lluvias $r_1(t)$ y $r_2(t)$ están dadas por:

$$\begin{aligned} r_1(t) &= e^{-\frac{(t-10)^2}{60}} + e^{-\frac{(t-20)^2}{60}}, \\ r_2(t) &= e^{-\frac{(t-15)^2}{60}} + e^{-\frac{(t-30)^2}{60}} + e^{-\frac{(t-45)^2}{60}}. \end{aligned}$$

Las funciones de liberación de agua $u_1(t)$ y $u_2(t)$ están dadas por:

$$u_1(t) = u_2(t) = s(t) + s(t-10) + s(t-20) + s(t-30) + s(t-40) + s(t-50),$$

donde

$$s(t) = \begin{cases} 0, & t < 5, \\ 1, & 5 \leq t \leq 10, \\ 0, & t > 10. \end{cases}$$

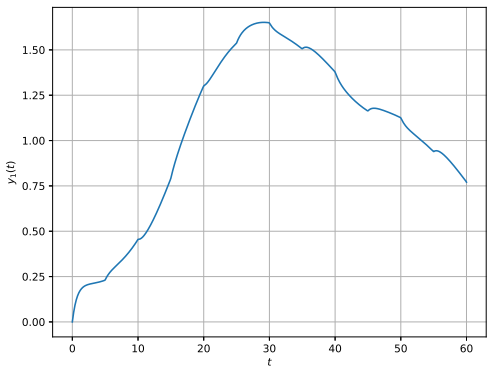
Tomamos $\mathbf{x}_0 = (1,1,0,1)$ como el vector de estados inicial del sistema.

En las Figuras 7.9 y 7.13 se muestran la evolución temporal de las salidas $y_1(t)$ e $y_2(t)$, y de los estados de almacenamiento $x_i(t)$, para $i = 1,2,3,4$, respectivamente. Las gráficas de las Figuras 7.9 y 7.13 se generaron ejecutando el Script que se muestra en el siguiente programa.

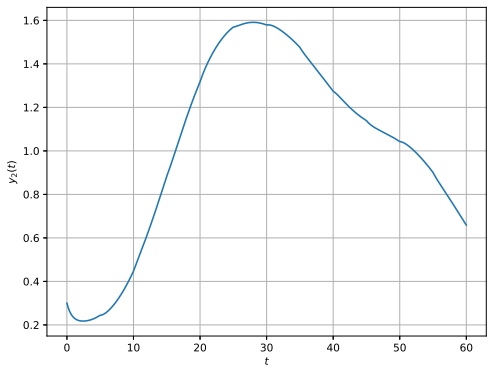
Programa 7.12. Script de la dinámica de reservorios de agua

```
def r(t):
    def r1(t):
        return np.exp(-(t-10)**2/60) + np.exp(-(t-20)**2/60)
    def r2(t):
        return np.exp(-(t-15)**2/60) + np.exp(-(t-30)**2/60) \
               + np.exp(-(t-45)**2/60)
    return([r1(t),r2(t),0,0])

def u(t):
    def s(t):
        return np.piecewise(t,[t<5,(t>=5)&(t<=10),t>10],[0,1,0])
    def u1(t):
        return s(t) + s(t-10) + s(t-20) + s(t-30) \
               + s(t-40) + s(t-50)
    return ([u1(t),u1(t)])

def y(x,k,l3):
    C = np.array([[0, 0, k, 0], [0, 0, -l3, l3]])
    m = x.shape[0]
    yn = np.zeros((m, 2))
    for i in range(m):
        yn[i] = C.dot(x[i])
    return yn

def f_RA(t, x, u, r, args):
    l1,l2,l3,k,g11,g22 = args
    A = np.array([[-l1,0,0,0],[0,-l2,0,0],[0,0,-(k+l3),l3],
                  [l1,l2,l3,-l3]])
    B = np.array([[-g11,0],[0,-g22],[g11,g22],[0,0]])
    return A.dot(x) + B.dot(u(t)) + r(t)

# Parámetros del sistema
k   = 1
l1  = .1
l2  = .1
l3  = 0.3
g11 = 0.05
g22 = 0.015

# Definición de la función f(t,y)
args = l1,l2,l3,k,g11,g22
```

```
f = lambda t,x: f_RA(t, x, u, r, args)

# Condición inicial
x0 = np.array([1.,1.,0,1.])

# Cálculo de las aproximaciones
tn,xn = cn.rk4_v(0,x0,f,0.1,60)
yn = y(xn, k, l3)

# Generación de gráficas
plt.figure(1)
plt.plot(tn,yn[:,0])
plt.grid()
plt.xlabel('$t$')
plt.ylabel('$y_{1}(t)$')
plt.figure(2)
plt.plot(tn,yn[:,1])
plt.grid()
plt.xlabel('$t$')
plt.ylabel('$y_{2}(t)$')
for i in range(4):
    plt.figure(i+3)
    plt.plot(tn,xn[:,i])
    plt.grid()
    plt.xlabel('$t$')
    texto = '$x_{' + str(i+1) + '}(t)$'
    plt.ylabel(texto)
plt.show()
```

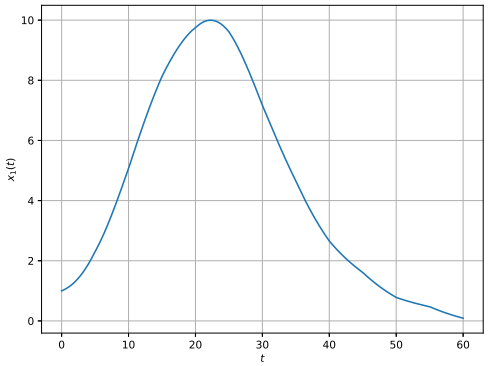


(a) Salida $y_1(t)$.

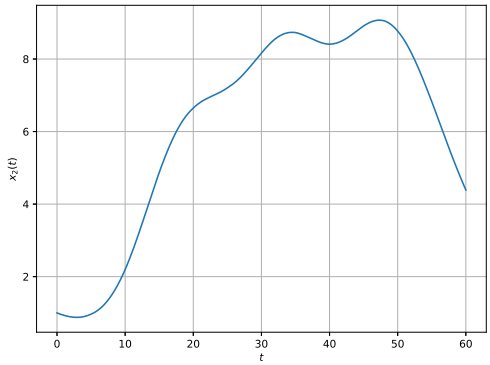


(b) Salida $y_2(t)$.

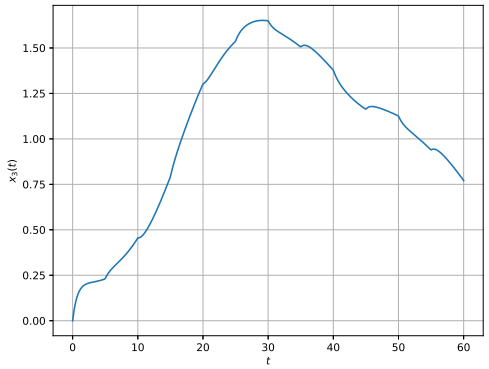
Figura 7.9. Salidas $y_1(t)$ e $y_2(t)$.



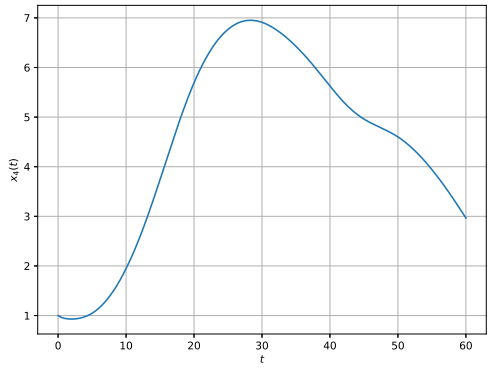
(a) Estado $x_1(t)$.



(b) Estado $x_2(t)$.



(c) Estado $x_3(t)$.



(d) Estado $x_4(t)$.

Figura 7.10. Salidas $y_1(t)$ e $y_2(t)$ y los estados de almacenamiento $x_1(t)$, $x_2(t)$, $x_3(t)$ y $x_4(t)$.

7.7 Ecuaciones de Orden Superior

Ahora consideremos el caso de una ecuación diferencial ordinaria de orden m,

$$y^{(m)}(t) = f(t, y', y'', \ldots, y^{(m-1)}), \tag{7.43}$$

para $t \in [a,b]$ y condiciones iniciales

$$y(a) = y_0, y'(a) = y_1, \ldots, y^{(m-1)}(a) = y_{m-1}.$$

Definiendo las funciones

$$w_1(t) = y(t), w_2(t) = y'(t), \ldots, w_m(t) = y^{(m)}(t),$$

la ecuación (7.43) puede transformarse en un sistema de m ecuaciones diferenciales de primer orden

$$\begin{cases} w_1'(t) = w_2(t), \\ w_2'(t) = w_3(t), \\ \vdots \\ w_m(t) = f_m(t, w_1, \ldots, w_m), \end{cases} \tag{7.44}$$

con condiciones iniciales

$$w_1(a) = y_0, w_2(a) = y_1, \ldots, w_m(a) = y_{m-1}.$$

Por lo tanto, siempre podemos aproximar la solución de una ecuación diferencial de orden $m > 1$ recurriendo a un sistema equivalente de m ecuaciones de primer orden, y luego aplicando a este sistema un método numérico conveniente.

La existencia y unicidad de la solución de una ecuación diferencial ordinaria de orden m dada por (7.43), se obtiene aplicando el Teorema 7.8 al sistema de ecuaciones diferenciales de primer orden (7.44).

Ejemplo 7.10. Considere el circuito eléctrico de la Figura 7.11.

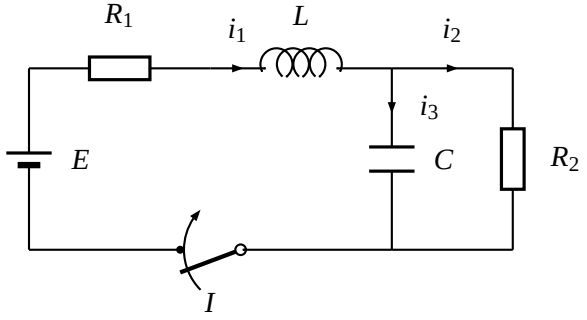


Figura 7.11. Circuito eléctrico

Queremos calcular la función $v(t)$ que representa la caída de potencial en los extremos del condensador C a partir del tiempo inicial $t = 0$ en el cual el interruptor I se ha apagado. Supongamos que la inductancia L puede expresarse como una función explícita de la intensidad de la corriente i, es decir $L = L(i)$. La ley de Ohm produce

$$E - \frac{d}{dt}(i_1 L(i_1)) = R_1 i_1 + v,$$

donde R_1 es una resistencia. Suponiendo que los flujos actuales se dirigen como se indica en la Figura 7.11, al diferenciar con respecto a t en ambos lados de la ley de Kirchoff $i_1 = i_2 + i_3$ y observando que $i_3 = Cdv/dt$ e $i_2 = v/R_2$, encontramos la ecuación

$$\frac{di_1}{dt} = C\frac{d^2v}{dt^2} + \frac{1}{R_2}\frac{dv}{dt}.$$

Por lo tanto, hemos encontrado un sistema de dos ecuaciones diferenciales cuya solución permite la descripción de la variación de tiempo de las dos incógnitas $i_1(t)$ y $v(t)$. La segunda ecuación tiene orden dos. Para el ejemplo numérico supongamos que $L = L(i)$ es constante y que $R_1 = R_2 = R$. Por lo tanto, combinando las dos ecuaciones anteriores llegamos a la siguiente ecuación diferencial de segundo orden:

$$LCv''(t) + \left(\frac{L}{R} + RC\right)v'(t) + 2v(t) = E,$$

con condiciones iniciales $v(0) = v'(0) = 0$. Tomando $w(t) = v'(t)$, de la ecuación de segundo orden obtenemos el sistema de ecuaciones diferenciales de primer orden

$$\begin{cases} v'(t) = w(t), \\ w'(t) = -\dfrac{1}{LC}\left(\dfrac{L}{R} + RC\right)w(t) - \dfrac{2}{LC}v(t) + \dfrac{E}{LC}, \end{cases} \tag{7.45}$$

con condiciones iniciales $v(0) = w(0) = 0$. Supongamos, en unidades físicas consistentes, que los valores de los parámetros físicos son: $L = 0.1$, $C = 0.003$, $R = 10$ y $E = 5$. En este caso, la función $\mathbf{f}(t, v, w)$ del sistema (7.45) es

$$\mathbf{f}(t, v, w) = \left(w, -\frac{1}{LC}\left(\frac{L}{R} + RC\right) w - \frac{2}{LC} v + \frac{E}{LC} \right)^T,$$

cuya implementación es:

```
def f_CE(t,v,w,args):
    L,C,R,E = args
    return np.array([w,(E-(L/R+R*C)*w-2.*v)/(L*C)])
```

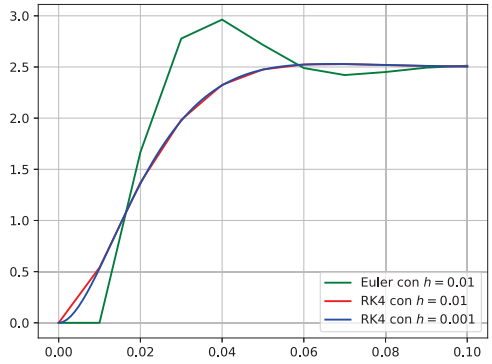
Seguidamente cargamos los parámetros e implementamos la función $\mathbf{f}(t, \mathbf{y})$.

```
>>> args = .1, .003, 10., 5.
>>> f = lambda t,y: f_CE(t,y[0],y[1],args)
```

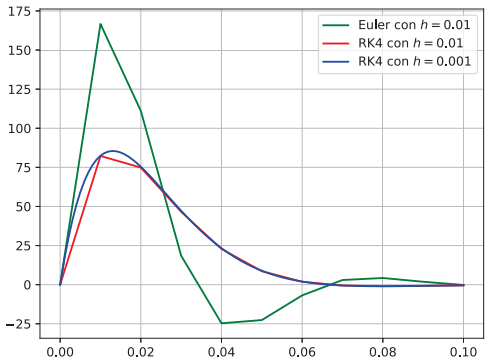
Aquí $y_1(t) = v(t)$ y $y_2(t) = w(t)$. Ahora aplicamos los métodos de Euler y RK4 con $h = 0.01$ en el intervalo $[0, 0.1]$, mediante los Programas 7.10 y 7.11, respectivamente:

```
>>> tn,ye = euler_v(0,[0,0],f,0.01,0.1)
>>> tn,yrk = rk4_v(0,[0,0],f,0.01,0.1)
```

En la Figura 7.12 reportamos los valores aproximados de $v(t)$ y $w(t)$. Como se esperaba, $v(t)$ tiende a $E/2 = 2.5$, para t grande.



(a) La caída de potencial $v(t)$.



(b) Derivada de $v(t)$, es decir, $w(t)$.

Figura 7.12. Soluciones numéricas de la ecuación (7.45). La línea verde representa la solución obtenida por el método de Euler con para $h = 0.01$ y las líneas roja y azul representan las soluciones halladas por el método RK4 con $h = 0.01$ y $h = 0.001$, respectivamente.

7.8 Problemas de Valores en la Frontera

Un *problema de valores en la frontera* es una ecuación diferencial de orden m con algunos de los valores de la variable dependiente $y(t)$ y su derivada especificados en el intervalo $[a,b]$. Esto expresado matemáticamente es

$$y^{(m)}(t) = f(t,y',y'',\ldots,y^{(m-1)}),$$

para $t \in [a,b]$ y condiciones de frontera

$$y(t_1) = y_{0,1}, y'(t_2) = y_{1,2} \ldots, y^{(m-1)}(t_m) = y_{m-1,m}.$$

Nos centraremos en los llamados *problemas de valores en la frontera en dos puntos,*

$$\begin{cases} y''(t) = f(t,y(t),y'(t)), \\ y(a) = \alpha, \qquad y(b) = \beta, \end{cases} \tag{7.46}$$

donde $f : \mathbb{R}^3 \to \mathbb{R}$. Para problemas de valores en la frontera, las preguntas de existencia y unicidad de una solución son más difíciles y requieren más hipótesis que las exigidas en problemas de valor inicial. El próximo teorema (ver Keller [15]) se dan las condiciones que garantizan una solución única; estas condiciones son suficientes, pero no son necesarias. Aún podría existir una solución, aunque estas condiciones no se satisfagan, pero esto no lo podemos garantizar a priori. Esta falta de seguridad de saber si un problema tiene o no una solución a menudo crea serias dificultades prácticas, un problema que afecta a muchas ecuaciones no lineales.

Teorema 7.9. *Supongamos que:*

a) $f(t,u,v)$ *es continua para todo* $u,v \in \mathbb{R}$ *y* $t \in [a,b]$*;*

b) $f(t,u,v)$ *satisface la condición de Lipschitz en las variables u y v, esto es, existe una constante* $\ell > 0$ *tal que*

$$|f(t,u_1,v) - f(t,u_2,v)| \leq \ell |u_1 - u_2|,$$
$$|f(t,u,v_1) - f(t,u,v_2)| \leq \ell |v_1 - v_2|,$$

para todo $u,v \in \mathbb{R}$ *y* $t \in [a,b]$*;*

c) $\dfrac{\partial f}{\partial u}(t,u,v) > 0$ *para todo* $u,v \in \mathbb{R}$ *y* $t \in (a,b)$*;*

d) $\left|\dfrac{\partial f}{\partial v}(t,u,v)\right|$ *es acotada para todo* $u,v \in \mathbb{R}$ *y* $t \in (a,b)$.

Entonces, el problema de valores en la frontera en dos puntos (7.46) *tiene solución única.*

Note que la solución numérica paso a paso de (7.46) usando los métodos numéricos dados en la Sección 7.6 requiere de dos condiciones iniciales, pero en (7.46) solo se presenta una condición inicial en $t = a$. Esto hace que los problemas de valores en la frontera en dos puntos sean difíciles de resolver. Existen distintos procedimientos para resolver problemas de valores en la frontera en dos puntos, de los cuales describiremos dos de ellos: *método de disparo* y *método de diferencias finitas.*

7.8.1 Método de Disparo

La idea del método de disparo *(shooting method)* es asumir el valor $y'(a)$, luego resolver repetitivamente la ecuación diferencial $y''(t) = f(t,y(t),y'(t))$ con condiciones iniciales $(y(a),y'(a))$, ajustando el valor de $y'(a)$ hasta que el valor final $y(b)$ coincida con el valor dado β con la precisión deseada. Es similar a ajustar el ángulo de disparo de un cañón para que el proyectil alcance el objetivo y es por eso que este método recibe el nombre de método de disparo.

En sí, el método de disparo se basa en la resolución del problema de valor inicial

$$\begin{cases} y''(t) = f(t,y(t),y'(t)), \\ y(a) = \alpha, \qquad y'(a) = z, \end{cases} \tag{7.47}$$

en el intervalo $[a,b]$, donde z es una aproximación del valor de $y'(a)$. De esta forma, el valor de la solución hallada y en b es una función de z, esto es, $\varphi(z) = y(b)$. El objetivo es ajustar z hasta que se encuentre un valor para el cual

$$\varphi(z) = \beta,$$

en otras palabras, resolver la ecuación no lineal

$$\varphi(z) - \beta = 0.$$

Cualquiera de los métodos dados en el Capítulo 4 se pueden emplear para hallar un cero de esta ecuación, pero note que la expresión matemática de la función $\varphi(z)$ no se conoce explícitamente, sino que el valor $\varphi(z)$ es igual a $y(b)$, donde y es la solución del problema (7.47). Por ello, el método más usado para hallar un cero de la ecuación $\varphi(z) - \beta = 0$ es el método de la Secante. Tenga presente que para utilizar el método de la Secante es necesario conocer dos aproximaciones iniciales de $y'(a)$, digamos z_1 y z_2. Con tales aproximaciones iniciales se construye z_3 a través de la fórmula

$$z_3 = z_2 + (\beta - \varphi(z_2)) \left(\frac{z_2 - z_1}{\varphi(z_2) - \varphi(z_2)} \right).$$

Repitiendo este procedimiento se genera la sucesión $\{z_n\}$,

$$z_{n+1} = z_n + (\beta - \varphi(z_n)) \left(\frac{z_n - z_{n-1}}{\varphi(z_n) - \varphi(z_{n-1})} \right), \qquad n \geq 2, \tag{7.48}$$

de manera que $\lim_{n\to\infty} \varphi(z_n) = \beta$. De esta forma, el método de disparo se puede resumir en el siguiente proceso iterativo.

Paso 1. Escoge dos aproximaciones iniciales de $y'(a)$, digamos z_1 y z_2; resuelve dos veces el problema de valor inicial (7.47) con z_1 y z_2; y calcula $\varphi(z_1)$ y $\varphi(z_2)$.

Paso 2. Halla el valor z_{n+1} para $n \geq 2$, a través de la fórmula (7.48) y calcula $\varphi(z_{n+1})$ resolviendo el problema de valor inicial (7.47).

Paso 3. Repite el Paso 2 hasta que se cumpla $|\varphi(z_{n+1}) - \beta| < \varepsilon$, donde $\varepsilon > 0$ es una tolerancia dada.

