Resolución de sistemas lineales: métodos iterativos ¶


```
import numpy as np
```

Establecemos el formato

```
np.set_printoptions(precision = 8)   # ocho decimales
np.set_printoptions(suppress = True) # no usar notación exponencial
```

Vamos a resolver sistemas de ecuaciones lineales con el mismo número de ecuaciones que de incógnitas con matriz de coeficientes no singular que, por lo tanto, tiene solución única.

Dados los números a_{ij} y b_j para $i,j=1,2,\ldots,n$ se trata de hallar los números $x_1,x_2,\ldots,x_n$ que verifican las n ecuaciones lineales

$$\begin{aligned} a_{11}x_1+a_{12}x_2+\ldots+a_{1n}x_n &= b_1,\\ a_{21}x_1+a_{22}x_2+\ldots+a_{2n}x_n &= b_2,\\ &\vdots\\ a_{n1}x_1+a_{n2}x_2+\ldots+a_{nn}x_n &= b_n. \end{aligned}$$

Métodos iterativos

Los métodos iterativos son, en general, más eficientes que los métodos directos para resolver sistemas de ecuaciones lineales poco densos o dispersos (con muchos elementos de la matriz de coeficientes igual a cero).

Reescribimos el sistema

$$A\mathbf{x} = \mathbf{b} \Longleftrightarrow \mathbf{x} = B\mathbf{x} + \mathbf{c}$$

donde B es una matriz de orden n y c es un vector columna de dimensión n.

Entonces, si $\mathbf{x}^{(0)}$ es el punto inicial, para $k = 1, 2, \ldots$

$$\mathbf{x}^{(k)} = B\mathbf{x}^{(k-1)} + \mathbf{c}$$

y generamos una sucesión de vectores (matrices columna) $\left\{\mathbf{x}^{(m)}\right\}$ que, caso que converja, converge a la solución del sistema $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$.

Método de Jacobi

Si tenemos un sistema 4×4

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 &= b_3 \\ a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 &= b_4 \end{aligned}$$

y despejamos la primera incógnita en la primera ecuación, la segunda incógnita en la segunda y así sucesivamente

$$\begin{aligned} x_1 &= \frac{b_1 - a_{12}x_2 - a_{13}x_3 - a_{14}x_4}{a_{11}} \\ x_2 &= \frac{b_2 - a_{21}x_1 - a_{23}x_3 - a_{24}x_4}{a_{22}} \\ x_3 &= \frac{b_3 - a_{31}x_1 - a_{32}x_2 - a_{34}x_4}{a_{33}} \\ x_4 &= \frac{b_4 - a_{41}x_1 - a_{42}x_2 - a_{43}x_3}{a_{44}} \end{aligned}$$

Y obtenemos los valores de la primera iteración a partir de los valores iniciales

$$\begin{aligned} x_1^{(1)} &= \frac{b_1 - a_{12}x_2^{(0)} - a_{13}x_3^{(0)} - a_{14}x_4^{(0)}}{a_{11}} \\ x_2^{(1)} &= \frac{b_2 - a_{21}x_1^{(0)} - a_{23}x_3^{(0)} - a_{24}x_4^{(0)}}{a_{22}} \\ x_3^{(1)} &= \frac{b_3 - a_{31}x_1^{(0)} - a_{32}x_2^{(0)} - a_{34}x_4^{(0)}}{a_{33}} \\ x_4^{(1)} &= \frac{b_4 - a_{41}x_1^{(0)} - a_{42}x_2^{(0)} - a_{43}x_3^{(0)}}{a_{44}} \end{aligned}$$

Si generalizamos tenemos el siguiente algoritmo

Algoritmo

- Elegir una aproximación inicial $\mathbf{x}^{(0)}$
- Para $k = 1, 2, \ldots, \mathrm{MaxIter}$
  - Para $i = 1, 2, \ldots, n$, calcular

$$x_i^{(k)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k-1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k-1)} \right)$$

  - Si se cumple el criterio de parada, tomar $\mathbf{x}^{(k)}$ como aproximación a la solución.

Ejercicio 1

Escribir una **`jacobi(A,b,tol,maxiter=1000)`** para resolver $Ax = b$ por el método de Jacobi.

- Resolver

$$\begin{cases} 5x_1 + x_2 - x_3 - x_4 &= 1 \\ x_1 + 4x_2 - x_3 + x_4 &= 1 \\ x_1 + x_2 - 5x_3 - x_4 &= 1 \\ x_1 + x_2 + x_3 - 4x_4 &= 1 \end{cases}$$

  por el método de Jacobi, usando una tolerancia absoluta **`tol = 1.e-6`** con norma 2. Es decir, usaremos como criterio de parada **`np.linalg.norm(x-xant) < tol`** , con **`x`** la solución aproximada en esta iteración y **`xant`** la solución aproximada en la iteración anterior. Tomar como punto inicial $\mathbf{x}^{(0)} = \mathbf{0}$.

```
A = np.array([[5.,1,-1,-1],[1,4,-1,1],[1,1,-5,-1],[1,1,1,-4]])
b = np.array([1.,1,1,1])
```

- Como argumento de salida, dar la solución **`x`** y el número de iteraciones **`num_iter`** .
- Comprobar que la solución es correcta comparándola con **`xs = np.linalg.solve(A,b)`** .
- Comprobar que también funciona correctamente para resolver el sistema $Ax = b$ de matrices A y b generadas de la forma siguiente

```
n = 9
A1 = np.diag(np.ones(n))*2
A2 = np.diag(np.ones(n-1),1)
A = A1 + A2 + A2.T
b = np.concatenate((np.arange(1,6),np.arange(4,0,-1)))*1.
```

Nota

Para asignar un vector o matriz se debe hacer una copia

- **`xant = np.copy(x)`**

```
%run Ejercicio1.py
```

```
------------- Sistema 1 -------------

---- Datos ----
A
[[ 5.  1. -1. -1.]
 [ 1.  4. -1.  1.]
 [ 1.  1. -5. -1.]
 [ 1.  1.  1. -4.]]
b
[1. 1. 1. 1.]

---- Solución ----

 18  iteraciones

x aproximada
[ 0.09374986  0.24999988 -0.09374986 -0.18749994]

x exacta
[ 0.09375  0.25    -0.09375 -0.1875 ]

------------- Sistema 2 -------------

---- Datos ----
A
[[2. 1. 0. 0. 0. 0. 0. 0. 0.]
 [1. 2. 1. 0. 0. 0. 0. 0. 0.]
 [0. 1. 2. 1. 0. 0. 0. 0. 0.]
 [0. 0. 1. 2. 1. 0. 0. 0. 0.]
 [0. 0. 0. 1. 2. 1. 0. 0. 0.]
 [0. 0. 0. 0. 1. 2. 1. 0. 0.]
 [0. 0. 0. 0. 0. 1. 2. 1. 0.]
 [0. 0. 0. 0. 0. 0. 1. 2. 1.]
 [0. 0. 0. 0. 0. 0. 0. 1. 2.]]
b
[1. 2. 3. 4. 5. 4. 3. 2. 1.]

---- Solución ----

 307  iteraciones

x aproximada
[0.5        0.00000025 1.5        0.00000041 2.5        0.00000041
 1.5        0.00000025 0.5       ]

x exacta
[0.