El lector debe estar preguntándose: ¿por qué no se usa el método de Newton en lugar de la Secante? La simple razón de ello es que Newton requiere el Jacobiano en cada iteración, pero éste no está disponible, además, en este caso su aproximación es muy costosa computacionalmente.

El método de disparo puede consumir mucho tiempo de computo, si cada solución del problema de valor inicial (7.47) requiere de un valor pequeño de la longitud de paso h. En consecuencia,

para disminuir el tiempo de computo, generalmente se utiliza un valor relativamente grande de h hasta que $|\varphi(z_{n+1}) - \beta|$ sea suficientemente pequeño y luego se reduce h para obtener la precisión requerida.

El problema de valor inicial (7.47) se puede resolver, por ejemplo, utilizando la forma vectorial del método RK4. En el Programa 7.13 se aprecia la implementación del método de disparo. Aquí empleamos la forma vectorial del método RK4 dada en el Programa 7.11.

Programa 7.13. Método de Disparo

```
def disparo(f,a,b,alfa,beta,h=0.1,tol=1.0e-6,maxit=1000):
    z1 = (beta-alfa)/(b-a)
    tn,yn = rk4_v(a,[alfa,z1],f,h,b)
    phi_z1 = yn[-1,0]
    er = beta - phi_z1
    k = 1
    if np.abs(er) < tol:
        return tn,yn,k
    z2 = z1 - 0.1*np.sign(er)
    k += 1
    while True:
        if k>maxit:
            print('Alcanzó el máximo de iteraciones')
            break
        if np.abs(er)<0.1:
            h = 0.001
        tn,yn = rk4_v(a,[alfa,z2],f,h,b)
        phi_z2 = yn[-1,0]
        er = beta-phi_z2
        if np.abs(er) < tol:
            print('Terminó satisfactoriamente')
            break
        z_aux = z2 + er*((z2-z1)/(phi_z2-phi_z1))
        z1,phi_z1 = z2,phi_z2
        z2 = z_aux
        k +=1
    return tn,yn,k
```

Ejemplo 7.11. Empleemos el método de disparo para aproximar la solución del siguiente problema de valores en la frontera en dos puntos.

$$\begin{cases} y''(t) = 2y^2(t) + 4ty(t)y'(t), \\ y(0) = \frac{1}{4}, \qquad y(1) = \frac{1}{3}, \end{cases} \tag{7.49}$$

cuya solución exacta es

$$y(t) = \frac{1}{4-t^2}.$$

Para aplicar el método de disparo al problema (7.49) es necesario resolver, en cada iteración, el siguiente problema de valor inicial de segundo orden.

$$\begin{cases} y''(t) = 2y^2(t) + 4ty(t)y'(t), \\ y(0) = \frac{1}{4}, \qquad y'(0) = z. \end{cases}$$

Esta ecuación de segundo orden se resuelve usando la forma vectorial del método RK4. Para ello, se construye el sistema de ecuaciones diferenciales de primer orden asociado. Tomando $w_1(t) = y(t)$ y $w_2(t) = y'(t)$, el sistema anterior adquiere la forma

$$\begin{cases} w_1'(t) = w_2(t), \\ w_2'(t) = 2w_1^2(t) + 4t\, w_1(t)\, w_2(t), \\ w_1(0) = \frac{1}{4}, \qquad w_2(0) = z. \end{cases} \tag{7.50}$$

Como primer paso, el Programa 7.13 genera una aproximación inicial de $y'(0)$, a saber:

$$z_1 = \frac{\beta - \alpha}{b - a} = \frac{1}{3} - \frac{1}{4} = \frac{1}{12}$$

y resuelve el sistema (7.50) con $z = z_1$. Luego, dependiendo del signo de la diferencia entre el valor final $w_1(1)$ de la solución $w_1(t)$ y el valor final $y(1) = \beta$, hace que la siguiente aproximación de $y'(0)$, digamos z_2, sea más grande o más pequeña que la aproximación inicial z_1; seguidamente resuelve el sistema (7.50) nuevamente con $z = z_2$. Inmediatamente, el Programa 7.13 pone en marcha el método de la secante con los dos valores iniciales z_1 y z_2 para obtener z_3, y repite la iteración hasta que la diferencia $|\varphi(z_{n+1}) - \beta|$ sea lo suficientemente pequeña. Corramos el Programa 7.13 para obtener la solución numérica y compararla con la solución exacta; antes de ello debemos definir la función `f`,

```
>>> def f(t,w): return np.array([w[1],2.*w[0]**2+4.*t*w[0]*w[1]])
```

Ahora aplicamos el Programa 7.13 con $h = 0.1$ como longitud de paso inicial y `tol` $= 10^{-6}$ como tolerancia.

```
>>> tn,yn,k = disparo(f,0.,1.,1/4,1/3,tol=1e-3)
>>> print(k)
6
```

Aquí, `tn` es el arreglo de nodos t_n, `yn` es el arreglo de las aproximaciones y_n y k es el número de iteraciones realizadas por el método de disparo. Se aprecia que el método halló una solución numérica en 6 iteraciones. La Figura 7.13 muestra que la solución numérica se acerca más a la solución exacta a medida que pasan las iteraciones. Además, se obtuvo el error

$$|y(b) - \beta| = 1.360082 \times 10^{-8},$$

para la solución obtenida.

7.8.2 Método de Diferencias Finitas

Los métodos directos para resolver problemas de valores en la frontera discretizan la ecuación en su forma original para producir un sistema finito. Una de las formas más obvias de hacer esto es usar diferencias finitas en lugar de derivadas. En el Capítulo 5 mostramos como aproximar derivadas usando diferencias. Las fórmulas de diferencias más usadas son las diferencias centrales sobre una red uniforme de ancho h, específicamente las fórmulas

$$y''(t) \approx \frac{y(t+h) - 2y(t) + y(t-h)}{h^2}, \tag{7.51}$$

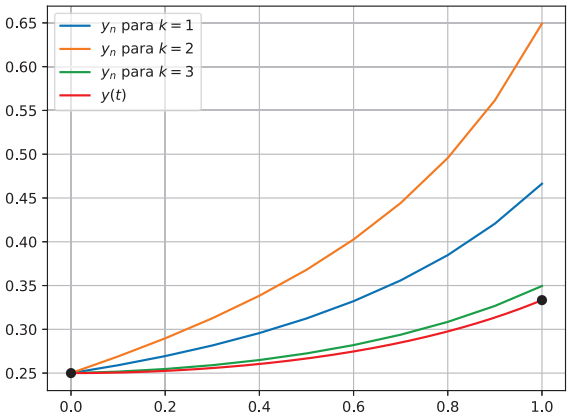


Figura 7.13. Solución numérica del problema de valores en la frontera en dos puntos (7.50). Se reporta las soluciones obtenidas en las iteraciones: $k=1$ (línea azul), $k=2$ (línea naranja) y $k=3$ (línea verde). También se muestra la solución exacta $y(t)$ (línea roja).

y

$$y'(t) \approx \frac{y(t+h)-y(t-h)}{2h}. \tag{7.52}$$

Para $y(t)$ suficientemente suave, las aproximaciones anteriores tienen $\mathbf{O}(h^2)$ de exactitud.

Para resolver el problema de valores en la frontera en dos puntos (7.46) utilizando el método de diferencia finita, primero se selecciona un conjunto de puntos igualmente espaciados $t_0, t_1, \ldots, t_N$ en el intervalo $[a,b]$ dados por:

$$t_i = a+ih, \qquad \text{con} \qquad h=\frac{b-a}{N}, \quad i=0,1,\ldots,N.$$

Seguidamente, se aproximan las derivadas usando las fórmulas (7.51) y (7.51), en cada punto t_i. El valor aproximado de $y(t_i)$ se denota por y_i. De esta forma, el problema (7.46) se convierte en el siguiente sistema de ecuaciones:

$$\begin{cases} y_0 = \alpha \\ \dfrac{1}{h^2}(y_{i+1}-2y_i+y_{i-1}) = f\left(t_i, y_i, \dfrac{1}{2h}(y_{i+1}-y_{i-1})\right), \quad i=1,2,\ldots,N-1 \\ y_N = \beta \end{cases} \tag{7.53}$$

Generalmente, (7.53) es un sistema de ecuaciones no lineales con $N-1$ incógnitas, $y_1,\ldots,y_{N-1}$, ya que f usualmente envuelve relaciones no lineales de y_i. La solución de un sistema de este tipo rara vez es fácil, pero podría abordarse utilizando los métodos del Capítulo 4. El método más convencional para resolver el sistema (7.53) es el método de Newton. El sistema (7.53) se puede escribir vectorialmente como:

$$F(\mathbf{y}) = A\mathbf{y} - \mathbf{G}(\mathbf{y}) = 0, \tag{7.54}$$

donde $\mathbf{y} = (y_0, y_1, \ldots, y_N)$,

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & -2 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & \cdots & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{G}(\mathbf{y}) = \begin{bmatrix} \alpha \\ h^2 f\left(t_1, y_1, \frac{y_2 - y_0}{2h}\right) \\ h^2 f\left(t_2, y_2, \frac{y_3 - y_1}{2h}\right) \\ \vdots \\ h^2 f\left(t_{N-1}, y_{N-1}, \frac{y_N - y_{N-2}}{2h}\right) \\ \beta \end{bmatrix}.$$

El método de Newton aplicado al sistema (7.54) genera una sucesión $\{\mathbf{y}_k\}$ construida mediante la fórmula iterativa

$$\mathbf{y}_{k+1} = \mathbf{y}_k + \mathbf{s}_k, \qquad k = 0, 1, 2, \ldots$$

donde $\mathbf{s}_k$ es la solución del sistema de ecuaciones lineales

$$\left(A - \mathbf{G}'(\mathbf{y}_k)\right) \mathbf{s}_k = \mathbf{G}(\mathbf{y}_k) - A\mathbf{y}_k,$$

con $\mathbf{G}'(\mathbf{y})$ el Jacobiano de $\mathbf{G}(\mathbf{y})$.

El Caso Lineal

Un caso importante de (7.46) es el *problema lineal de valores en la frontera en dos puntos*,

$$\begin{cases} y''(t) = p(t)y'(t) + q(t)y(t) + r(t), \\ y(a) = \alpha, \qquad y(b) = \beta, \end{cases}$$

donde las funciones p, q y r son continuas y $q(t) > 0$ para todo $t \in [a, b]$. En este caso especial, la ecuación principal en el Sistema (7.53) se convierte en

$$\frac{1}{h^2}(y_{i+1} - 2y_i + y_{i+1}) = \frac{1}{2h}\, p(t_i)\,(y_{i+1} - y_{i-1}) + q(t_i)y_i + r(t_i), \quad i = 1, 2, \ldots, N-1$$

o, equivalentemente,

$$-\left(1 + \frac{h}{2}p_i\right) y_{i-1} + \left(2 + h^2 q_i\right) y_i - \left(1 - \frac{h}{2}p_i\right) y_{i+1} = -h^2 r_i, \quad i = 1, 2, \ldots, N-1, \qquad (7.55)$$

donde $p_i = p(t_i)$, $q_i = q(t_i)$ y $r_i = r(t_i)$. Ahora, colocando

$$\begin{cases} a_i = -\left(1 + \frac{h}{2}p_i\right), \\ d_i = \left(2 + h^2 q_i\right), \\ c_i = -\left(1 - \frac{h}{2}p_i\right), \\ v_i = -h^2 r_i, \end{cases}$$

la Ecuación (7.55) adquiere la forma

$$a_i y_{i-1} + d_i y_i + c_i y_{i+1} = v_i, \quad i = 1, 2, \ldots, N-1.$$

Las ecuaciones correspondientes a $i = 1$ e $i = N-1$ son diferentes, ya que conocemos y_0 e y_N. Así, el sistema se puede escribir como:

$$\begin{cases} d_1 y_1 + c_1 y_2 &= v_1 - a_1\alpha \\ a_i y_{i-1} + d_i y_i + c_i y_{i+1} &= v_i, \qquad i = 2,3,\ldots,N-2 \\ a_{N-1} y_{N-2} + d_{N-1} y_{N-1} &= v_{N-1} - c_{N-1}\beta \end{cases} \tag{7.56}$$

La forma matricial del Sistema (7.56) es:

$$A\mathbf{y} = \mathbf{b},$$

donde

$$A = \begin{bmatrix} d_1 & c_1 & & & & \\ a_2 & d_2 & c_2 & & & \\ & a_3 & d_3 & c_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & a_{N-2} & d_{N-2} & c_{N-2} \\ & & & & a_{N-1} & d_{N-1} \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{N-2} \\ y_{N-1} \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} v_1 - a_1\alpha \\ v_2 \\ v_3 \\ \vdots \\ v_{N-2} \\ v_{N-1} - c_{N-1}\beta \end{bmatrix}.$$

Como la matriz A es tridiagonal, podemos resolver el sistema $A\mathbf{y} = \mathbf{b}$ con la eliminación Gaussiana con pivoteo parcial (ver Capítulo 2). En el Programa 7.14 se aprecia la implementación del método de diferencia finita para el caso lineal. Aquí empleamos la Eliminación Gaussiana con pivoteo parcial y la Sustitución Regresiva, cuya implementación está dada en los Programas 2.4 y 2.1, respectivamente.

Programa 7.14. Método de Diferencia Finita para el Caso Lineal

```
def diffincl(a,b,alfa,beta,p,q,r,N=100):
    ti = np.linspace(a,b,N+1)
    h = ti[1]-ti[0]
    yi = np.empty(N+1)
    yi[0],yi[-1] = alfa,beta
    pi = h*p(ti[1:-1])/2.
    ai = -1.-pi
    di = (2.+h**2*q(ti[1:-1]))
    ci = -1.+pi
    vi = -h**2*r(ti[1:-1])
    A = np.diag(ai[1:],k=-1)+np.diag(di,k=0)+np.diag(ci[1:],k=1)
    vb = np.empty(N-1)
    vb[0] = vi[0]-ai[0]*alfa
    vb[1:-1] = vi[1:-1]
    vb[-1] = vi[-1] - ci[-1]*beta
    A,vb = gaussiana_pp(A,vb)
    yi[1:-1] = Sust_Regresiva(A,vb)
    return ti,yi
```

7.9 Ejercicios

1. Demuestre que el problema de valor inicial

$$y'(t) = \tan y(t), \quad y(0) = 0,$$

tiene una solución en el intervalo $|t| < \pi/4$. ¿Es única la solución?

2. Demuestre que el problema de valor inicial

$$y'(t) = t^2 + e^{y(t)}, \quad y(0) = 0,$$

tiene una solución única en el intervalo $|t| \leq 0.351$.

3. Muestre que el problema de valor inicial

$$y'(t) = 1 + x(t) + x(t)^2 \cos t, \quad y(0) = 0,$$

tiene una solución en el intervalo $-1/3 \leq t \leq 1/3$. ¿Es única la solución?

4. Si se desea resolver mediante el método de Runge-Kutta de orden 4 un problema de valor inicial que atañe a la ecuación diferencial $y(t) + 2ty'(t) + y(t)y'(t) = 3$, ¿cuál es la función que debe programarse?

5. Reduzca la siguiente ecuación diferencial en un sistema de ecuaciones diferenciales de primer orden:

$$ay'''(t) + by''(t) + cy'(t) + dy(t) = e, \quad y(0) = \alpha_0, \quad y'(0) = \alpha_1, \quad y''(0) = \alpha_2.$$

Determine si el sistema de ecuaciones diferenciales de primer orden posee solución única.

6. Reduzca las siguientes ecuaciones diferenciales en un sólo sistema de ecuaciones diferenciales de primer orden:

$$\begin{aligned} &ay''(t) + by'(t) + cz'(t) + dy(t) + ez(t) = F(t), \quad y(0) = \alpha_0, \quad y'(0) = \alpha_0, \\ &Az''(t) + By'(t) + Cz'(t) + Dy(t) + Ez(t) = G(t), \quad z(0) = \beta_0, \quad z'(0) = \beta_0. \end{aligned}$$

7. El desplazamiento angular $\theta(t)$ de un péndulo sin fricción está gobernado por la siguiente ecuación diferencial:

$$\theta''(t) + \frac{g}{L} \sin \theta(t) = 0, \quad \theta(0) = \alpha_0, \quad \theta'(0) = \alpha_1.$$

Reduzca esta ecuación diferencial de segundo orden en un sistema de ecuaciones diferenciales de primer orden y determine un intervalo $[0, b]$ donde existe una solución única.

8. La ecuación diferencial que gobierna la mezcla de un extracto laminado es

$$\frac{d^3 f}{d\eta^3} + f \frac{df}{d\eta} + \left(\frac{df}{d\eta} \right)^2 = 0, \quad f(0) = 0, \quad f'(0) = 0, \quad \lim_{\eta \to \infty} f'(\eta) = 0.$$

Reduzca esta ecuación diferencial de tercer orden en un sistema de ecuaciones diferenciales de primer orden.

9. Estudie la estabilidad, consistencia y convergencia del siguiente método:

$$y_{n+2} + 4y_{n+1} - 5y_n = h(4f_{n+1} + 2f_n).$$

10. Estudie la estabilidad, consistencia y convergencia del siguiente método:

$$y_n - 2y_{n-1} + y_{n-2} = h(f_n - f_{n-1}).$$

Problemas Computacionales

11. Los métodos para resolver problemas de valor inicial también se pueden usar para calcular integrales, tanto definidas como indefinidas. Por ejemplo, podemos calcular

$$\int_0^2 e^{-s^2}\, ds,$$

resolviendo el problema de valor inicial

$$y'(t) = e^{-t^2}, \quad y(0) = 0,$$

con t en el intervalo $[0,2]$. Dé la solución a este problema usando el método de Taylor de orden 4. Al consultar una tabla de la **función error**

$$\operatorname{erf}(t) = \frac{2}{\sqrt{\pi}} \int_0^t e^{-s^2}\, ds,$$

obtenemos $y(2) \approx 0.8820813907$.

12. La función

$$\varphi(t) = \int_0^t \operatorname{sen} s^2\, ds,$$

se conoce como **integral de Fresnel**. Haga una tabla de esta función en el intervalo $[0,10]$. Utilice el método de Taylor de orden 4 con $h = 0.1$. Si es posible, obtenga la gráfica de la función $\varphi(t)$.

13. Verifique que la función $y(t) = t^2/4$ resuelve el problema de valor inicial

$$y'(t) = \sqrt{y(t)}, \quad y(0) = 0.$$

Aplique el método de Euler y explique por qué la solución numérica difiere de la solución $y(t) = t^2/4$.

14. Calcule $y(0.1)$ resolviendo la ecuación diferencial

$$y'(t) = -tx(t)^2, \quad y(0) = 2,$$

con un paso del método de Taylor de orden 2.

15. Resolviendo un problema de valor inicial apropiado, calcule una tabla de la función **dilogaritmo**

$$f(t) = -\int_0^t \frac{\log(1-\tau)}{\tau}\, d\tau,$$

en el intervalo $0 \le t \le 5$ resolviendo un problema de valor inicial adecuado. Use el método de Runge-Kutta de orden 4 con $h = 1/64$.

16. Usando diferentes valores de λ, tales como 5, -5 o -10, resuelva numéricamente el siguiente problema de valor inicial con el método de Runge-Kutta de orden 4.

$$y'(t) = \lambda y(t) + \operatorname{sen} t - \lambda \cos t, \quad y(0) = 0.$$

Compare la solución numérica con la solución analítica en el intervalo $[0,5]$. Tome $h = 0.01$. ¿Cuál es el efecto de λ sobre la precisión numérica?

17. Use la forma vectorial del método de Runge-Kutta de orden 4 para resolver los siguientes sistemas de ecuaciones diferenciales de primer orden, desde $t = 0.0$ hasta 1.0, con $h = 0.2$ y $h = 0.1$.

a) $\begin{cases} y'(t) = 2y(t) + z(t) + 1 \\ z'(t) = y(t) + z(t) + 1 \\ y(0) = z(0) = 1 \end{cases}$

b) $\begin{cases} y'(t) = 2y(t) + z(t) + e^t + 1 + t \\ z'(t) = y(t) + z(t) + t \\ y(0) = 0,\ z(0) = 1 \end{cases}$

c) $\begin{cases} y'(t) = z(t) \\ z'(t) = -6.25y(t) - 4z(t) + 1 + t + 2e^t \\ y(0) = z(0) = 1 \end{cases}$

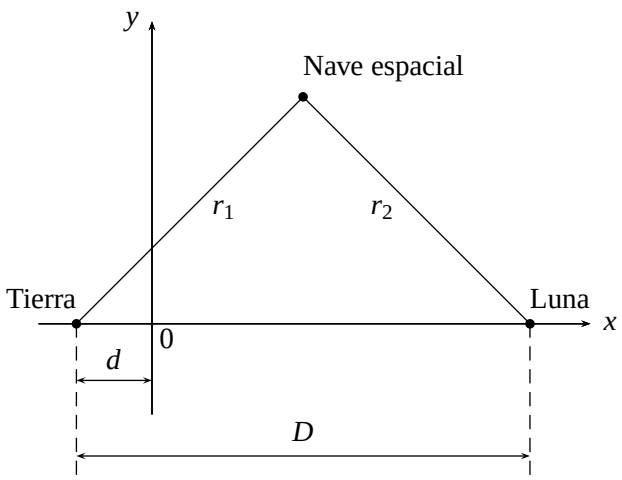
d) $\begin{cases} y'(t) = 0.1z(t)^2 + 1 \\ z'(t) = 0.1y(t)^2 + 0.1z(t)^2 \\ y(0) = z(0) = 1 \end{cases}$

18. El modelo de *Kermack-McKendrick* para el comportamiento de una epidemia en una población está dado por el sistema de ecuaciones diferenciales ordinarias

$$\begin{cases} y_1'(t) = -cy_1(t)y_2(t), \\ y_2'(t) = cy_1(t)y_2(t) - dy_2(t), \\ y_3'(t) = dy_2(t), \end{cases}$$

donde y_1 representa susceptibles, y_2 representa infectados en circulación, y y_3 representa infectados eliminados por aislamiento, muerte o recuperación e inmunidad. Los parámetros c y d representan la tasa de infección y la tasa de eliminación, respectivamente. Use la forma vectorial del método RK4 para resolver este sistema numéricamente, con los valores de los parámetros $c = 1$ y $d = 5$, y los valores iniciales $y_1(0) = 95$, $y_2(0) = 5$, $y_3(0) = 0$. Integrar desde $t = 0$ a $t = 1$. Grafique cada componente de la solución en el mismo gráfico en función de t. Como debería esperarse con una epidemia, primero debería crecer el número de infectados, luego disminuir a cero. Experimente con otros valores para los parámetros y condiciones iniciales. ¿Se pueden encontrar valores para los cuales la epidemia no crece, o para los cuales se destruye a toda la población?

19. Considere una forma restringida del problema de tres cuerpos en el que un cuerpo de pequeña masa orbita a otros dos cuerpos con masas mucho más grandes, como una nave espacial Apolo que orbita el sistema de la Tierra y la Luna. Usaremos un sistema de coordenadas bidimensional en el plano determinado por los tres cuerpos, con el origen en el centro de masa de los dos cuerpos más grandes, y el sistema de coordenadas girando de manera que los dos cuerpos más grandes aparezcan fijos. El sistema de coordenadas se muestra en el diagrama.



donde D es la distancia de la Tierra a la luna, d es la distancia desde el centro de la Tierra al centro de masa, r_1 es la distancia de la Tierra a la nave espacial, y r_2 es la distancia de la Luna a la nave espacial. Se supone que la masa de la nave espacial es insignificante en comparación con las otras masas.

Al utilizar las leyes de movimiento y gravitación de Newton, y al permitir las fuerzas centrífugas y de Coriolis debido a la rotación del sistema de coordenadas, el siguiente sistema de ecuaciones diferenciales ordinarias de segundo orden describe el movimiento de la nave espacial:

$$x''(t) = -G\left[\frac{M(x(t)+\mu D)}{r_1^3} + \frac{m(x(t)-\mu^* D)}{r_2^3}\right] + \Omega^2 x(t) + 2\Omega y'(t),$$
$$y''(t) = -G\left[\frac{My(t)}{r_1^3} + \frac{my(t)}{r_2^3}\right] + \Omega^2 y(t) + 2\Omega x'(t),$$

donde G es la constante gravitatoria, M y m son las masas de la Tierra y la Luna, μ^* y μ son las fracciones en masa de la Tierra y la Luna, y Ω es la velocidad angular de rotación de la Luna alrededor de la Tierra (y, por lo tanto, del sistema de coordenadas). Los valores numéricos de estas cantidades se dan continuación:

$$\begin{aligned}
G &= 6.67259 \times 10^{-11}\text{m}^3/(\text{kgs}^2),\\
M &= 5.974 \times 10^{24}\text{kg},\\
m &= 7.348 \times 10^{22}\text{kg},\\
\mu^* &= M/(m+M),\\
\mu &= m/(m+M),\\
D &= 3.844 \times 10^{8}\text{m},\\
d &= 4.669 \times 10^{6}\text{m},\\
r_1 &= [(x+d)^2 + y^2]^{1/2},\\
r_2 &= [(D-d-x)^2 + y^2]^{1/2},\\
\Omega &= 2.661 \times 10^{-6}/\text{s}.
\end{aligned}$$

Use la versión vectorial del método RK4 para resolver este sistema de ecuaciones diferenciales con condiciones iniciales

$$\begin{aligned}
x(0) &= 4.613 \times 10^{8}; \quad & y(0) &= 0;\\
x'(0) &= 0; & y'(0) &= -1074.
\end{aligned}$$

Grafique la trayectoria de la solución resultante $(x(t), y(t))$ en el plano como una función del tiempo. Indica las posiciones de la Tierra y la Luna en la gráfica. Calcule la solución para al menos una órbita completa (es decir, hasta que la nave vuelva a su ubicación original), que es desde $t = 0$ hasta aproximadamente $t = 2.4 \times 10^6$s. Experimente con varias longitud de paso h para ver cuánta diferencia hacen en la órbita para que esté realmente cerrada. Intente controlar el tamaño de paso utilizado por el método RK4 a medida que avanza la integración. ¿Cuándo el tamaño del paso se hace más pequeño o más grande? ¿Qué tan cerca está la nave espacial a la superficie de la tierra? (El radio de la Tierra es 6.378×10^6m, por lo que el centro de masa del sistema Tierra-Luna está realmente dentro de la Tierra).

8 | Optimización Numérica

La optimización es el término que a menudo se emplea para denotar el proceso de minimizar o maximizar una función $f(\mathbf{x})$. Consideraremos solamente el problema de minimización, ya que la maximización de $f(\mathbf{x})$ se logra simplemente minimizando $-f(\mathbf{x})$. En ingeniería, la optimización está estrechamente relacionada con el diseño. La función $f(\mathbf{x})$, denominada *función mérito* o *función objetivo*, modela la cantidad que deseamos mantener lo más pequeña posible, como el costo o el peso. Las componentes de $\mathbf{x}$, conocidas como *variables de decisión* o *variables de diseño*, son variables independientes que somos libres de ajustar. Las dimensiones físicas (longitudes, áreas, ángulos, etc.) son ejemplos comunes de variables de decisión.

La optimización es un tema importante en matemáticas aplicadas y existen muchos libros especializados. Como el objetivo de este libro no es un estudio riguroso de la optimización y, además, el espacio que le dedicaremos es limitado, nos centraremos en introducir algunos métodos básicos que son lo suficientemente buenos para un gran número de problemas y se comportan razonablemente bien. Al omitir los métodos más sofisticados, es posible que realmente no olvidemos tanto. Todos los algoritmos de optimización no son completamente fiables; cualquiera puede funcionar en un problema y fallar en otro. Se tiene una regla general: al subir en sofisticación se gana eficiencia computacional, pero no necesariamente se gana confiabilidad. Los algoritmos de minimización son procedimientos iterativos que requieren valores iniciales de las variables de decisión $\mathbf{x}$. Si $f(\mathbf{x})$ tiene varios mínimos locales, la elección inicial de $\mathbf{x}$ determina cuál de estos se calculará. No existe un procedimiento que garantice encontrar el *punto óptimo global* o, simplemente, *óptimo global*, salvo en casos excepcionales. El procedimiento frecuentemente usado para encontrar una aproximación del óptimo global, es hacer varias corridas del algoritmo usando diferentes puntos de partida y elegir el mejor resultado.

A menudo, las variables de decisión también están sujetas a *restricciones* que pueden tener la forma de igualdades o desigualdades. Los problemas de optimización cuyas variables de decisión están sujetas a restricciones se denominan problemas de *optimización con restricciones*; en cambio, si las variables de decisión no tienen restricciones, se denominan problemas de *optimización sin restricciones*. La mayoría de los métodos disponibles están diseñados para la optimización sin restricciones. En estos problemas, los mínimos (si existen) son puntos estacionarios (puntos donde el vector gradiente de $f(\mathbf{x})$ se anula). Los problemas de optimización con restricciones son más difíciles de resolver, los mínimos generalmente se ubican donde la superficie $f(\mathbf{x})$ cumple con las restricciones. Existen algoritmos especializados para la optimización con restricciones, pero no son fácilmente accesibles debido a su complejidad y especialización. Nos centraremos en la descripción de los métodos básicos de la optimización sin restricciones.

8.1 Optimización sin Restricciones

Un problema de optimización sin restricciones se puede expresar matemáticamente como

$$\text{minimizar } f(\mathbf{x}) \quad \text{sujeto a } \mathbf{x} \in \mathbb{R}^n, \tag{8.1}$$

donde $f : \mathbb{R}^n \to \mathbb{R}$ es la función objetivo y $n \geq 1$. Cuando $n = 1$ el problema de optimización (8.1) se denomina *unidimensional*, y se llama *multidimensional* cuando $n \geq 2$. En este capítulo solo trataremos problemas multidimensionales. En esta sección y en el resto del capítulo, la norma $\|\cdot\|$ denotará la norma Euclídea. También denotaremos con $\nabla f(\mathbf{x})$ y $\nabla^2 f(\mathbf{x})$ al gradiente y la Hessiana de f en $\mathbf{x}$, respectivamente.

8.1.1 Condiciones de Optimalidad

En esta sección presentaremos las condiciones de optimalidad para el problema (8.1), que incluyen condiciones de primer orden y condiciones de segundo orden. En general, tenemos dos tipos de mínimos: *locales* y *globales* (ver Figura 8.1). A continuación damos sus definiciones exactas.

Definición 8.1. Un punto $\mathbf{x}^* \in \mathbb{R}^n$ se denomina *mínimo local* de una función $f : \mathbb{R}^n \to \mathbb{R}$, si existe $\delta > 0$ tal que $f(\mathbf{x}^*) \leq f(\mathbf{x})$ para todo $\mathbf{x} \in \mathbb{R}^n$ con $\|\mathbf{x} - \mathbf{x}^*\| < \delta$. Un punto $\mathbf{x}^* \in \mathbb{R}^n$ se llama *mínimo local estricto* de f, si $f(\mathbf{x}^*) < f(\mathbf{x})$ para todo $\mathbf{x} \in \mathbb{R}^n$ con $\mathbf{x} \neq \mathbf{x}^*$ y $\|\mathbf{x} - \mathbf{x}^*\| < \delta$.

Definición 8.2. Un punto $\mathbf{x}^* \in \mathbb{R}^n$ se denomina *mínimo global* de una función $f : \mathbb{R}^n \to \mathbb{R}$, si $f(\mathbf{x}^*) \leq f(\mathbf{x})$ para todo $\mathbf{x} \in \mathbb{R}^n$. Un punto $\mathbf{x}^* \in \mathbb{R}^n$ se llama *mínimo global estricto* de f, si $f(\mathbf{x}^*) < f(\mathbf{x})$ para todo $\mathbf{x} \in \mathbb{R}^n$ con $\mathbf{x} \neq \mathbf{x}^*$.

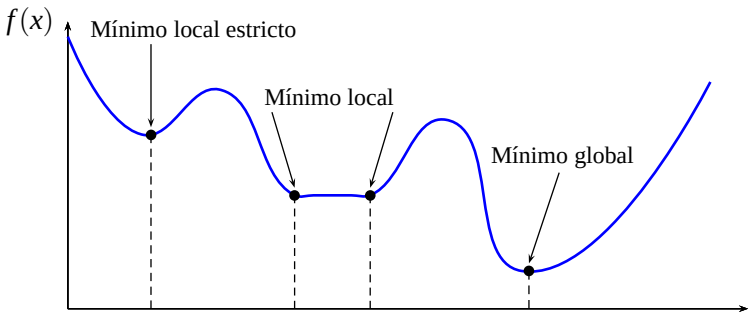


Figura 8.1. Tipos de mínimos

Es de notar que, en la práctica, la mayoría de los algoritmos solo pueden encontrar un mínimo local. Normalmente, encontrar un mínimo global es una tarea difícil. En muchas aplicaciones prácticas es suficiente con obtener un mínimo local. Además, muchos algoritmos de optimización global utilizan algoritmos de optimización local para generar secuencias de puntos. Por ello, solo describiremos algoritmos de optimización local. Por lo general, cuando digamos mínimo nos referiremos a un mínimo local.

Las direcciones de descenso son muy importantes en el desarrollo de los métodos de optimización local. Seguidamente damos su definición exacta.

Definición 8.3. Sea $f : \mathbb{R}^n \to \mathbb{R}$ una función diferenciable en $\mathbf{x} \in \mathbb{R}^n$. Si existe un vector $\mathbf{d} \in \mathbb{R}^n$ tal que

$$\nabla f(\mathbf{x})^T \mathbf{d} < 0,$$

entonces se dice que $\mathbf{d}$ es una *dirección de descenso* de f en $\mathbf{x}$.

En el siguiente teorema se presentan las condiciones necesarias de optimalidad de primer orden.

Teorema 8.1 (Condiciones Necesarias de Primer Orden)**.** *Sea $f : D \subset \mathbb{R}^n \to \mathbb{R}$ una función continuamente diferenciable en D. Si $\mathbf{x}^* \in D$ es un mínimo local de* (8.1)*, entonces*

$$\nabla f(\mathbf{x}^*) = 0.$$

El Teorema 8.1 establece que si $\mathbf{x}^*$ es un mínimo local, entonces f tiene una pendiente cero en $\mathbf{x}^*$. Ahora, el Teorema 8.2 indica que si $\mathbf{x}^*$ es un mínimo local, entonces f tiene una curvatura no negativa en el mínimo local, además de pendiente cero. En el Teorema 8.3 se presentan las condiciones suficientes de optimalidad de segundo orden.

Teorema 8.2 (Condiciones Necesarias de Segundo Orden)**.** *Sea $f : D \subset \mathbb{R}^n \to \mathbb{R}$ una función dos veces continuamente diferenciable en D. Si $\mathbf{x}^* \in D$ es un mínimo local de* (8.1)*, entonces $\nabla f(\mathbf{x}^*) = 0$ y $\nabla^2 f(\mathbf{x}^*)$ es una matriz simétrica semidefinida positiva.*

Teorema 8.3 (Condiciones Suficientes de Segundo Orden)**.** *Sea $f : D \subset \mathbb{R}^n \to \mathbb{R}$ una función dos veces continuamente diferenciable en D. Si $\nabla f(\mathbf{x}^*) = 0$ y $\nabla^2 f(\mathbf{x}^*)$ es una matriz simétrica y definida positiva, entonces $\mathbf{x}^* \in D$ es un mínimo local estricto de* (8.1)*.*

Definición 8.4. Sea $f : \mathbb{R}^n \to \mathbb{R}$ una función diferenciable en $\mathbf{x}^* \in \mathbb{R}^n$. Si $\nabla f(\mathbf{x}^*) = 0$, entonces $\mathbf{x}^*$ se denomina *punto estacionario* o *punto crítico* de f.

El Teorema 8.1 nos dice que si $\mathbf{x}^*$ es un mínimo local, entonces es un punto estacionario. Sin embargo, lo contrario no es cierto. Si $\mathbf{x}^*$ es un punto estacionario, es posible que $\mathbf{x}^*$ sea un mínimo o un máximo local, también es posible que $\mathbf{x}^*$ no sea un punto extremo. Si un punto estacionario $\mathbf{x}^*$ no es mínimo ni máximo, se denomina punto de *inflexión* (o punto de silla de montar). Por lo tanto, un punto estacionario no es necesariamente un mínimo local.

En la práctica, las condiciones de optimalidad, en particular las condiciones de segundo orden, son difíciles de comprobar, además a ciertas funciones objetivo solo se le puede hallar el gradiente. En general, el objetivo de los algoritmos de optimización local es encontrar un punto estacionario de f, por supuesto, garantizando que sea un mínimo local.

8.1.2 Esquema Básico de los Métodos de Optimización

Para los problemas de optimización multidimensional existen dos clases generales de métodos, a saber, los métodos de *búsqueda* o de *optimización global* y los métodos de *optimización local*. En los métodos de búsqueda global, la solución se obtiene utilizando solo evaluaciones de la función objetivo. Generalmente estos métodos exploran el espacio de búsqueda en forma organizada, con el propósito de encontrar una trayectoria que conduzca progresivamente a valores de las variables

de decisión que disminuyan el valor de la función objetivo. Un método rudimentario de este tipo podría consistir en ajustar todas las variables de decisión en un punto de partida específico, una a la vez, y luego seleccionar un nuevo punto comparando los valores calculados de la función objetivo. Este mismo procedimiento puede repetirse en el nuevo punto, y así sucesivamente. Los métodos de búsqueda no son muy eficientes. Como resultado de ello, su aplicación se limita a problemas donde no esté disponible información de primer o segundo orden, o es difícil de obtener, por ejemplo, en aplicaciones donde la función objetivo no es continua.

Por otro lado, los métodos de optimización local son procesos iterativos que encuentran un mínimo local. Estos pueden dividirse en dos grupos: libre de derivadas (derivative-free) y con derivadas. Los métodos de optimización local libre de derivadas, no utilizan en forma explícita el gradiente o la Hessiana de f, sino que emplean solo evaluaciones de la función. Nosotros nos centraremos en métodos con derivada. Por comodidad de notación, en esta sección y en las subsiguientes, denotaremos con $\mathbf{g}_k = \nabla f(\mathbf{x}_k)$, para $\mathbf{x}_k \in \mathbb{R}^n$.

Los métodos de optimización local con derivadas se pueden dividir en dos grupos: tipo gradiente y tipo Newton. Los métodos tipo gradiente solo utilizan información del gradiente de f; en cambio, los métodos tipo Newton utilizan tanto información del gradiente como de la Hessiana de f. Estudiaremos los métodos tipo gradiente y tipo Newton básicos, los cuales listamos a continuación:

- Métodos tipo gradiente: *Método de Cauchy* o *método de máximo descenso*, y *Métodos de Gradientes Conjugados*.
- Métodos tipo Newton: *Método de Newton*, y *Métodos Quasi-Newton*.

Los métodos tipo gradiente y tipo Newton son procesos iterativos cuya idea básica es generar, dado un punto inicial $\mathbf{x}_0 \in \mathbb{R}^n$, una sucesión de iterados $\{\mathbf{x}_k\}$ a través de una fórmula iterativa, de manera que un punto límite de esta sucesión sea un mínimo local $\mathbf{x}^*$. Un comportamiento típico de un algoritmo que se considera aceptable es que los iterados $\mathbf{x}_k$ se mueven constantemente hacia una vecindad de $\mathbf{x}^*$, y luego convergen rápidamente a $\mathbf{x}^*$. Cuando se cumple una regla de convergencia dada, el proceso iterativo termina. En general, el criterio de parada más natural es

$$\|\mathbf{g}_k\| \leq \varepsilon, \tag{8.2}$$

donde ε es una tolerancia prefijada. Si el criterio de parada (8.2) se cumple, implica que el gradiente $\mathbf{g}_k$ tiende a cero y la sucesión $\{\mathbf{x}_k\}$ converge a un punto estacionario. El esquema básico de un método de optimización local se muestra en el Algoritmo 8.1.

Algoritmo 8.1 (Esquema básico de un método de optimización local)**.**

1: Sean $\mathbf{x}_0 \in \mathbb{R}^n$ un iterado inicial y una tolerancia $\varepsilon > 0$. Tome $k := 0$.
2: **while** $\|\mathbf{g}_k\| > \varepsilon$ **do** ▷ Criterio de parada
3: Encuentre una dirección de descenso $\mathbf{d}_k$.
4: Determine la longitud de paso $\lambda_k > 0$ de manera que: ▷ Búsqueda lineal

$$f(\mathbf{x}_k + \lambda_k \mathbf{d}_k) < f(\mathbf{x}_k).$$

5: Defina $\mathbf{x}_{k+1} = \mathbf{x}_k + \lambda_k \mathbf{d}_k$ y tome $k := k+1$.
6: **end while**

Como se indica en el Algoritmo 8.1, los algoritmos de optimización local consideran un iterado $\mathbf{x}_k \in \mathbb{R}^n$, una dirección de descenso $\mathbf{d}_k \in \mathbb{R}^n$ y una longitud de paso $\lambda_k > 0$. El $k+1$-ésimo iterado

se genera por la siguiente fórmula iterativa:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \lambda_k \mathbf{d}_k, \quad k \geq 0. \tag{8.3}$$

De (8.3) se infiere que la manera de tomar una longitud de paso y escoger una dirección de descenso $\mathbf{d}_k$ da a lugar diversos métodos.

Búsqueda Lineal

En el esquema básico de los métodos de optimización local dado en el Algoritmo 8.1, la mayor parte del esfuerzo computacional se emplea en encontrar la longitud de paso $\lambda_k > 0$ tal que $f(\mathbf{x}_k + \lambda_k \mathbf{d}_k) < f(\mathbf{x}_k)$. Tal proceso se denomina *búsqueda lineal*. En consecuencia, la cantidad de cálculo requerida en los algoritmos de optimización que estudiaremos dependerá de la eficiencia y la precisión de la búsqueda lineal utilizada. Si se necesitan búsquedas lineales de alta precisión, la cantidad de cálculos será grande.

Al calcular la longitud de paso λ_k nos enfrentamos a una disyuntiva. Nos gustaría elegir λ_k para reducir sustancialmente el valor de f, pero al mismo tiempo no queremos perder demasiado tiempo haciendo la elección. La elección ideal sería la solución del problema de minimización

$$\min_{\lambda > 0} \varphi(\lambda) = f(\mathbf{x}_k + \lambda \mathbf{d}_k),$$

pero, en general, hallar el mínimo global de φ requiere de un alto costo computacional. Encontrar incluso un mínimo local de φ generalmente requiere demasiadas evaluaciones de la función objetivo f y posiblemente del gradiente ∇f. Las estrategias más eficientes realizan una búsqueda lineal inexacta para identificar una longitud de paso que logre reducciones adecuadas en f a un costo mínimo.

Los procedimientos de búsqueda lineal más utilizados prueban secuencialmente candidatos de λ, aceptando uno de estos cuando se satisfacen ciertas condiciones. La búsqueda lineal se realiza en dos etapas: una primera fase que encuentra un intervalo que contiene longitudes de paso aceptables, y una segunda fase de bisección o interpolación donde se calcula una buena longitud de paso dentro de ese intervalo. Una condición de búsqueda lineal muy popular exige que la longitud de paso debería producir una reducción de f que sea proporcional tanto a la longitud de paso λ_k como a la derivada direccional $\mathbf{g}_k^T \mathbf{d}_k$; en otras palabras, esta búsqueda lineal inexacta escoge λ_k tal que satisfaga la siguiente desigualdad:

$$f(\mathbf{x}_k + \lambda) \leq f(\mathbf{x}_k) + \gamma \lambda \mathbf{g}_k^T \mathbf{d}_k, \tag{8.4}$$

para alguna constante $\gamma \in (0,1)$. La desigualdad (8.4) se denomina *condición de Armijo*, cuya interpretación gráfica se muestra en la Figura 8.2. El lado derecho de (8.4),

$$\ell(\lambda) = f(\mathbf{x}_k) + \gamma \lambda \mathbf{g}_k^T \mathbf{d}_k,$$

es una recta con pendiente negativa $\gamma \mathbf{g}_k^T \mathbf{d}_k$; además, como $\gamma \in (0,1)$, $\ell(\lambda)$ está por encima del gráfico de φ para valores pequeños de $\lambda > 0$. La condición de Armijo establece que λ es aceptable solamente si $\varphi(\lambda) \leq \ell(\lambda)$. A manera de ejemplo, en la Figura 8.2 se muestran los intervalos en los que se cumple la condición de Armijo. En la práctica, el parámetro γ se elige bastante pequeño, digamos $\gamma = 10^{-4}$.

Existen búsquedas lineales más sofisticadas cuya implementación puede ser bastante complicada, por lo que no las estudiaremos en este libro (para ver otras búsquedas lineales el lector puede revisar los siguientes libros CITAS).

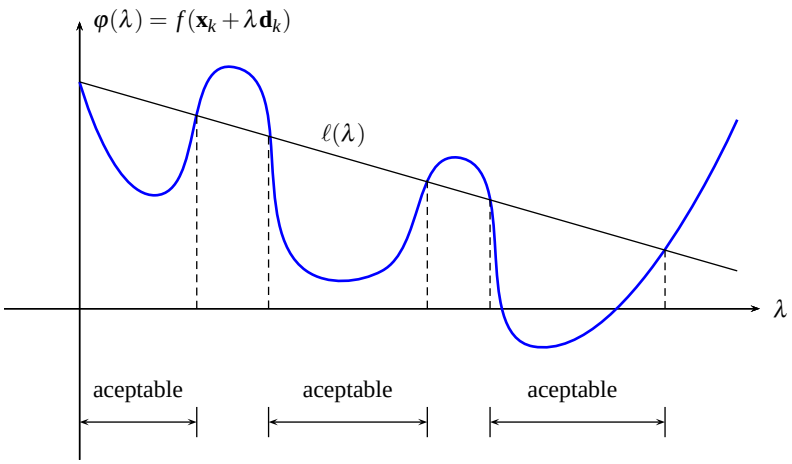


Figura 8.2. Intervalos de longitudes de paso que satisfacen la condición de Armijo.

Selección de la Longitud de Paso por Backtracking

Pasemos ahora a explicar como la búsqueda lineal elige λ_k mediante el uso del proceso de *"backtracking"* (reducción). Las implementaciones modernas del proceso de backtracking comienzan tomando $\lambda = 1$ y luego verifican que este valor de λ satisface o no la condición de Armijo (8.4). En caso que $\lambda = 1$ satisfaga (8.4), entonces toman $\lambda_k = 1$ como longitud de paso; en caso contrario, si $\lambda = 1$ no satisface (8.4), entonces reducen secuencialmente el valor de λ hasta encontrar un valor de λ que satisfaga (8.4), tomando la última reducción de λ como la longitud de paso λ_k. El algoritmo del proceso de backtracking se muestra a continuación. En el Programa 8.1 se muestra la implementación del proceso de backtracking.

Algoritmo 8.2 (Proceso de Backtracking)**.**

1: Escoja $\sigma \in (0,1)$ y $\gamma \in (0,1)$.
2: Tome $\lambda := 1$.
3: **while** $f(\mathbf{x}_k + \lambda \mathbf{d}_k) > f(\mathbf{x}_k) + \gamma\lambda \nabla f(\mathbf{x}_k)^T \mathbf{d}_k$ **do**
4: $\quad \lambda := \sigma\lambda$
5: **end while**
6: Defina $\lambda_k = \lambda$.

Programa 8.1. Proceso de Backtracking

```
def backtracking(f,x_k,f_k,g_k,d_k):
    lam, fc, gamma,sigma=1.0,0,0.0001,0.5
    gkdk = g_k.dot(d_k)
    while True:
        x_new = x_k + lam*d_k
        f_new = f(x_new)
        fc += 1
        if f_new > f_k + lam*gamma*gkdk:
            lam *= sigma
        else:
            return x_new, f_new, fc
```

Como dijimos anteriormente, el valor de γ se elige lo suficientemente pequeño; en la práctica se toma $\gamma = 10^{-4}$. En cuanto al parámetro σ, habitualmente se toma $\sigma = 0.5$, es decir, en cada reducción se considera la mitad de la longitud de paso anterior. Veamos un ejemplo del uso del proceso de backtracking para seleccionar la longitud de paso.

Ejemplo 8.1. Sea $f : \mathbb{R}^2 \to \mathbb{R}$ la función objetivo definida por

$$f(\mathbf{x}) = x_1^2 + 4x_2^2 + 2x_1x_2.$$

Se tiene que $\nabla f(\mathbf{x}) = (2x_1 + 2x_2\,,\, 2x_1 + 8x_2)$. Tomemos $\sigma = 0.5$ y $\gamma = 10^{-4}$ en el Algoritmo 8.2. Supongamos que el *k*-ésimo iterado del Algoritmo 8.1 es $\mathbf{x}_k = (1,1)$ y tomemos $\mathbf{d}_k = -\mathbf{g}_k = (-4,-10)$ como dirección de descenso. Así, el valor de la función objetivo en $\mathbf{x}_k$ es $f(\mathbf{x}_k) = 6$. Inicialmente, el proceso de backtracking toma $\lambda = 1$, calcula $f(\mathbf{x}_k + \mathbf{d}_k)$ y verifica la condición de Armijo (8.4),

$$f(\mathbf{x}_k + \mathbf{d}_k) = f((-3,-9)^T) = 387 > f(\mathbf{x}_k) + \gamma\lambda\mathbf{g}_k^T\mathbf{d}_k = \frac{17471}{2500} = 6.9884,$$

es decir, $\lambda = 1$ no satisface la condición (8.4), por ello se reduce su tamaño

$$\lambda := \sigma\lambda = \frac{1}{2}.$$

Ahora, verificamos si $\lambda = 1/2$ satisface o no la condición de Armijo,

$$f(\mathbf{x}_k + \lambda\mathbf{d}_k) = f((-1,-4)^T) = 73 > f(\mathbf{x}_k) + \gamma\lambda\mathbf{g}_k^T\mathbf{d}_k = \frac{34971}{5000} = 6.9942,$$

en otras palabras, $\lambda = 1/2$ tampoco satisface la condición (8.4), luego se debe reducir su tamaño

$$\lambda := \sigma\lambda = \frac{1}{4}.$$

Verificamos si $\lambda = 1/4$ satisface o no la condición de Armijo,

$$f(\mathbf{x}_k + \lambda\mathbf{d}_k) = f((0,-3/2)^T) = 9 > f(\mathbf{x}_k) + \gamma\lambda\mathbf{g}_k^T\mathbf{d}_k = \frac{69971}{10000} = 6.9971.$$

Se tiene que $\lambda = 1/4$ no satisface la condición (8.4), luego se debe reducir su tamaño

$$\lambda := \sigma\lambda = \frac{1}{8}.$$

Verificamos si $\lambda = 1/8$ satisface o no la condición de Armijo,

$$f(\mathbf{x}_k + \lambda\mathbf{d}_k) = f((1/2,-1/4)^T) = \frac{1}{4} < f(\mathbf{x}_k) + \gamma\lambda\mathbf{g}_k^T\mathbf{d}_k = \frac{139971}{20000} = 6.99855.$$

Apreciamos que $\lambda = 1/8$ efectivamente satisface la condición de Armijo. Por lo tanto, el tamaño de paso elegido es $\lambda_k = 1/8$; además, con el iterado $\mathbf{x}_{k+1} = \mathbf{x}_k + \lambda\mathbf{d}_k = (1/2,-1/4)$ se reduce el valor de la función objetivo de $f(\mathbf{x}_k) = 6$ a $f(\mathbf{x}_{k+1}) = 1/4$. Note que para obtener la longitud de paso $\lambda_k = 1/8$ se requirieron tres reducciones, con un costo computacional de 4 evaluaciones de la función objetivo y un producto interno.

■

8.1.3 Método de Cauchy

El *método de Cauchy* o *método de máximo descenso* es uno de los métodos de minimización más simples y fundamentales para la optimización sin restricciones. Deduzcamos el método de Cauchy. Utilizaremos la desigualdad de Cauchy-Schwartz (1.1).

Supongamos que la función $f : \mathbb{R}^n \to \mathbb{R}$ es continuamente diferenciable y $\mathbf{g}_k = \nabla f(\mathbf{x}_k) \neq 0$, con $\mathbf{x}_k \in \mathbb{R}^n$. Consideremos el desarrollo de Taylor centrado en $\mathbf{x}_k$,

$$f(\mathbf{x}) = f(\mathbf{x}_k) + \mathbf{g}_k^T(\mathbf{x} - \mathbf{x}_k) + o(\|\mathbf{x} - \mathbf{x}_k\|), \tag{8.5}$$

donde $o(\|\mathbf{x}-\mathbf{x}_k\|)$ tiende a cero cuando $\mathbf{x}$ tiende a $\mathbf{x}_k$. Si tomamos $\lambda \mathbf{d}_k = \mathbf{x} - \mathbf{x}_k$ con $\lambda > 0$, entonces por la continuidad de f, para $\mathbf{x}$ suficientemente cerca de $\mathbf{x}_k$, de (8.5) se infiere que

$$\mathbf{g}_k^T \mathbf{d}_k \approx f(\mathbf{x}) - f(\mathbf{x}_k) < 0,$$

en otras palabras, $\mathbf{d}_k$ es una dirección de descenso de f en $\mathbf{x}_k$ y

$$f(\mathbf{x}_k + \lambda) < f(\mathbf{x}_k).$$

Al fijar λ, se deduce que cuanto menor es el valor $\mathbf{g}_k^T \mathbf{d}_k$, más rápido disminuye el valor de la función. Por la desigualdad de Cauchy-Schwartz (1.1),

$$|\mathbf{g}_k^T \mathbf{d}_k| \leq \|\mathbf{g}_k\| \, \|\mathbf{d}_k\|$$

obtenemos que el valor de $\mathbf{g}_k^T \mathbf{d}_k$ es el más pequeño si, y sólo si $\mathbf{d}_k = -\mathbf{g}_k$. De esta forma, $\mathbf{d}_k = -\nabla f(\mathbf{x}_k)$ es la dirección de máximo descenso. De esta forma, el esquema iterativo del método de Cauchy es

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \lambda_k \mathbf{g}_k, \qquad k \geq 0.$$

Seguidamente damos el algoritmo del método de Cauchy.

Algoritmo 8.3 (Método de Cauchy)**.**

1: Sean $\mathbf{x}_0 \in \mathbb{R}^n$ un iterado inicial y una tolerancia $\varepsilon > 0$. Tome $k := 0$.
2: **while** $\|\mathbf{g}_k\| > \varepsilon$ **do** ▷ Criterio de parada
3: Tome $\mathbf{d}_k = -\mathbf{g}_k$ como dirección de descenso.
4: Encuentra la longitud de paso $\lambda_k > 0$ tal que ▷ Búsqueda lineal exacta

$$f(\mathbf{x}_k + \lambda_k \mathbf{d}_k) = \min_{\lambda > 0} f(\mathbf{x}_k + \lambda \mathbf{d}_k). \tag{8.6}$$

5: Defina $\mathbf{x}_{k+1} = \mathbf{x}_k + \lambda_k \mathbf{d}_k$ y tome $k := k + 1$.
6: **end while**

La dirección de mayor descenso es solo una propiedad local del método de Cauchy. Para muchos problemas, el método de Cauchy es efectivamente el de "mayor descenso", pero es muy lento. Aunque el método generalmente funciona bien en los primeros pasos, cuando se acerca a un punto estacionario, desciende muy lentamente con fenómenos en zigzag. Ilustremos este fenómeno de zigzagueo con un ejemplo.

Ejemplo 8.2. Consideremos el problema de optimización

$$\text{minimizar } f(\mathbf{x}) = \frac{1}{5}x_1^2 + x_2^2 \quad \text{sujeto a } \mathbf{x} \in \mathbb{R}^2, \tag{8.7}$$

cuyo óptimo global es $\mathbf{x}^* = (0,0)$. Tenemos que el gradiente de f está dado por:

$$\nabla f(\mathbf{x}) = \left(\frac{2}{5}x_1\,,\, 2x_2\right),$$

además, la dirección de descenso del método de Cauchy es $\mathbf{d}_k = -\mathbf{g}_k$, para todo $k \geq 0$. Sea $\varphi(\lambda) = f(\mathbf{x}_k + \lambda \mathbf{d}_k)$. Ahora, considerando la función objetivo, la expresión matemática de la función φ es:

$$\varphi(\lambda) = \frac{1}{5}\left(x_1^{(k)} + \lambda d_1^{(k)}\right)^2 + \left(x_2^{(k)} + \lambda d_2^{(k)}\right)^2.$$

donde $\mathbf{x}_k = \left(x_1^{(k)}, x_2^{(k)}\right)$ y $\mathbf{d}_k = \left(d_1^{(k)}, d_2^{(k)}\right)$. Se tiene que una solución del problema de optimización (8.6) de la búsqueda lineal exacta es un punto estacionario de φ, es decir, es una solución de la ecuación $\varphi'(\lambda) = 0$, que al considerar la expresión de φ dada arriba, obtenemos la siguiente ecuación no lineal:

$$\frac{2}{5}d_1^{(k)}\left(x_1^{(k)} + \lambda d_1^{(k)}\right) + 2d_2^{(k)}\left(x_2^{(k)} + \lambda d_2^{(k)}\right) = 0.$$

Despejando λ en la ecuación anterior, obtenemos que la longitud de paso λ_k del método de Cauchy es:

$$\lambda_k = -\frac{\frac{1}{5}d_1^{(k)}x_1^{(k)} + d_2^{(k)}x_2^{(k)}}{\frac{1}{5}\left(d_1^{(k)}\right)^2 + \left(d_2^{(k)}\right)^2}, \quad k \geq 0. \tag{8.8}$$

De esta forma, los iterados del método de Cauchy aplicado al problema (8.8) están dados por

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \lambda_k \mathbf{g}_k, \quad k \geq 0$$

con λ_k dado por (8.8). Veamos el comportamiento de estos iterados, tomando $\mathbf{x}_0 = (4,3)$ como iterado inicial, con $f(\mathbf{x}_0) = 12.2$.

- $k = 0$

$$\mathbf{x}_0 = \begin{bmatrix}4\\3\end{bmatrix}, \quad \mathbf{d}_0 = \begin{bmatrix}-\frac{8}{5}\\-6\end{bmatrix}, \quad \lambda_0 = \frac{1205}{2282}, \quad \Rightarrow \quad \mathbf{x}_1 = \begin{bmatrix}\frac{3600}{1141}\\-\frac{192}{1141}\end{bmatrix}, \quad f(\mathbf{x}_1) \approx 2.0193$$

- $k = 1$

$$\mathbf{d}_1 = \begin{bmatrix}-\frac{1440}{1141}\\\frac{384}{1141}\end{bmatrix}, \quad \lambda_1 = \frac{241}{122}, \quad \Rightarrow \quad \mathbf{x}_2 = \begin{bmatrix}\frac{46080}{69601}\\\frac{34560}{69601}\end{bmatrix}, \quad f(\mathbf{x}_2) \approx 0.3342$$

- $k = 2$

$$\mathbf{d}_2 = \begin{bmatrix}-\frac{18432}{69601}\\-\frac{69120}{69601}\end{bmatrix}, \quad \lambda_2 = \frac{1205}{2282}, \quad \Rightarrow \quad \mathbf{x}_3 = \begin{bmatrix}\frac{41472000}{79414741}\\-\frac{2211840}{79414741}\end{bmatrix}, \quad f(\mathbf{x}_3) \approx 0.0553$$

- $k = 3$

$$\mathbf{d}_3 \approx \begin{bmatrix}-0.2089\\0.0557\end{bmatrix}, \quad \lambda_3 \approx 1.9754, \quad \Rightarrow \quad \mathbf{x}_4 \approx \begin{bmatrix}4.3\times 10^{-94}\\5.1\times 10^{-92}\end{bmatrix}, \quad f(\mathbf{x}_4) \approx 4.8\times 10^{-124}$$

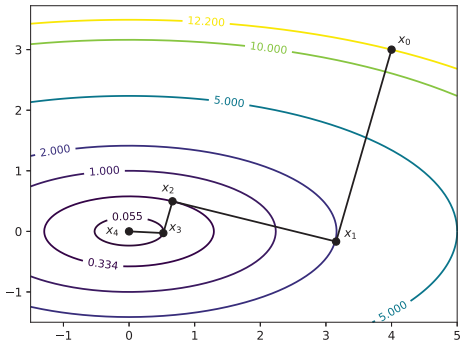


Figura 8.3. Zigzagueo en los iterados del método de Cauchy para el Ejemplo 8.2

El fenómeno de zigzagueo en los iterados del método de Cauchy se ilustra en la Figura 8.3.

En efecto, el zigzag del método de Cauchy que apreciamos en el ejemplo anterior se puede explicar por los siguientes hechos. Como la longitud de paso λ_k es un punto estacionario de la función $\varphi(\lambda) = f(\mathbf{x}_k + \lambda \mathbf{d}_k)$ y

$$\varphi'(\lambda) = \nabla f(\mathbf{x}_k + \lambda \mathbf{d}_k)^T \mathbf{d}_k = -\nabla f(\mathbf{x}_k + \lambda \mathbf{d}_k)^T \mathbf{g}_k,$$

entonces

$$0 = \nabla f(\mathbf{x}_k + \lambda_k \mathbf{d}_k)^T \mathbf{g}_k = \mathbf{g}_{k+1}^T \mathbf{g}_k \qquad \Rightarrow \qquad \mathbf{d}_{k+1}^T \mathbf{d}_k = 0.$$

Esto muestra que dos gradientes son ortogonales entre sí en iteraciones sucesivas y, por lo tanto, dos direcciones sucesivas también son ortogonales, lo que lleva a zigzaguear. El comportamiento de zigzagueo de los iterados del método de Cauchy puede variar si se emplea una búsqueda lineal inexacta, en lugar de la búsqueda lineal exacta (8.6). En el Programa 8.2 se muestra la implementación del método de Cauchy con la condición de Armijo y el proceso de backtracking.

Programa 8.2. Método de Cauchy

```
def cauchy(x0,f,g,tol=1.0e-6,maxit=1000):
    x_k = np.asarray(x0, dtype=float)
    f_k = f(x_k)
    g_k = g(x_k)
    k, ef = 0, 1
    while True:
        if(np.linalg.norm(g_k)<tol):
            print('Terminó satisfactoriamente')
            break
        if (k >= maxit):
            print('Alcanzó el número máxim+o de iteraciones')
            break
        x_k, f_k, fc = backtracking(f,x_k,f_k,g_k,-g_k)
        ef += fc
        g_k = g(x_k)
        k += 1
    return x_k, k, ef
```

Veamos con un ejemplo el comportamiento del método de Cauchy cuando utiliza la condición de Armijo como búsqueda lineal.

Ejemplo 8.3. Consideremos el problema de optimización (8.7) del Ejemplo 8.2. Apliquemos el método de Cauchy al problema (8.7) utilizando la condición de Armijo (8.4), en lugar de la búsqueda lineal exacta (8.6). Tomemos $\gamma = 10^{-4}$ en la condición de Armijo. Consideremos dos iterados iniciales: a) $\mathbf{x}_0 = (4,3)$, y b) $\mathbf{x}_0 = (-4,-3)$. Para emplear la función `cauchy` se procede de la siguiente forma. Primero se define la función f y su gradiente ∇f, que generalmente se define como la función g:

```
>>> def f(x): return x[0]**2/5.0 + x[1]**2
>>> def g(x): return np.array([2.0*x[0]/5.0,2.0*x[1]])
```

Seguidamente se ejecuta la función `cauchy` con cada una de los iterados:

```
>>> xk,k,ef = cauchy([4,3],f,g)
Terminó satisfactoriamente
>>> print(xk,k,ef)
[1.96510151e-06 0.00000000e+00] 29 31
>>> xk,k,ef = cauchy([4,3],f,g)
Terminó satisfactoriamente
>>> print(xk,k,ef)
[1.96510151e-06 0.00000000e+00] 29 31
```

Tanto para el iterado inicial $\mathbf{x}_0 = (4,3)$ coma para $\mathbf{x}_0 = (-4,-3)$, el método de Cauchy obtuvo la solución $\mathbf{x}^* = (1.96510151 \times 10^{-6}, 0)$ empleando 29 iteraciones y 31 evaluaciones de la función objetivo. En la Figura 8.4 se muestra el comportamiento del método de Cauchy. Se puede apreciar que las direcciones consecutivas no son ortogonales.

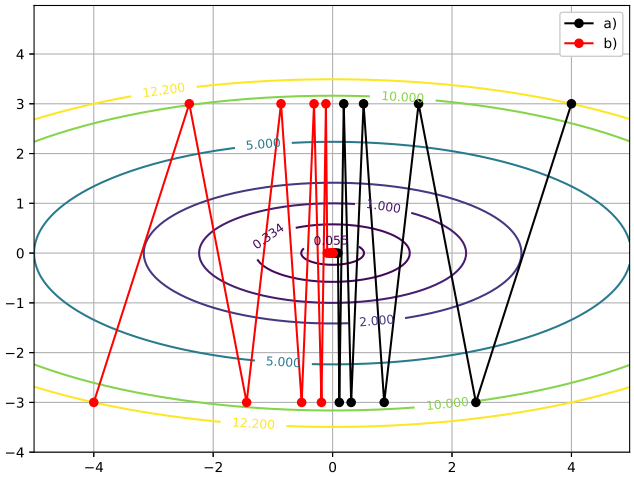


Figura 8.4. Comportamiento del método de Cauchy para el problema (8.7), utilizando la condición de Armijo. a) Comportamiento de Cauchy con el iterado $(4,3)$, y b) comportamiento de Cauchy con el iterado $(-4,-3)$.

Convergencia del Método de Cauchy

El método de Cauchy es importante en el área de la optimización desde el punto de vista teórico, ya que su análisis de convergencia puede ser aplicado a otros métodos de optimización. A continuación, discutimos la convergencia global y la tasa de convergencia local del método de Cauchy. Las características importantes de convergencia local del método de Cauchy se revelan estudiando cuando éste es aplicado a problemas cuadráticos, esto es, cuando la función objetivo del problema (8.1) es cuadrática. Consideremos la función cuadrática definida por

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{x}^T\mathbf{b}, \tag{8.9}$$

donde $A \in \mathbb{R}^{n\times n}$ es una matriz simétrica definida positiva y $\mathbf{b} \in \mathbb{R}^n$ es un vector. Como A es simétrica definida positiva, todos sus autovalores son positivos, además, asumamos que están ordenados en forma creciente

$$0 < \alpha_{min} = \alpha_1 \leq \alpha_2 \leq \cdots \leq \alpha_n = \alpha_{max}.$$

La función f tiene un mínimo global, $\mathbf{x}^* \in \mathbb{R}^n$, que puede encontrarse directamente resolviendo el sistema de ecuaciones lineales

$$A\mathbf{x} = \mathbf{b}.$$

Por otra parte, se tiene que el gradiente de f es

$$\nabla f(\mathbf{x}) = A\mathbf{x} - \mathbf{b}.$$

Así, la forma iterativa del método de Cauchy aplicado a la función (8.9) está dada por

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \lambda_k \mathbf{g}_k, \quad k \geq 0,$$

donde $\mathbf{g}_k = A\mathbf{x}_k - \mathbf{b}$, con

$$\lambda_k = \underset{\lambda>0}{\operatorname{argmin}}\, \varphi(\lambda) = f(\mathbf{x}_k - \lambda\mathbf{g}_k).$$

Considerando la forma de f, se tiene que

$$\begin{aligned}\varphi(\lambda) &= \frac{1}{2}(\mathbf{x}_k - \lambda\mathbf{g}_k)^T A(\mathbf{x}_k - \lambda\mathbf{g}_k) - (\mathbf{x}_k - \lambda\mathbf{g}_k)^T\mathbf{b} \\ &= \frac{1}{2}\mathbf{x}_k^T A\mathbf{x}_k - \lambda\mathbf{g}_k^T A\mathbf{x}_k + \frac{1}{2}\lambda^2\mathbf{g}_k^T A\mathbf{g}_k - \mathbf{x}_k^T\mathbf{b} + \lambda\mathbf{g}_k^T\mathbf{b}.\end{aligned}$$

Ahora, derivando a φ y tomando en cuenta que $\mathbf{g}_k = A\mathbf{x}_k - \mathbf{b}$, podemos escribir:

$$\begin{aligned}\varphi'(\lambda) &= -\mathbf{g}_k^T A\mathbf{x}_k + \lambda\mathbf{g}_k^T A\mathbf{g}_k + \mathbf{g}_k^T\mathbf{b} \\ &= \lambda\mathbf{g}_k^T A\mathbf{g}_k - \mathbf{g}_k^T(A\mathbf{x}_k - \mathbf{b}) \\ &= \lambda\mathbf{g}_k^T A\mathbf{g}_k - \mathbf{g}_k^T\mathbf{g}_k.\end{aligned}$$

Igualando a cero la ecuación anterior y luego despejando λ, obtenemos:

$$\lambda = \frac{\mathbf{g}_k^T\mathbf{g}_k}{\mathbf{g}_k^T A\mathbf{g}_k}.$$

Por tanto, el método de Cauchy para el caso cuadrático adquiere la forma explícita

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \left(\frac{\mathbf{g}_k^T\mathbf{g}_k}{\mathbf{g}_k^T A\mathbf{g}_k}\right)\mathbf{g}_k, \quad k \geq 0. \tag{8.10}$$

En el siguiente teorema se establece la convergencia global del método de Cauchy para el caso cuadrático.

Teorema 8.4. *Para algún iterado inicial* $\mathbf{x}_0 \in \mathbb{R}^n$*, el método de Cauchy* (8.10) *converge al mínimo global* $\mathbf{x}^* \in \mathbb{R}^n$ *de* f *dada por* (8.9)*. Además,*

$$\|\mathbf{x}_{k+1} - \mathbf{x}^*\| \leq \left(\frac{\alpha_{max} - \alpha_{min}}{\alpha_{max} + \alpha_{min}}\right)^2 \|\mathbf{x}_k - \mathbf{x}^*\|, \quad k \geq 0. \tag{8.11}$$

El teorema anterior indica que el método de Cauchy converge linealmente al mínimo global de una función cuadrática. Veamos un ejemplo.

Ejemplo 8.4. Consideremos la función cuadrática $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x}$ definida de $\mathbb{R}^2$ en $\mathbb{R}$, donde

$$A = \begin{bmatrix} 1 & 2 \\ 2 & 8 \end{bmatrix}.$$

Se tiene que el mínimo global de f es $\mathbf{x}^* = (0,0)$ y los autovalores de A son $\alpha_{min} = 5 - \sqrt{13}$ y $\alpha_{max} = 5 + \sqrt{13}$. Así,

$$\left(\frac{\alpha_{max} - \alpha_{min}}{\alpha_{max} + \alpha_{min}}\right)^2 = \left(\frac{\sqrt{13}}{5}\right)^2 = 0.52$$

Esto significa que, según el Teorema 8.4, sólo podemos asegurar que los iterados generados por el método de Cauchy satisfacen que la distancia de $\mathbf{x}_{k+1}$ al mínimo $\mathbf{x}^*$ es alrededor de $1/2$ de la distancia de $\mathbf{x}_k$ a $\mathbf{x}^*$. La Figura 8.5 muestra la gráfica k vs. $e_k = \|\mathbf{x}_k - \mathbf{x}^*\|$ (en escala logarítmica en el eje y).

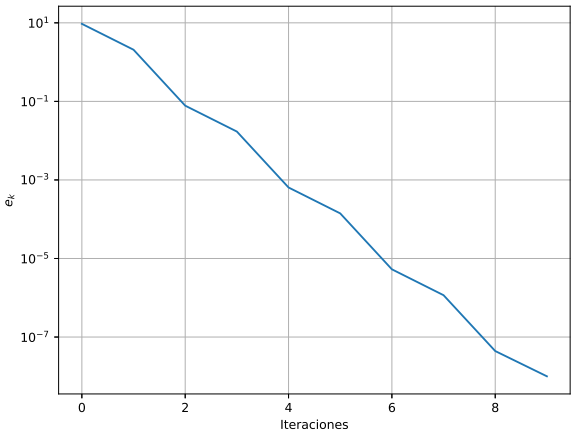


Figura 8.5. Evolución de los errores del método de Cauchy para el Ejemplo 8.4

En la Tabla 8.1 se reportan los iterados y el error e_k para las primeras nueve iteraciones del método de Cauchy, tomando $\mathbf{x}_0 = (5,8)$ como iterado inicial.

Se observa que en cada iteración los iterados se aproximan al mínimo $\mathbf{x}^* = (0,0)$ en a lo sumo una cifra significativa. Además, de la Figura 8.5 se desprende que la disminución de los errores se aproxima a una línea recta, con lo cual se verifica que la convergencia del método de Cauchy es Q-lineal.

Tabla 8.1. Iterados $\mathbf{x}_k$ y el error $e_k = \|\mathbf{x}_k - \mathbf{x}^*\|$ en cada iteración del método de Cauchy.

k	$\mathbf{x}_k$	e_k	e_k/e_{k-1}
0	$(5.000000000, 8.000000000)$	9.433981132	-
1	$(1.973815650, -0.612986227)$	2.066809216	0.219081339
2	$(0.041261954, 0.066019126)$	0.077852899	0.037668159
3	$(0.016288698, -0.005058602)$	0.017056117	0.219081339
4	$(0.000340510, 0.000544816)$	0.000642473	0.037668159
5	$(0.000134421, -0.000041746)$	0.000140754	0.219081339
6	$(0.000002810, 0.000004496)$	0.000005302	0.037668159
7	$(0.000001109, -0.000000345)$	0.000001162	0.219081339
8	$(0.000000023, 0.000000037)$	0.000000044	0.037668159
9	$(0.000000009, -0.000000003)$	0.000000010	0.219081339

Del Teorema 8.5 también se puede inferir que la velocidad de convergencia del método de Cauchy está relacionada con la condición de la matriz A. En efecto, como la condición de A se define por $\kappa_2(A) = \alpha_{max}/\alpha_{min}$, entonces

$$\frac{\alpha_{max} - \alpha_{min}}{\alpha_{max} + \alpha_{min}} = \frac{\kappa_2(A) - 1}{\kappa_2(A) + 1}$$

y la desigualdad (8.11) adquiere la forma

$$\|\mathbf{x}_{k+1} - \mathbf{x}^*\| \leq \left(\frac{\kappa_2(A) - 1}{\kappa_2(A) + 1} \right)^2 \|\mathbf{x}_k - \mathbf{x}^*\|, \quad k \geq 0. \tag{8.12}$$

Esta desigualdad implica que la velocidad de convergencia del método de Cauchy se desmejora a medida que la condición $\kappa_2(A)$ aumenta; por ejemplo, si $\kappa_2(A) = 1000$ entonces (8.12) sugiere que la distancia del iterado $\mathbf{x}_{k+1}$ a $\mathbf{x}^*$ disminuirá en una taza de 0.998 de la distancia de $\mathbf{x}_k$ a $\mathbf{x}^*$, lo cual es una convergencia sumamente lenta.

El comportamiento de la velocidad de convergencia del método de Cauchy es esencialmente el mismo para una función objetivo general. En el siguiente resultado asumimos que el método de Cauchy utiliza una búsqueda lineal exacta.

Teorema 8.5. *Sea $f : \mathbb{R}^n \to \mathbb{R}$ una función dos veces continuamente diferenciable y sea $\mathbf{x}^* \in \mathbb{R}^n$ un mínimo relativo de f, en el cual la matriz Hessiana $\nabla^2 f(\mathbf{x}^*)$ es definida positiva. Sea $\{\mathbf{x}_k\}$ la sucesión generada por el método de Cauchy con búsqueda lineal exacta. Si la sucesión $\{\mathbf{x}_k\}$ converge a $\mathbf{x}^*$, entonces*

$$\|\mathbf{x}_{k+1} - \mathbf{x}^*\| \leq \left(\frac{\alpha_{max} - \alpha_{min}}{\alpha_{max} + \alpha_{min}} \right)^2 \|\mathbf{x}_k - \mathbf{x}^*\|, \quad k \geq 0,$$

donde α_{min} y α_{max} son el menor y el mayor autovalor de $\nabla^2 f(\mathbf{x}^*)$.

En general, no podemos esperar que la velocidad de convergencia del método de Cauchy mejore si se utiliza una búsqueda lineal inexacta (por ejemplo, la de Armijo). En consecuencia, el Teorema 8.5 muestra que el método de Cauchy puede tener una velocidad de convergencia extremadamente lenta, incluso cuando la matriz Hessiana está razonablemente bien condicionada.

Ejemplo 8.5. La eficiencia computacional de un algoritmo se estudia utilizando funciones estándar o funciones test con iterados iniciales estándar $\mathbf{x}_0$. Usualmente se toma como una medida de la

eficiencia del algoritmo, el número total de evaluaciones de la función requeridas para encontrar $\mathbf{x}^*$. Una de las funciones test clásicas es la *función de Rosenbrock*:

$$f(\mathbf{x}) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2, \tag{8.13}$$

cuyo mínimo global es $\mathbf{x}^* = (1,1)$ y se toma $\mathbf{x}_0 = (-1.2, 1)$ como iterado inicial. Se tiene que el gradiente de f es:

$$\nabla f(\mathbf{x}) = \begin{bmatrix} -2\left(200 * x_1(x_1 - x_2^2) + 1 - x_1\right) \\ 200(x_2 - x_1^2) \end{bmatrix}.$$

Apliquemos el método de Cauchy con condición de Armijo a la función de Rosenbrock.

```
>>> def f(x): return 100.0*(x[1]-x[0]*x[0])**2.0+(1-x[0])**2.0
>>> def g(x):
        return np.array([-2.0*(200.0*x[0]*(x[1]-x[0]*x[0])+1.0-x[0]),
                         200.0*(x[1]-x[0]*x[0])])
>>> xk,k,ef,err=cauchy([-1.2,1],f,g,maxit=100000)
Terminó satisfactoriamente
>>> print(xk,k,ef)
[0.99999922  0.99999843] 13756 136800
```

El método de Cauchy requirió de 13756 iteraciones y 136800 evaluaciones de la fun-ción objetivo, para satisfacer el criterio de parada $\|\nabla f(\mathbf{x}_k)\| < 10^{-6}$ y obtener $\mathbf{x} = (0.99999922, 0.99999843)^T$ como aproximación del mínimo $\mathbf{x}^* = (1,1)$.

En la Figura 8.6 se muestra el comportamiento del error del método de Cauchy. Claramente se aprecia que la velocidad de convergencia del método de Cauchy es lineal y sumamente lenta.

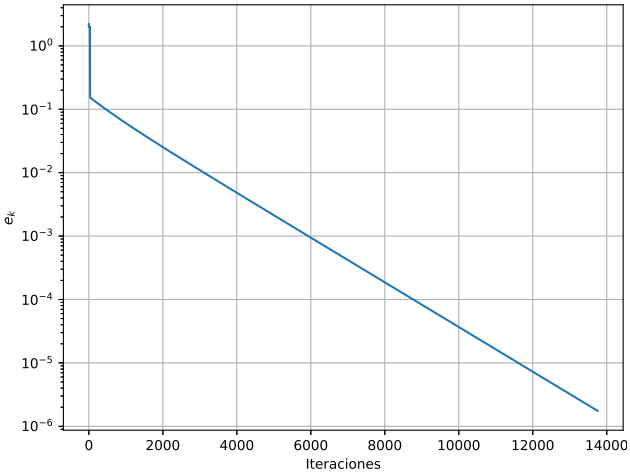


Figura 8.6. Evolución de los errores del método de Cauchy para el Ejemplo 8.5

8.1.4 Método de Newton

La idea detrás del método de Newton es que la función objetivo f se aproxima localmente mediante una función cuadrática, y a esta cuadrática se le halla su mínimo. Por lo tanto, cerca de $\mathbf{x}_k$ podemos aproximar a f usando la serie de Taylor truncada,

$$f(\mathbf{x}) \simeq f(\mathbf{x}_k) + \mathbf{g}_k^T(\mathbf{x}-\mathbf{x}_k) + (\mathbf{x}-\mathbf{x}_k)^T H_k(\mathbf{x}-\mathbf{x}_k),$$

donde $\mathbf{g}_k = \nabla f(\mathbf{x}_k)$ y $H_k = \nabla^2 f(\mathbf{x}_k)$. El mínimo de la función cuadrática del lado derecho de esta ecuación está dado por:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - H_k^{-1}\mathbf{g}_k, \quad k \geq 0 \tag{8.14}$$

que se denomina *forma pura del método de Newton*.

En vista de las condiciones de suficiencia de segundo orden para un punto mínimo (Teorema 8.3), asumiremos que en un punto mínimo relativo, $\mathbf{x}^*$, la matriz Hessiana $\nabla^2 f(\mathbf{x}^*)$ es definida positiva. Entonces, podemos argumentar que si f tiene derivadas parciales continuas, $\nabla^2 f(\mathbf{x})$ es definida positiva cerca de $\mathbf{x}^*$, por lo tanto, el método de Newton está bien definido cerca de la solución. Esto último significa que la dirección de Newton, $\mathbf{d}_k = -H_k^{-1}\mathbf{g}_k$, es una dirección de descenso cerca de la solución.

El método de Newton tiene propiedades muy deseables si se inicia suficientemente cerca del mínimo. Su velocidad de convergencia es cuadrática, lo cual se establece en el siguiente teorema.

Teorema 8.6 (Convergencia del Método de Newton)**.** *Sea $f : \mathbb{R}^n \to \mathbb{R}$ una función dos veces continuamente diferenciable. Sea $\mathbf{x}^* \in \mathbb{R}^n$ un mínimo local de f. Supongamos que la matriz Hessiana $\nabla f(\mathbf{x}^*)$ es definida positiva. Entonces, si tomamos un iterado inicial $\mathbf{x}_0 \in \mathbb{R}^n$ lo suficientemente cerca de $\mathbf{x}^*$, la sucesión generada por el método de Newton* (8.14) *converge $\mathbf{x}^*$ con velocidad de convergencia Q-cuadrática.*

El esquema iterativo del método de Newton se muestra en el Algoritmo 8.4.

Algoritmo 8.4 (Método de Newton)**.**

1: Sean $\mathbf{x}_0 \in \mathbb{R}^n$ un iterado inicial y una tolerancia $\varepsilon > 0$. Tome $k := 0$.
2: **while** $\|\mathbf{g}_k\| > \varepsilon$ **do** ▷ Criterio de parada
3: Resuelva el sistema $H_k\mathbf{d}_k = -\mathbf{g}_k$ ▷ $\mathbf{d}_k$ incógnita del sistema
4: Defina $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{d}_k$ y tome $k := k+1$.
5: **end while**

Aunque el método de Newton es muy atractivo en términos de su velocidad de convergencia cerca de la solución, requiere modificaciones antes de que pueda usarse en puntos que están alejados de la solución. En la literatura existe una gran variedad de modificaciones del método de Newton. En la práctica, la modificación más utilizada consiste en combinar el método de Newton con una búsqueda lineal inexacta, en particular, se combina con la llamada *condición de Wolfe*:

$$f(\mathbf{x}_k + \lambda_k\mathbf{d}_k) \leq f(\mathbf{x}_k) + c_1\lambda_k\nabla f(\mathbf{x}_k)^T\mathbf{d}_k, \tag{8.15}$$

$$\left|\nabla f(\mathbf{x}_k + \lambda_k\mathbf{d}_k)^T\mathbf{d}_k\right| \leq -c_2\nabla f(\mathbf{x}_k)^T\mathbf{d}_k, \tag{8.16}$$

donde $0 < c_1 < c_2 < 1$. La desigualdad (8.15) se denomina *condición débil de Wolfe* y la desigualdad (8.15) se denomina *condición fuerte de Wolfe*. La condición de Wolfe garantiza que la

dirección de Newton $\mathbf{d}_k = -H_k^{-1}\mathbf{g}_k$ sea una dirección de descenso en la mayoría de las iteraciones. Para conocer más detalles de las condiciones de Wolfe se recomiendan los siguientes libros [20], [24], [18], [7], [2].

En el Programa 8.3 se muestra la implementación del método de Newton. El Programa 8.3 utiliza una búsqueda lineal que considera la condición de Wolfe implementada en la función `line_search` del módulo `optimize` de la Librería `Scipy`. Se toma $c_1 = 10^{-4}$ y $c_2 = 0.1$, por defecto. Cuando esta búsqueda lineal no converge, es decir, que no halla una longitud de paso λ_k, la función `newton_min` utiliza la versión pura del método de Newton, esto es, toma $\lambda_k = 1$ y genera un iterado $\mathbf{x}_{k+1}$ dado por la fórmula (8.14).

Programa 8.3. Método de Newton

```
def newton_min(x0,f,g,H,tol=1.0e-6,maxit=1000,ls=True,
               c1=0.0001,c2=0.1):
    x_k = np.asarray(x0, dtype=float)
    f_k = f(x_k)
    g_k = g(x_k)
    H_k = H(x_k)
    k, ef, eg = 0, 1, 1
    while True:
        if(np.linalg.norm(g_k)<tol):
            print('Terminó satisfactoriamente')
            break
        if (k >= maxit):
            print('Alcanzó el número máximo de iteraciones')
            break
        d_k = -np.linalg.solve(H_k,g_k)    #Dirección de Newton
        if ls:
            lam,fc,gc,f_new,f_k,g_k=optimize.line_search\
                (f,g,x_k,d_k,gfk=g_k,old_fval=f_k,c1=c1,c2=c2)
            if lam == None:
                x_k = x_k + d_k
                f_k = f(x_k)
                g_k = g(x_k)
                ef += (fc+1)
                eg += (gc+1)
            else:
                x_k = x_k + lam*d_k
                f_k = f_new
                ef += fc
                eg += gc
        else:
            x_k = x_k + d_k
            f_k = f(x_k)
            g_k = g(x_k)
            ef += 1
            eg += 1
        H_k = H(x_k)
        k += 1
    return x_k, k, ef, eg
```

Ejemplo 8.6. Sea $f:\mathbb{R}^2\to\mathbb{R}$ definida por

$$f(\mathbf{x}) = (x_1-2)^4 + (x_1-2)^2x_2^2 + (x_2+1)^2,$$

cuyo mínimo es $\mathbf{x}^* = (2,-1)$. Veamos el comportamiento del método de Newton para los iterados iniciales: $\mathbf{x}_0^{(1)} = (-2,0.6)$, $\mathbf{x}_0^{(2)} = (0,0)$, $\mathbf{x}_0^{(3)} = (1,1)$ y $\mathbf{x}_0^{(4)} = (4,4)$. Se tiene que el gradiente y la matriz Hessiana de f están dados por:

$$\nabla f(\mathbf{x}) = \begin{bmatrix} 4(x_1-2)^3 + 2(x_1-2)x_2^2 \\ 2(x_1-2)^2x_2 + 2(x_2+1) \end{bmatrix}, \qquad \nabla^2 f(\mathbf{x}) = \begin{bmatrix} 12(x_1-2)^2 + 2x_2^2 & 4x_2(x_1-2) \\ 4x_2(x_1-2) & 2(x_2-1) \end{bmatrix}.$$

Para utilizar la función `newton_min`, primero debemos cargar el módulo `optimize` de la Librería `Scipy`, y definir la función f, el gradiente y la Hessiana.

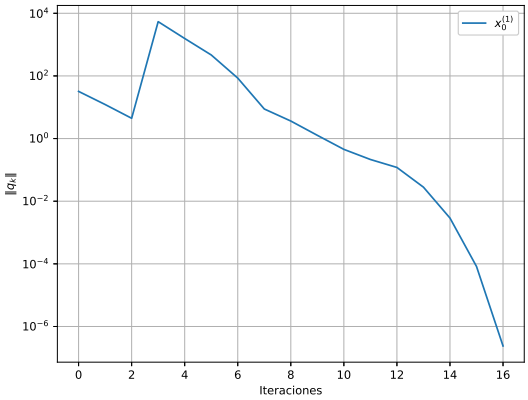
```
>>> from scipy import optimize
>>> def f(x): return (x[0]-2.)**4. + ((x[0]-2.)*x[1])**2. + (x[1]+1.)**2.
>>> def g(x): return np.array([4*(x[0]-2.)**3.+2.*(x[0]-2.)*x[1]**2.,
                               2.*x[1]*(x[0]-2.)+2.*(x[1]+1.)])
>>> def H(x):
        return np.array([[12.*(x[0]-2.)**2.+2.*x[1]**2.,4.*x[1]*(x[0]-2.)],
                         [4.*x[1]*(x[0]-2.),2.*(x[0]-1)]])
```

Luego se ejecuta la función `newton_min` con cada uno de los iterados, escogiendo la opción `ls=False`, con la cual estamos usando Newton en su versión pura, sin búsqueda lineal.

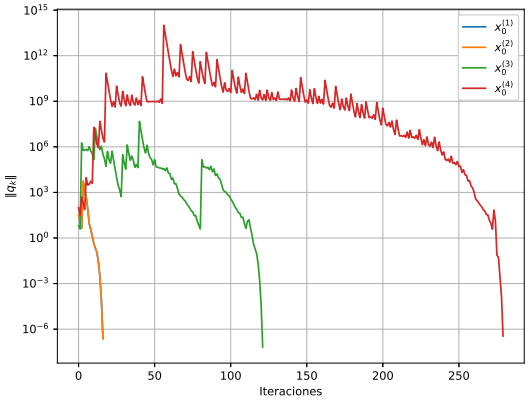
```
>>> xk1, k1, ef1, eg1 = newton_min([-2.,0.6],f,g,H,ls=False)
Terminó satisfactoriamente
>>> print(xk1,k1,ef1,eg1)
[ 2.         -1.00000012] 16 17 17
>>> xk2, k2, ef2, eg2 = newton_min([0,0],f,g,H,ls=False)
Terminó satisfactoriamente
>>> print(xk2,k2,ef2,eg2)
[ 2.         -1.00000012] 16 17 17
>>> xk3, k3, ef3, eg3 = newton_min([1,1],f,g,H,ls=False)
Terminó satisfactoriamente
>>> print(xk3,k3,ef3,eg3)
[ 2.         -1.00000003] 121 122 122
>>> xk4, k4, ef4, eg4 = newton_min([4,4],f,g,H,ls=False)
Terminó satisfactoriamente
>>> print(xk4,k4,ef4,eg4)
[ 2.         -0.99999982] 279 280 280
```

En la Figura 8.7 se describe el comportamiento del método de Newton sin búsqueda lineal. En la Tabla 8.2 se resumen los resultados obtenidos. Aquí se reporta la solución hallada ($\overline{\mathbf{x}}$), el número de iteraciones (it), el número de evaluaciones de la función (ef), el número de evaluaciones del gradiente (eg) y el valor de la función objetivo ($f(\overline{\mathbf{x}})$) en la solución hallada, para cada iterado inicial.

Se observa que el método Newton es muy preciso y su convergencia es rápida cuando el iterado inicial se escoge cerca del mínimo, pero si el iterado inicial está lejos, el esfuerzo computacional del método de Newton aumenta, convirtiéndose prácticamente en un comportamiento errático, llegando en algunas iteraciones a generar iterados que aumentan el valor de la función objetivo. Este hecho se debe a que la dirección de búsqueda de Newton, $\mathbf{d}_k = -H_k^{-1}\mathbf{g}_k$, no es necesariamente una



(a) Iteraciones vs. $\|\mathbf{g}_k\|$ para el iterado inicial $\mathbf{x}_0^{(1)}$.



(b) Iteraciones vs. $\|\mathbf{g}_k\|$ para cada uno de los iterados iniciales.

Figura 8.7. Comportamiento del método de Newton sin búsqueda lineal, para la función f y cada uno de los iterados iniciales del Ejemplo 8.6

Tabla 8.2. Solución, número de iteraciones, número de evaluaciones de la función, número de evaluaciones del gradiente y valor de la función objetivo, para el método de Newton sin búsqueda lineal.

$\mathbf{x}_0$	$\overline{\mathbf{x}}$	it	ef	eg	$f(\overline{\mathbf{x}})$
$\mathbf{x}_0^{(1)}$	$(2.000000000, -1.000000120)$	16	17	17	1.429597e-14
$\mathbf{x}_0^{(2)}$	$(2.000000000, -1.000000120)$	16	17	17	1.429597e-14
$\mathbf{x}_0^{(3)}$	$(2.000000000, -1.000000033)$	121	122	122	1.115987e-15
$\mathbf{x}_0^{(4)}$	$(2.000000000, -0.999999823)$	279	280	280	3.133800e-14

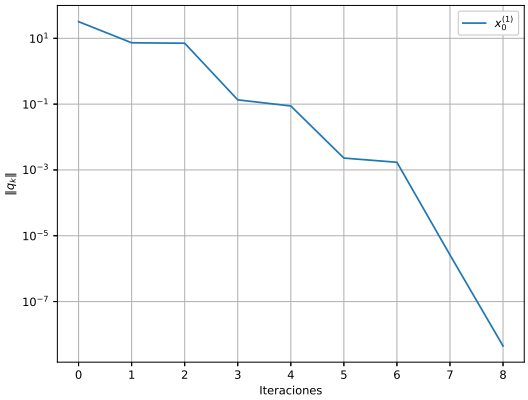
dirección de descenso; por ello, en algunas iteraciones aumenta el valor de la norma del gradiente de f en $\mathbf{x}_k$, $\|\mathbf{g}_k\|$, como se aprecia la Figura 8.7.

Veamos ahora el comportamiento del método de Newton si se combina con una búsque-da lineal. En la Figura 8.8 se describe el comportamiento del método de Newton con búsqueda lineal. En este caso, tomamos `ls=True` en la ejecución de `newton_min`. La Tabla 8.3 muestra los resultados obtenidos cuando `newton_min` utiliza la búsqueda lineal.

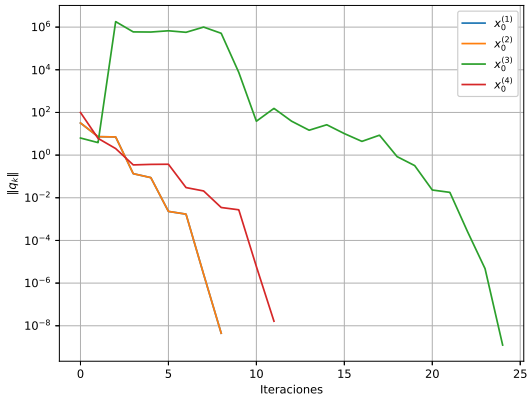
Tabla 8.3. Solución, número de iteraciones, número de evaluaciones de la función, número de evaluaciones del gradiente y valor de la función objetivo, para el método de Newton con búsqueda lineal.

$\mathbf{x}_0$	$\overline{\mathbf{x}}$	it	ef	eg	$f(\overline{\mathbf{x}})$
$\mathbf{x}_0^{(1)}$	$(2.000000000, -0.999999998)$	8	49	19	5.013074e-18
$\mathbf{x}_0^{(2)}$	$(2.000000000, -0.999999998)$	8	49	19	5.013074e-18
$\mathbf{x}_0^{(3)}$	$(2.000000000, -1.000000001)$	24	209	98	3.884227e-19
$\mathbf{x}_0^{(4)}$	$(2.000000000, -0.999999992)$	11	51	29	6.678287e-17

Se observa que el método de Newton con búsqueda lineal es más eficiente que su forma pura;



(a) Iteraciones vs. $\|\mathbf{g}_k\|$ para el iterado inicial $\mathbf{x}_0^{(1)}$.



(b) Iteraciones vs. $\|\mathbf{g}_k\|$ para cada uno de los iterados iniciales.

Figura 8.8. Comportamiento del método de Newton con búsqueda lineal, para la función f y cada uno de los iterados iniciales del Ejemplo 8.6

por supuesto, con esta función y los iterados iniciales escogidos. Pudiera ocurrir que con otras funciones u otros iterados iniciales, la forma pura del método de Newton sea más eficiente que su modificación. Se deja como ejercicio para el lector encontrar una función y un iterado inicial tales que la forma pura sea más eficiente que Newton con búsqueda lineal.

En esencia, el método de Newton para minimización no es más que el método de Newton para sistemas de ecuaciones no lineales aplicado a la ecuación $\nabla f(x) = 0$, es decir, el método de Newton halla puntos estacionarios de la función f. En consecuencia, todas las propiedades del método de Newton para sistemas no lineales se pueden extender para la minimización de funciones.

8.1.5 Métodos de Gradientes Conjugados

El método de Cauchy y el método de Newton poseen características muy variadas en cuanto al rendimiento computacional. El método de Cauchy tiene convergencia lenta, pero requiere poco espacio de almacenamiento en memoria, solo almacena dos vectores de $\mathbb{R}^n$. En cambio, el método de Newton es un algoritmo rápido, pero requiere un gran espacio de almacenamiento en memoria, necesita almacenar una matriz de orden $n \times n$ y dos vectores de $\mathbb{R}^n$.

Existe un grupo de algoritmos de poco almacenamiento, cuya velocidad de convergencia es superior a la del método de Cauchy, pero inferior a la del método de Newton. Tales algoritmos se basan en el concepto de direcciones conjugadas.

Definición 8.5. Sea $A \in \mathbb{R}^{n\times n}$ una matriz. Se dice que los vectores $\mathbf{v}, \mathbf{w} \in \mathbb{R}^n$ son *A-conjugados* si $\mathbf{v}^T A\mathbf{w} = 0$.

En el siguiente teorema se establece que un conjunto de vectores A-conjugados son linealmente independientes, cuando A es una matriz simétrica definida positiva.

Teorema 8.7. *Sea $A \in \mathbb{R}^{n\times n}$ una matriz. Sean $\mathbf{d}_0, \mathbf{d}_1, \ldots, \mathbf{d}_{k-1} \in \mathbb{R}^n$ k vectores A-conjugados normalizados, esto es:*

i) $\mathbf{d}_i^T A\mathbf{d}_i = 1$, $i = 0, 1, \ldots, k-1$;

ii) $\mathbf{d}_i^T A\mathbf{d}_j = 0$, $i \neq j$.

Si la matriz A es simétrica definida positiva (definida negativa), entonces los vectores $\mathbf{d}_0, \mathbf{d}_1, \ldots, \mathbf{d}_{k-1}$ son linealmente independientes.

Antes de presentar el algoritmo general de gradientes conjugados, discutiremos inicialmente el caso cuadrático, esto es, consideraremos una función objetivo dada por (8.9), donde $A \in \mathbb{R}^{n\times n}$ es una matriz simétrica definida positiva. Recordemos que la única solución $\mathbf{x}^*$ del problema de minimización cuadrática también es la única solución del sistema $A\mathbf{x} = \mathbf{b}$ y, por lo tanto, el problema de minimización cuadrática es equivalente a un sistema de ecuaciones lineales.

Sean $\mathbf{d}_0, \mathbf{d}_1, \ldots, \mathbf{d}_{n-1} \in \mathbb{R}^n$ n vectores A-conjugados normalizados. Por el Teorema 8.7, todos estos vectores son linealmente independientes, lo cual implica que el mínimo global $\mathbf{x}^*$ puede expresarse como una combinación lineal de estos vectores, es decir,

$$\mathbf{x}^* = \alpha_0\mathbf{d}_0 + \alpha_1\mathbf{d}_1 + \cdots + \alpha_{n-1}\mathbf{d}_{n-1}, \tag{8.17}$$

para un conjunto de escalares α_i. En efecto, multiplicando por A la ecuación (8.17) y tomando el producto escalar por $\mathbf{d}_i$, obtenemos:

$$\mathbf{d}_i^T A\mathbf{x}^* = \alpha_0\mathbf{d}_i^T A\mathbf{d}_0 + \alpha_1\mathbf{d}_i^T A\mathbf{d}_1 + \cdots + \alpha_{n-1}\mathbf{d}_i^T A\mathbf{d}_{n-1} = \alpha_i\mathbf{d}_i^T A\mathbf{d}_i.$$

Como $A\mathbf{x}^* = \mathbf{b}$, de la ecuación anterior se tiene

$$\alpha_i = \frac{\mathbf{d}_i^T\mathbf{b}}{\mathbf{d}_i^T A\mathbf{d}_i}, \quad i = 0, 1, \ldots, n-1.$$

Esto muestra que los α_i y en consecuencia el mínimo $\mathbf{x}^*$, pueden encontrase evaluando simplemente productos escalares. En sí, el resultado final es

$$\mathbf{x}^* = \sum_{i=0}^{n-1} \frac{\mathbf{d}_i^T\mathbf{b}}{\mathbf{d}_i^T A\mathbf{d}_i}\,\mathbf{d}_i.$$

Hay dos ideas básicas implícitas en la ecuación anterior. La primera es la idea de seleccionar un conjunto de vectores conjugados $\mathbf{d}_i$ para que al tomar un producto escalar apropiado, todos los términos en el lado derecho de (8.17), excepto el i-ésimo, desaparezcan. Esto podría, por supuesto, haberse logrado haciendo que el vector $\mathbf{d}_i$ sea ortogonal en el sentido ordinario, en lugar de hacerlos A-conjugados. La segunda observación básica, sin embargo, es que al usar la A-conjugación, la ecuación resultante para α_i se puede expresar en términos del vector conocido $\mathbf{b}$ en lugar del vector desconocido $\mathbf{x}^*$; por lo tanto, los coeficientes se pueden evaluar sin conocer $\mathbf{x}^*$.

La expansión de $\mathbf{x}^*$ puede ser considerada como un proceso iterativo de n pasos, donde $\alpha_i\mathbf{d}_i$ es sumado en la i-ésima iteración. Viendo el procedimiento de esta manera, y permitiendo un punto inicial arbitrario, se obtiene el *Método de Gradiente Conjugado* o método CG (por sus siglas en inglés, Conjugate Gradient), que es uno de los métodos más conocidos para resolver sistemas de ecuaciones lineales simétricos definidos positivos y dispersos, cuyo esquema iterativo se muestra en el Algoritmo 8.5. La implementación del método CG se muestra en el Programa 8.4.

Algoritmo 8.5 (Método de Gradiente Conjugado)**.**

1: Sean $\mathbf{x}_0 \in \mathbb{R}^n$ un iterado inicial. Tome $k := 0$.
2: Calcule: $\mathbf{g}_k = \mathbf{b} - A\mathbf{x}_k$, $\mathbf{d}_k = \mathbf{g}_k$
3: **for** $k = 1, 2, \ldots$ hasta convergencia **do**
4: $\quad \alpha_k = (\mathbf{g}_k^T \mathbf{g}_k)/(\mathbf{d}_k^T A \mathbf{d}_k)$
5: $\quad \mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k$
6: $\quad \mathbf{g}_{k+1} = \mathbf{g}_k - \alpha_k A \mathbf{d}_k$
7: $\quad \beta_k = (\mathbf{g}_{k+1}^T \mathbf{g}_{k+1})/(\mathbf{g}_k^T \mathbf{g}_k)$
8: $\quad \mathbf{d}_{k+1} = \mathbf{g}_{k+1} + \beta_k \mathbf{d}_{k+1}$
9: **end for**

Programa 8.4. Método de Gradiente Conjugado

```
def cgm(x0,A,b,tol=1.0e-15):
    A = np.asarray(A, dtype=float)
    b = np.asarray(b, dtype=float)
    x_k = np.asarray(x0, dtype=float)
    n = len(b)
    g_k = b-A.dot(x_k)
    gg = g_k.dot(g_k)
    d_k = np.asarray(g_k,dtype=float)
    k = 0
    while k<=n:
        if(np.sqrt(gg)<=tol):
            print('Terminó satisfactoriamente')
            break
        Adk = A.dot(d_k)
        alpha_k = gg/(d_k.dot(Adk))
        x_k = x_k + alpha_k*d_k
        g_k = g_k - alpha_k*Adk
        gg_1 = g_k.dot(g_k)
        beta_k = gg_1 / gg
        d_k = g_k + beta_k*d_k
        gg = gg_1
        k += 1
    return x_k, k
```

El método CG es un método de direcciones conjugadas con una propiedad muy especial: para generar su conjunto de vectores conjugados, calcula un nuevo vector $\mathbf{d}_k$ utilizando solo el vector anterior $\mathbf{d}_{k-1}$. No necesita conocer todos los elementos previos $\mathbf{d}_0, \mathbf{d}_1, \ldots, \mathbf{d}_{k-2}$; el vector $\mathbf{d}_k$ se conjuga automáticamente con estos vectores. Esta notable propiedad implica que el método requiere poco almacenamiento y cálculo. En el siguiente teorema se establece la convergencia del método CG.

Teorema 8.8. *Sea $A \in \mathbb{R}^{n\times n}$ una matriz simétrica definida positiva. Para algún $\mathbf{x}_0 \in \mathbb{R}^n$, la sucesión $\{\mathbf{x}_k\}$ generada por el Algoritmo 8.5 converge al mínimo global $\mathbf{x}^*$ de la cuadrática f dada por* (8.9)*, a lo sumo en n iteraciones.*

El Teorema 8.8 también nos dice que el método CG converge a la solución del sistema de ecuaciones lineales $A\mathbf{x} = b$, a lo sumo en n iteraciones. Veamos un ejemplo de ello.

Ejemplo 8.7. Consideremos una matriz $A \in \mathbb{R}^{3\times 3}$ dada por

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}.$$

y el vector $\mathbf{b} = (1,1,1)$. Empleemos el método CG para resolver el sistema lineal $A\mathbf{x} = \mathbf{b}$, cuya solución es $\mathbf{x}^* = \left(1, \frac{1}{2}, \frac{1}{3}\right)$. Tomemos $\mathbf{x}_0 = (0,0,0)$ como iterado inicial. Se tiene

$$\mathbf{x}_0 = (0,0,0), \quad \mathbf{g}_0 = \mathbf{b} - A\mathbf{x}_0 = (1,1,1), \quad \mathbf{d}_0 = (1,1,1).$$

Así, tenemos para cada $k \geq 0$:

- $k = 0$

$$\alpha_0 = \frac{\mathbf{g}_0^T \mathbf{g}_0}{\mathbf{d}_0^T A \mathbf{d}_0} = \frac{1}{2}, \quad \mathbf{x}_1 = \mathbf{x}_0 + \alpha_0 \mathbf{d}_0 = \left(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}\right), \quad \mathbf{g}_1 = \mathbf{g}_0 - \alpha_0 A\mathbf{d}_0 = \left(\frac{1}{2}, 0, -\frac{1}{2}\right)$$
$$\beta_0 = \frac{\mathbf{g}_1^T \mathbf{g}_1}{\mathbf{g}_0^T \mathbf{g}_0} = \frac{1}{6}, \quad \mathbf{d}_1 = \mathbf{g}_1 + \beta_0 \mathbf{d}_0 = \left(\frac{2}{3}, \frac{1}{6}, -\frac{1}{3}\right)$$

- $k = 1$

$$\alpha_1 = \frac{\mathbf{g}_1^T \mathbf{g}_1}{\mathbf{d}_1^T A \mathbf{d}_1} = \frac{3}{5}, \ \mathbf{x}_2 = \mathbf{x}_1 + \alpha_1 \mathbf{d}_1 = \left(\frac{9}{10}, \frac{3}{5}, \frac{3}{10}\right), \ \mathbf{g}_2 = \mathbf{g}_1 - \alpha_1 A\mathbf{d}_1 = \left(\frac{1}{10}, -\frac{1}{5}, \frac{1}{10}\right)$$
$$\beta_1 = \frac{\mathbf{g}_2^T \mathbf{g}_2}{\mathbf{g}_1^T \mathbf{g}_1} = \frac{3}{25}, \quad \mathbf{d}_2 = \mathbf{g}_2 + \beta_1 \mathbf{d}_1 = \left(\frac{9}{50}, -\frac{9}{50}, \frac{3}{50}\right)$$

- $k = 2$

$$\alpha_2 = \frac{\mathbf{g}_2^T \mathbf{g}_2}{\mathbf{d}_2^T A \mathbf{d}_2} = \frac{5}{9}, \quad \mathbf{x}_3 = \mathbf{x}_2 + \alpha_2 \mathbf{d}_2 = \left(1, \frac{1}{2}, \frac{1}{3}\right), \quad \mathbf{g}_3 = \mathbf{g}_2 - \alpha_2 A\mathbf{d}_2 = (0,0,0)$$
$$\beta_2 = \frac{\mathbf{g}_3^T \mathbf{g}_3}{\mathbf{g}_2^T \mathbf{g}_2} = 0, \quad \mathbf{d}_3 = \mathbf{g}_3 + \beta_2 \mathbf{d}_2 = (0,0,0).$$

De esta forma, el método CG halló la solución $\mathbf{x}^*$ en 3 iteraciones.

Hemos visto que en aritmética exacta el método CG terminará en la solución a lo sumo en n iteraciones. Lo que es más notable es que cuando la distribución de los autovalores de A tiene ciertas características favorables, el algoritmo identificará la solución en mucho menos de n iteraciones. El siguiente teorema nos dice que si la matriz A tiene m autovalores distintos, entonces el método CG termina en la solución $\mathbf{x}^*$ a lo sumo en m iteraciones.

Teorema 8.9. *Sea $A \in \mathbb{R}^{n\times n}$ una matriz simétrica definida positiva. Si la matriz A tiene m autovalores distintos, entonces para algún $\mathbf{x}_0 \in \mathbb{R}^n$, la sucesión $\{\mathbf{x}_k\}$ generada por el Algoritmo 8.5 converge al mínimo global $\mathbf{x}^*$ de la cuadrática f dada por* (8.9)*, a lo sumo en m iteraciones.*

Hemos mostrado que el método CG puede verse como un algoritmo de minimización para una función cuadrática f (8.9), donde $A \in \mathbb{R}^{n\times n}$ es una matriz simétrica definida positiva. Es natural

preguntar si podemos adaptar el enfoque para minimizar funciones no lineales generales f. De hecho, como describiremos más adelante, las variantes no lineales del gradiente conjugado están bien estudiadas y han demostrado ser bastante exitosas en la práctica. Las implementaciones modernas de los métodos de gradientes conjugados para el problema de minimización sin restricciones (8.1), generan una sucesión de puntos $\{\mathbf{x}_k\}$ de la forma

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \lambda_k \mathbf{d}_k, \quad k \geq 0$$

donde $\lambda_k > 0$ es la longitud de paso obtenida usando una búsqueda lineal y $\mathbf{d}_k$ es la dirección de búsqueda definida por

$$\mathbf{d}_k = \begin{cases} -\mathbf{g}_k, & \text{para } k = 0, \\ -\mathbf{g}_k + \beta_k \mathbf{d}_{k-1}, & \text{para } k \geq 2. \end{cases}$$

Aquí, β_k se denomina *parámetro de gradiente conjugado*. En el Algoritmo 8.6 se muestra el esquema general de los métodos gradientes conjugados para el problema de minimización (8.1).

Algoritmo 8.6 (Métodos de Gradientes Conjugados)**.**

1: Sean $\mathbf{x}_0 \in \mathbb{R}^n$ un iterado inicial y una tolerancia $\varepsilon > 0$.
2: Tome $k := 0$ y $\mathbf{d}_0 = -\mathbf{g}_0$;
3: **while** $\|\mathbf{g}_k\| > \varepsilon$ **do**
4: Calcule la longitud de paso λ_k usando una búsqueda lineal;
5: Tome $\mathbf{x}_{k+1} = \mathbf{x}_k + \lambda_k \mathbf{d}_k$ y calcule β_k;
6: Calcule

$$\mathbf{d}_k = \begin{cases} -\mathbf{g}_k, & \text{para } k = 0, \\ -\mathbf{g}_k + \beta_k \mathbf{d}_{k-1}, & \text{para } k \geq 2. \end{cases}$$

7: Tome $k = k + 1$
8: **end while**

Note que cuando f es una cuadrática dada por (8.9), con A simétrica definida positiva, y se emplea una búsqueda lineal exacta, entonces el parámetro β_k adquiere la forma

$$\beta_k = \frac{\mathbf{g}_{k+1}^T \mathbf{g}_{k+1}}{\mathbf{g}_k^T \mathbf{g}_k}$$

y el Algoritmo 8.6 es el método CG. Así, dependiendo de la búsqueda lineal empleada y la manera de como se calcule el parámetro β_k, se obtiene un tipo de método de gradiente conjugado. En la literatura existen muchas variantes del parámetro β_k. A continuación listamos las más usadas.

a) Fletcher y Reeves [8]

$$\beta_k^{FR} = \frac{\mathbf{g}_{k+1}^T \mathbf{g}_{k+1}}{\mathbf{g}_k^T \mathbf{g}_k}$$

b) Polak y Ribière [23]

$$\beta_k^{PR} = \frac{\mathbf{g}_{k+1}^T (\mathbf{g}_{k+1} - \mathbf{g}_k)}{\mathbf{g}_k^T \mathbf{g}_k}$$

Para la implementación del Algoritmo 8.6 es necesario tener precisión de la búsqueda lineal a emplear, ya que la dirección $\mathbf{d}_k$ pudiera no ser de descenso. En efecto, tomando el producto interno de $\mathbf{d}_k$ con el gradiente $\mathbf{g}_k$, obtenemos

$$\mathbf{g}_k^T \mathbf{d}_k = -\|\mathbf{g}_k\|^2 + \beta_k \mathbf{g}_k^T \mathbf{d}_{k-1}, \quad \text{para } k \geq 1. \tag{8.18}$$

Si la búsqueda lineal es exacta, entonces la longitud de paso es un mínimo de f a lo largo de $\mathbf{d}_{k-1}$, luego $\mathbf{g}_k^T \mathbf{d}_{k-1} = 0$. Entonces, por (8.18) $\mathbf{g}_k^T \mathbf{d}_k < 0$, es decir, $\mathbf{d}_k$ es una dirección de descenso. Si la búsqueda lineal no es exacta, sin embargo, el segundo término de (8.18) puede dominar al primer término y se puede tener que $\mathbf{g}_k^T \mathbf{d}_k > 0$, implicando que $\mathbf{d}_k$ sea una dirección de ascenso. Afortunadamente, podemos evitar esta situación exigiendo que la longitud del paso λ_k satisfaga las condiciones de Wolfe. Escogiendo λ_k de manera que satisfaga (8.15) y (8.16), se asegura que todas las direcciones $\mathbf{d}_k$ son direcciones de descenso para la función f. En la práctica, los parámetros c_1 y c_2 de las condiciones débil y fuerte de Wolfe se escogen como $c_1 = 10^{-4}$ y $c_2 = 0.1$.

En la función `fmin_cg` del módulo `optimize` de la librería `scipy`, está implementado el método de gradiente conjugado de Polak-Ribière (CGPR), esto es, el Algoritmo 8.6 tomando β_k^{PR} como el parámetro de gradiente conjugado, con las condiciones de Wolfe (8.15) y (8.16). Con la instrucción:

```
scipy.optimize.fmin_cg(f,x0,fprime=None,args=(),gtol=1e-05,maxiter=None,
full_output=0,disp=1,retall=0)
```

se ejecuta la función `fmin_cg`. Para más detalles visite el sitio web

https://docs.scipy.org/doc/

Mostremos con un ejemplo la manera de emplear la función `fmin_cg`.

Ejemplo 8.8. Consideremos la función f del Ejemplo 8.6, cuyo mínimo es $\mathbf{x}^* = (2,-1)^T$; además, tomemos los mismos iterados iniciales dados en este ejemplo. Para ejecutar `fmin_cg`, primero debemos cargar el módulo `optimize` y definir la función f y el gradiente.

```
>>> from scipy import optimize
>>> def f(x): return (x[0]-2.)**4. + ((x[0]-2.)*x[1])**2. + (x[1]+1.)**2.
>>> def g(x): return np.array([4*(x[0]-2.)**3.+2.*(x[0]-2.)*x[1]**2.,
                               2.*x[1]*(x[0]-2.)+2.*(x[1]+1.)])
```

Para cada iterado, ejecutamos la función `fmin_cg`.

```
>>> xk1,iterados1 = optimize.fmin_cg(f,[-2.,0.6],fprime=g,gtol=1e-6,
                                     maxiter=10000,retall=1)
Warning: Desired error not necessarily achieved due to precision loss.
         Current function value: 0.002851
         Iterations: 6
         Function evaluations: 52
         Gradient evaluations: 40
>>> print(xk1)
[ 1.94698716 -1.00356385]
>>> xk2,iterados2 = optimize.fmin_cg(f,[0,0],fprime=g,gtol=1e-6,
                                     maxiter=10000,retall=1)
Warning: Desired error not necessarily achieved due to precision loss.
         Current function value: 0.000090
         Iterations: 5
         Function evaluations: 116
         Gradient evaluations: 104
>>> print(xk2)
[ 2.00826215 -1.0046117 ]
>>> xk3,iterados3 = optimize.fmin_cg(f,[1,1],fprime=g,gtol=1e-6,
                                     maxiter=10000,retall=1)
Warning: Desired error not necessarily achieved due to precision loss.
         Current function value: 0.000012
         Iterations: 4
         Function evaluations: 112
         Gradient evaluations: 100
```

```
>>> print(xk3)
[ 1.99666742 -1.00116102]
>>> xk4,iterados4 = optimize.fmin_cg(f,[4,4],fprime=g,gtol=1e-6,
                                     maxiter=10000,retall=1)
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 10
         Function evaluations: 23
         Gradient evaluations: 23
>>> print(xk4)
[ 2.00000006 -1.00000002]
```

Observe que la salida de la función `fmin_cg` es la solución y una lista de arreglos construida con los iterados $\mathbf{x}_k$ obtenidos en la k-ésima iteración; por ejemplo, para el iterado inicial $\mathbf{x}_0^{(1)}$, `xk1` es la solución obtenida e `iterados1` es la lista de iterados. En la Figura 8.9 se muestra la gráfica de $\|\mathbf{g}_k\|$ vs. iteraciones, para cada iterado inicial. En la Tabla 8.4 se resumen los resultados obtenidos.

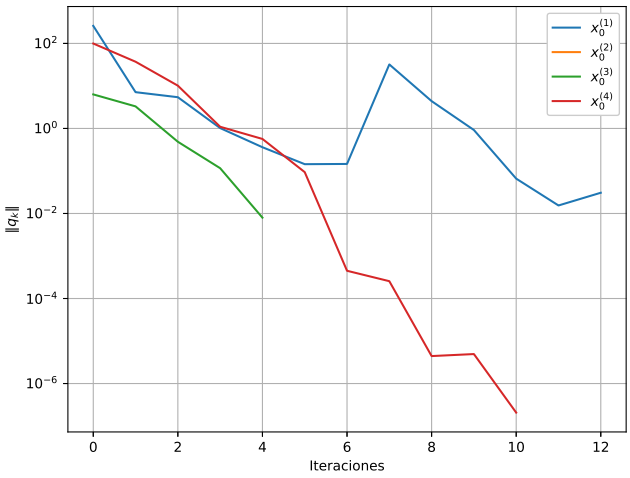


Figura 8.9. Iteraciones vs. $\|\mathbf{g}_k\|$ del método CGPR para la función f y cada uno de los iterados iniciales del Ejemplo 8.6

Tabla 8.4. Solución, número de iteraciones, número de evaluaciones de la función, número de evaluaciones del gradiente y valor de la función objetivo, para el método CGPR.

$\mathbf{x}_0$	$\overline{\mathbf{x}}$	it	ef	eg	$f(\overline{\mathbf{x}})$
$\mathbf{x}_0^{(1)}$	$(1.94698716, -1.00356385)$	6	52	40	0.002851
$\mathbf{x}_0^{(2)}$	$(2.00826215, -1.0046117)$	5	116	104	0.000090
$\mathbf{x}_0^{(3)}$	$(1.99666742, -1.00116102)$	4	112	100	0.000012
$\mathbf{x}_0^{(4)}$	$(2.00000006, -1.00000002)$	10	23	23	0.000000

Se observa que el método CGPR no es tan rápido como el método de Newton cuando el iterado inicial está cerca del mínimo (ver el Ejemplo 8.6); más aún, el método falló con los tres primeros iterados iniciales, sólo convergió con el último iterado inicial.

8.1.6 Métodos Quasi-Newton

En muchas situaciones prácticas, la evaluación y el uso de la matriz Hessiana no está disponible o su costo computacional es muy alto. La idea principal de los métodos quasi-Newton es utilizar una aproximación de la inversa de la Hessiana en lugar de la verdadera inversa que se requiere en el método de Newton, con el propósito de disminuir el costo computacional. La forma de esta aproximación varía entre diferentes métodos, que van desde el más simple donde se toma fija a lo largo del proceso iterativo, hasta el más avanzado donde las aproximaciones mejoradas se construyen sobre la base de la información recopilada en cada iteración.

Los métodos quasi-Newton generan una sucesión de puntos $\{\mathbf{x}_k\}$ a través de la fórmula general de los métodos de búsqueda local, a saber,

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \lambda_k \mathbf{d}_k, \quad k \geq 0$$

donde λ_k se escoge de manera que satisfaga la condición de Wolfe (desigualdades (8.15) y (8.16)) y la dirección $\mathbf{d}_k$ se calcula por

$$\mathbf{d}_k = -B_k^{-1}\mathbf{g}_k, \tag{8.19}$$

con $B_k \in \mathbb{R}^{n\times n}$ una matriz que aproxima a la matriz Hessiana $\nabla^2 f(\mathbf{x}_k)$. Para garantizar que la dirección $\mathbf{d}_k$ sea una dirección de descenso, los métodos quasi-Newton más utilizados construyen la matriz B_k de manera que sea simétrica definida positiva, garantizando con ello que $\mathbf{d}_k$ sea una dirección de descenso para la función objetivo.

Existen muchos métodos quasi-Newton, pero el más popular por su buen comportamiento práctico es el *método BFGS*, llamado así por sus descubridores Broyden, Fletcher, Goldfarb y Shanno. Dada una matriz B_0 simétrica definida positiva que aproxima a la Hessiana $\nabla^2 f(\mathbf{x}_0)$, el objetivo del método BFGS es generar una sucesión de matrices $\{B_k\}$ simétricas definidas positivas de manera que satisfagan la *ecuación de la secante*,

$$B_{k+1}\mathbf{s}_k = \mathbf{y}_k, \quad k \geq 0,$$

donde $\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$ e $\mathbf{y}_k = \mathbf{g}_{k+1} - \mathbf{g}_k$. Una variante para generar las matrices B_k del método BFGS está dada por la fórmula

$$B_{k+1} = B_k - \frac{B_k\mathbf{s}_k\mathbf{s}_k^T B_k}{\mathbf{s}_k^T B_k \mathbf{s}_k} + \frac{\mathbf{y}_k\mathbf{y}_k^T}{\mathbf{y}_k\mathbf{s}_k^T}, \tag{8.20}$$

denominada *fórmula de actualización*. La implementación de esta variante no es eficiente para la minimización sin restricciones, porque requiere resolver el sistema $B_k\mathbf{d}_k = -\mathbf{g}_k$ para obtener la dirección $\mathbf{d}_k$, aumentando de ese modo el costo del cálculo del paso a $\mathbf{O}(n^3)$. Existe una variante del método BFGS (normalmente se denomina método BFGS) que no genera directamente la sucesión $\{B_k\}$, sino que genera la sucesión de inversas $\{B_k^{-1}\}$ que es útil en la implementación del método, ya que permite que la dirección de búsqueda (8.19) sea calculada mediante una simple multiplicación matriz-vector. Denotando $H_k = B_k^{-1}$, el método BFGS genera una sucesión de matrices $\{H_k\}$ por la fórmula

$$H_{k+1} = \left(I - \frac{\mathbf{s}_k\mathbf{y}_k^T}{\mathbf{y}_k^T\mathbf{s}_k}\right) H_k \left(I - \frac{\mathbf{s}_k\mathbf{y}_k^T}{\mathbf{y}_k^T\mathbf{s}_k}\right) + \frac{\mathbf{s}_k\mathbf{y}_k^T}{\mathbf{y}_k^T\mathbf{s}_k}, \tag{8.21}$$

donde la matriz I es la matriz identidad de orden $n\times n$. En el Algoritmo 8.7 se muestra el esquema iterativo del método BFGS.

Algoritmo 8.7 (Método BFGS)**.**

1: Sean $\mathbf{x}_0 \in \mathbb{R}^n$ un iterado inicial, una tolerancia $\varepsilon > 0$ y $H_0 \in \mathbb{R}^{n\times n}$ la aproximación de la inversa de la matriz Hessiana en $\mathbf{x}_0$.
2: Tome $k = 0$;
3: **while** $\|\mathbf{g}_k\| > \varepsilon$ **do**
4: Calcule la dirección de búsqueda $\mathbf{d}_k = -H_k\mathbf{g}_k$;
5: Calcule λ_k tal que satisfaga la condición de Wolfe (8.15) y (8.16);
6: Tome $\mathbf{x}_{k+1} = \mathbf{x}_k + \lambda_k\mathbf{d}_k$, $\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$ e $\mathbf{y}_k = \mathbf{g}_{k+1} - \mathbf{g}_k$;
7: Calcule H_k usando la fórmula de actualización BFGS (8.21);
8: Tome $k = k+1$;
9: **end while**

El método BFGS es más rápido que el método de Cauchy, pero es más lento que el método de Newton. Bajo ciertas condiciones (función objetivo dos veces continuamente diferenciable, entre otras condiciones particulares, para más detalles ver [20] y referencias dadas allí), se demuestra que el método BFGS converge superlinealmente al mínimo.

Ahora bien, solo se debe resolver un problema antes de implementar el método BFGS: ¿cómo se debe elegir la aproximación inicial H_0? Desafortunadamente, no existe una fórmula mágica que funcione bien en todos los casos. Podemos usar información específica sobre el problema, por ejemplo, definiendo H_0 como la aproximación por diferencias finitas de la inversa de la Hessiana $\nabla^2 f(\mathbf{x}_0)$. En otros casos, podemos simplemente elegir que sea la matriz identidad, o un múltiplo de la matriz de identidad, donde se toma el múltiplo para reflejar la escala de las variables. En la literatura existen distintas estrategias para escoger H_0, unas más eficientes que otras, pero todo depende del problema a resolver (ver [20] para más detalles). En la función `fmin_bfgs` del módulo `optimize` de la librería `scipy`, está implementado el método BFGS con la ventaja de que el usuario no debe suministrar una aproximación inicial de H_0, ya que se genera utilizando una heurística dada en [20] (para detalles visite el sitio web https://docs.scipy.org/doc/). La función `fmin_bfgs` se ejecuta a través de la instrucción

```
scipy.optimize.fmin_bfgs(f,x0,fprime=None,args=(),gtol=1e-05,maxiter=None,
full_output=0,disp=1,retall=0)
```

Ejemplo 8.9. Consideremos la función f del Ejemplo 8.6, cuyo mínimo es $\mathbf{x}^* = (2,-1)$; además, tomemos los mismos iterados iniciales dados en este ejemplo. `fmin_bfgs` se emplea de manera semejante a la función `fmin_cg`; primero debemos cargar el módulo `optimize` y luego definir la función f y su gradiente.

```
>>> from scipy import optimize
>>> def f(x): return (x[0]-2.)**4. + ((x[0]-2.)*x[1])**2. + (x[1]+1.)**2.
>>> def g(x): return np.array([4*(x[0]-2.)**3.+2.*(x[0]-2.)*x[1]**2.,
                               2.*x[1]*(x[0]-2.)+2.*(x[1]+1.)])
```

Para cada iterado, ejecutamos la función `fmin_bfgs`.

```
>>> xk1,iterados1 = optimize.fmin_bfgs(f,[-2.,0.6],fprime=g,gtol=1e-6,
                                       maxiter=10000,retall=1)
Warning: Desired error not necessarily achieved due to precision loss.
         Current function value: 27.122039
         Iterations: 2
         Function evaluations: 44
         Gradient evaluations: 33
```

```
>>> print(xk1)
[-0.25050838  0.15865314]
>>> xk2,iterados2 = optimize.fmin_bfgs(f,[0,0],fprime=g,gtol=1e-6,
                                       maxiter=10000,retall=1)
Warning: Desired error not necessarily achieved due to precision loss.
         Current function value: 1.850124
         Iterations: 1
         Function evaluations: 44
         Gradient evaluations: 33
>>> print(xk2)
[ 1.0080331  -0.06300207]
>>> xk3,iterados3 = optimize.fmin_bfgs(f,[1,1],fprime=g,gtol=1e-6,
                                       maxiter=10000,retall=1)
Warning: Desired error not necessarily achieved due to precision loss.
         Current function value: 0.000000
         Iterations: 21
         Function evaluations: 131
         Gradient evaluations: 119
>>> print(xk3)
[ 1.99974274  -0.99996761]
>>> xk4,iterados4 = optimize.fmin_bfgs(f,[4,4],fprime=g,gtol=1e-6,
                                       maxiter=10000,retall=1)
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 20
         Function evaluations: 23
         Gradient evaluations: 23
>>> print(xk4)
[ 2.00000005 -1.        ]
```

En la Tabla 8.5 se resumen resultados obtenidos y en la Figura 8.10 se muestra la gráfica de $\|\mathbf{g}_k\|$ vs. iteraciones, para cada iterado inicial.

Tabla 8.5. Solución, número de iteraciones, número de evaluaciones de la función, número de evaluaciones del gradiente y valor de la función objetivo, para el método BFGS.

$\mathbf{x}_0$	$\overline{\mathbf{x}}$	it	ef	eg	$f(\overline{\mathbf{x}})$
$\mathbf{x}_0^{(1)}$	$(-0.25050838, 0.15865314)$	2	44	33	27.122039
$\mathbf{x}_0^{(2)}$	$(1.0080331, -0.06300207)$	1	44	33	1.850124
$\mathbf{x}_0^{(3)}$	$(1.99974274, -0.99996761)$	21	131	119	0.000000
$\mathbf{x}_0^{(4)}$	$(2.00000005, -1.)$	20	23	23	0.000000

Para la función f dada, se observa que el método BFGS es más lento que el método de Newton y el método CGPR (ver los Ejemplos 8.6 y 8.8), cuando el iterado inicial está cerca del mínimo; más aún, el método falló con los tres primeros iterados iniciales, sólo convergió con el último iterado inicial.

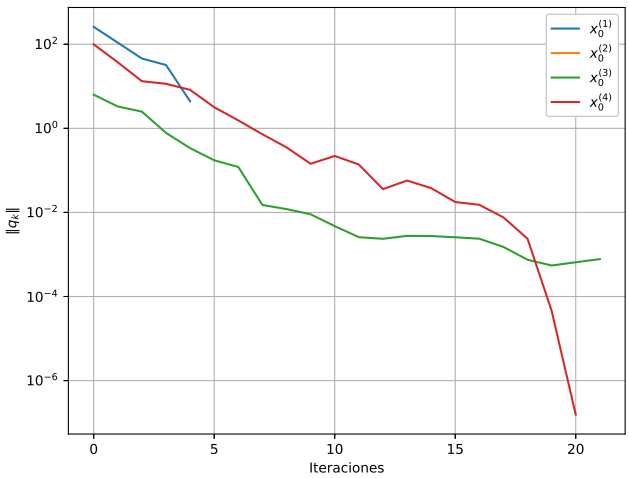


Figura 8.10. Iteraciones vs. $\|\mathbf{g}_k\|$ del método BFGS, para la función f y cada uno de los iterados iniciales del Ejemplo 8.6

Existen otros métodos quasi-Newton que se obtienen usando en el Algoritmo 8.7 una fórmula de actualización de B_k (o de H_k) distinta a la fórmula de actualización BFGS. A continuación listamos dos de ellos.

1. Método DFP (Devidon-Fletcher-Powell), cuya fórmula de actualización está dada por:

$$B_k = \left(I - \frac{\mathbf{y}_k \mathbf{s}_k^T}{\mathbf{y}_k^T \mathbf{s}_k}\right) H_k \left(I - \frac{\mathbf{y}_k \mathbf{s}_k^T}{\mathbf{y}_k^T \mathbf{s}_k}\right) + \frac{\mathbf{y}_k \mathbf{y}_k^T}{\mathbf{y}_k^T \mathbf{s}_k}$$

   o

$$H_k = H_k - \frac{H_k \mathbf{y}_k \mathbf{y}_k^T H_k}{\mathbf{y}_k^T H_k \mathbf{y}_k} + \frac{\mathbf{s}_k \mathbf{y}_k^T}{\mathbf{y}_k^T \mathbf{s}_k}.$$

2. Método SR1 (Symmetric rank-1), cuya fórmula de actualización está dada por:

$$B_k = B_k + \frac{(\mathbf{y}_k - B_k s_k)(\mathbf{y}_k - B_k s_k)^T}{(\mathbf{y}_k - B_k s_k)^T \mathbf{s}_k}$$

   o

$$H_k = H_k + \frac{(\mathbf{y}_k - H_k s_k)(\mathbf{y}_k - H_k s_k)^T}{(\mathbf{y}_k - H_k s_k)^T \mathbf{s}_k}.$$

Finalicemos esta sección comparando el comportamiento de las funciones `fmin_ncg`, `fmin_cg`, `fmin_bfgs` y `newton_min`, cuando son aplicadas a la función de Rosenbrock (8.13).

Ejemplo 8.10. Comparemos el comportamiento del método de Newton y los métodos CGPR y BFGS, cuando son aplicados a la función de Rosenbrock (8.13), cuyo mínimo global es $\mathbf{x}^* = (1,1)$. Tomemos $\mathbf{x}_0 = (-1.2,1)$ como iterado inicial. Consideremos la implementación del método de Newton dada en la función `fmin_ncg` del módulo `optimize` de la librería `scipy`,

```
scipy.optimize.fmin_ncg(f, x0, fprime, fhess=None, args=(), maxiter=None,
full_output=0, disp=1, retall=0)
```

La función `fmin_ncg` se emplea de manera similar que las funciones `fmin_cg` y `fmin_bfgs` (para detalles visite el sitio web https://docs.scipy.org/doc/). Carguemos el módulo `optimize` y definamos la función f, el gradiente y la Hessiana.

```
>>> from scipy import optimize
>>> def f(x): return 100.0*(x[1]-x[0]*x[0])**2.0+(1-x[0])**2.0
>>> def g(x):
        return np.array([-2.0*(200.0*x[0]*(x[1]-x[0]*x[0])+1.0-x[0]),
                         200.0*(x[1]-x[0]*x[0])])
>>> def H(x):
        return np.array([[1200*x[0]**2 - 400*x[1] + 2,-400*x[0]],
                         [-400*x[0],200]])
```

Ejecutemos las funciones `fmin_ncg`, `fmin_cg`, `fmin_bfgs` y `newton_min` (Programa 8.3).

```
>>> xk1,iterados1 = optimize.fmin_ncg(f,[-1.2,1],fprime=g,fhess=H,
                                      maxiter=100000,retall=1)
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 83
         Function evaluations: 105
         Gradient evaluations: 187
         Hessian evaluations: 83
>>> print(xk1, f(xk1))
[0.9999826  0.99996514] 3.031435040312226e-10
>>> xk2,iterados2 = optimize.fmin_cg(f,[-1.2,1],fprime=g,gtol=1e-8,
                                     maxiter=100000,retall=1)
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 37
         Function evaluations: 80
         Gradient evaluations: 79
>>> print(xk2, f(xk2))
[1. 1.] 3.109179757648041e-21
>>> xk3,iterados3 = optimize.fmin_bfgs(f,[-1.2,1],fprime=g,gtol=1e-8,
                                       maxiter=100000,retall=1)
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 34
         Function evaluations: 41
         Gradient evaluations: 41
>>> print(xk3, f(xk3))
[1. 1.] 4.350946175749837e-25
>>> xk4, k4, ef4, eg4 = newton_min([-1.2,1],f,g,H,tol=1e-8)
Terminó satisfactoriamente
>>> print(xk4, k4, ef4, eg4, f(xk4))
[1. 1.] 14 48 34 1.1290571705975731e-29
```

En la Tabla 8.6 se resumen los resultados obtenidos y en la Figura 8.11 se muestra la gráfica de $\|\mathbf{g}_k\|$ vs. iteraciones, para cada uno de los algoritmos ejecutados.

Tabla 8.6. Solución, número de iteraciones, número de evaluaciones de la función, número de evaluaciones del gradiente y valor de la función objetivo, para `fmin_ncg`, `fmin_cg`, `fmin_bfgs` y `newton_min`.

Algoritmo	$\overline{\mathbf{x}}$	it	ef	eg	$f(\overline{\mathbf{x}})$
`fmin_ncg`	$(0.9999826, 0.99996514)$	83	105	187	3.031e-10
`fmin_cg`	$(1.,1.)$	37	80	79	3.109e-21
`fmin_bfgs`	$(1.,1.)$	34	41	41	4.350e-25
`newton_min`	$(1.,1.)$	14	48	34	1.129e-29

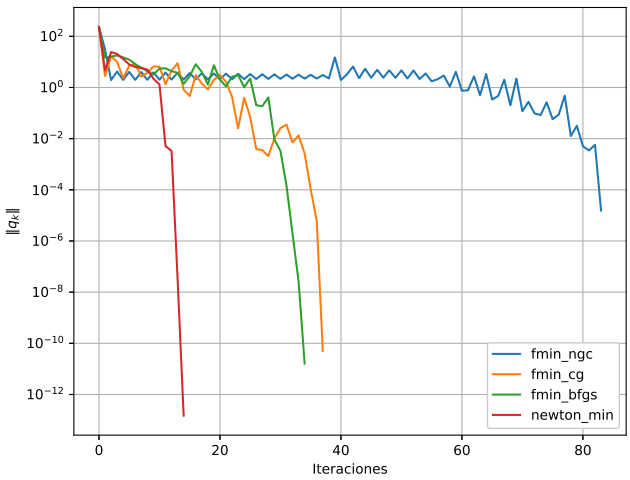


Figura 8.11. Iteraciones vs. $\|\mathbf{g}_k\|$ de los métodos `fmin_ncg`, `fmin_cg`, `fmin_bfgs` y `newton_min`, cuando son aplicados a la función de Rosenbrock

Se observa `newton_min` es más preciso que los otros algoritmos, además, requirió de menos iteraciones para satisfacer el criterio de parada ($\|\mathbf{g}_k\| \leq 10^{-8}$). Debemos tener presente que `newton_min` y `fmin_ncg` utilizan información de segundo orden, esto es, emplean la Hessiana; en cambio `fmin_cg` y `fmin_bfgs` sólo emplean el gradiente.

8.2 Ejercicios

1. Determine los puntos críticos de cada una de las siguientes funciones y caracterícelos como mínimo, máximo o punto de inflexión. También determine si cada función tiene un mínimo o un máximo global en $\mathbb{R}$.

a) $f(x) = x^3 + 6x^2 - 15x + 2$.
b) $f(x) = 2x^3 - 25x^2 - 12x + 15$.
c) $f(x) = 3x^3 + 7x^2 - 15x - 3$.
d) $f(x) = x^2 e^x$.

2. Determine los puntos críticos de cada una de las siguientes funciones y caracterícelos como mínimo, máximo o punto de ensilladura. También determine si cada función tiene un mínimo o un máximo global en $\mathbb{R}^2$.

a) $f(x,y) = x^2 - 4xy + y^2$.
b) $f(x,y) = x^4 - 4xy + y^4$.
c) $f(x,y) = 2x^3 - 3x^2 - 6xy(x - y - 1)$.
d) $f(x,y) = (x-y)^4 + x^2 - y^2 - 2x + 2y + 1$.

3. Obtenga expresiones para las primeras y segundas derivadas parciales de la función de dos variables

$$f(\mathbf{x}) = x_1^4 + x_l x_2 + (l + x_2)^2.$$

Evalúe estas derivadas en $\mathbf{x} = 0$ y demuestre que la Hessiana $H(0)$ no es definida positiva. Verifique que un mínimo local es $\mathbf{x}^* = (0.6959, -1.3479)$ con cuatro decimales.

4. Encuentre los puntos críticos de la función

$$f(\mathbf{x}) = 2x_1^3 - 3x_1^2 - 6x_1x_2(x_1 - x_2 - 1)$$

¿Cuáles de estos puntos son mínimos locales, cuáles son máximos locales y cuáles no son?

5. a) Dibujar las curvas de nivel de la función

$$f(\mathbf{x}) = x_1^2 + 4x_2^2 - 4x_1 - 8x2.$$

b) Demuestre que si el método de Cauchy se inicia desde $\mathbf{x}_0 = (0,0)$, no puede converger a $\mathbf{x}^*$ en un número finito de pasos. ¿Existen valores de $\mathbf{x}_0$ para los cuales habrá convergencia en un número finito de pasos?

6. El método de Cauchy aplicado a la función

$$f(\mathbf{x}) = 2x_1^2 - 2x_1x_2 + x_2^2 + 2x_1 - 2x_2$$

genera una sucesión $\{\mathbf{x}_k\}$. Si $\mathbf{x}_{2k+1} = (0, 1 - 1/5^k)$, show that $\mathbf{x}_{2k+3} = (0, 1 - 1/5^{k+1})$. Dibuje en el plano x_1 - x_2 los pasos sucesivos tomados por el método si se elige $\mathbf{x}_0 = 0$, y deduzca el mínimo de $f(\mathbf{x})$.

7. Para la función

$$f(\mathbf{x}) = x_1^2 + 2x_2^2 + 4x_1 + 4x_2$$

prueba por inducción que el método de Cauchy aplicado con un iterado inicial $\mathbf{x}_0 = 0$ genera la sucesión $\{\mathbf{x}_k\}$, donde

$$\mathbf{x}_{k+1} = \left(\frac{2}{3^k} - 2\,,\, \left(-\frac{1}{3}\right)^k - 1\right).$$

De aquí deduzca el mínimo de $f(\mathbf{x})$.

8. a) Encuentre los puntos críticos de la función

$$f(\mathbf{x}) = 2x_1^2 + x_2^2 - 2x_1x_2 + 2x_1^3 + x_1^4$$

que son mínimos globales.

b) Encuentre la más grande bola abierta centrada en $\mathbf{x}^* = 0$ en la cual la Hessiana $H(\mathbf{x}^*)$ es definida positiva.

c) ¿Para qué valores de $\mathbf{x}_0$ en la bola abierta hallada en b) el método de Newton converge, asumiendo que $\mathbf{x}_1 = \mathbf{x}_0$?

9. Obtenga expresiones para las primeras y segundas derivadas parciales de la función de dos variables

$$f(\mathbf{x}) = \frac{1}{2}(x_1^2 + x_2^2)e^{x_1^2 - x_2^2}$$

y verifique que $\mathbf{x}^* = 0$ es un mínimo local de esta función. Evalúe todas las derivadas en $\widehat{\mathbf{x}} = (1,1)$ y muestre que la Hessiana $H(\widehat{\mathbf{x}})$ no es definida positiva. Muestre que $\alpha = 3$ es el entero positivo más pequeño tal que $H(\widehat{\mathbf{x}}) + \alpha I$ es definida positiva y emplee la matriz resultante para realizar una iteración del método de Newton, comenzando en $\widehat{\mathbf{x}}$. (No utilice una búsqueda lineal.)

10. Aplique el método BFGS al problema 5, escogiendo $\mathbf{x}_0 = 0$ and $H(\mathbf{x}_0) = I$. Muestre que la línea $\mathbf{x} = \mathbf{x}_1 + \lambda \mathbf{d}_1$ pasa por $\mathbf{x}^*$, garantizando así la terminación en dos pasos.

Bibliografía

[1] T. M. Apostol (1985). *Calculus, Vol. I y II*, Segunda Edición, Editorial Reverté, Barcelona.

[2] D.P. Bertsekas (1999). Nonlinear Programming, second edition, Athena Scientific, Belmont, Massachusetts.

[3] J. Butcher (1987). *The Numerical Analysis of Ordinary Differential Equations: Runge-Kutta and General Linear Methods*, Wiley, Chichester.

[4] J. Casti (1987). *Linear Dynamical Systems*, Academic Press, Inc., Orlando.

[5] J.W. Cooley and J.W. Tukey (1965). An algorithm for the machine calculation of complex Fourier series. *Math. Comput.* **19**, 297—301.

[6] B.N. Datta, L.M. Hernández-Ramos y M. Raydan (2017). *Análisis Numérico: Teoría y Práctica*, Editorial de la Universidad Nacional del Sur. Ediuns, Bahía Blanca.

[7] R. Fletcher (2013). Practical Methods of Optimization, second edition, John Wiley & Sons, Chichester.

[8] R. Fletcher and C.M. Reeves (1964). Function minimization by conjugate gradients. *Computer Journal* **7**, 149—154.

[9] N. Giordano and H. Nakanishi (2006). *Computational Physics*. second edition. Prentice-Hall, Upper Saddle River NJ.

[10] W. Hackbusch (2016). *Iterative Solution of Large Sparse Systems of Equations*, second Edition, Springer.

[11] E. Hairer and G. Wanner (1980). *Solving Ordinary Differential Equations I and II*, Springer-Verlag, Berlín.

[12] M. T. Heath 2018). *Scientific Computing: An Introductory Survey*, Second Edition, SIAM, Philadelphia.

[13] N.J. Higham (2002). *Accuracy and Stability of Numerical Algorithms*, Second Edition, SIAM, Philadelphia.

[14] D. Kincaid y W. Cheney (1994). *Análisis Numérico. Las Matemáticas del Cálculo Científico*, Addison-Wesley Iberoamericana, Wilmintong, Delaware.

[15] HB. Keller (1968). *Numerical Methods for Two-Point Boundary-Value Problems*, Blaisdell.

[16] C. T. Kelley (1995). *Iterative Methods for Linear and Nonlinear Equations*, SIAM, Philadelphia.

[17] .D. Lambert (1991). *Numerical Methods for Ordinary Differential Systems: The Initial Value Problem*, John Wiley & Sons, New York.

[18] D.G. Luenberger and Y. Ye (2008). Linear and Nonlinear Programming, third edition, Springer, New York.

[19] B. Noble and J. W. Daniel (1977). *Applied Linear Algebra*, Prentice-Hall, Englewood Cliffs, NJ.

[20] J. Nocedal Stephen and J. Wright (2006). *Numerical Optimization*, Second Edition, Springer, New York.

[21] J. Ortega and W. Rheinboldt (1970). *Iterative Solution of Nonlinear Equations in Several Variables*, Academic Press, New York.

[22] M. L. Overton (2001). *Computing with IEEE Floating Point Arithmetic*, SIAM, Philadelphia.

[23] E. Polak and G. Ribière (1969). Note sur la convergence de méthodes de directions conjuguées. *Revue Française d'Informatique et de Recherche Opérationnelle* **16**, 35—43.

[24] P. Pedregal (2004). Introduction to Optimization, Springer-Verlag, Berlin.

[25] A. Quarteroni, R. Sacco, and F. Saleri (2000). *Numerical Mathematics*, Springer-Verlag, New York,

[26] W.F. Ramirez (1998). *Computational Methods in Process Simulation*, Elsevier Science & Technology Books.

[27] S. Rosłoniec (2008). *Fundamental Numerical Methods for Electrical Engineering*, Springer-Verlag, Berlín.

[28] W. Rudin (1980). *Principios de Análisis Matemático*, Tercera Edición, McGraw-Hill, México.

[29] Y. Saad (2003). *Iterative Methods for Sparse Linear Systems*, Second Edition, SIAM, Philadelphia.

[30] L. R. Scott (2011). *Numerical Analysis*, Princeton University Press, New Jersey.

[31] G. Strang (1988). *Linear Algebra and its Applications*, Academic Press, New York.

[32] L. N. Trefethen and D. Bau (1997). *Numerical Linear Algebra*. SIAM, Philadelphia.

[33] H. Zhang, C. Guo, X. Su, and C. Zhu. (2015). Measurement Data Fitting Based on Moving Least Squares Method. *Mathematical Problems in Engineering*, V. 2015, Article ID 195023, 10 pages.

[34] F. Zhang (1999). *Matrix Theory: Basic Results and Techniques*, Springer, New York.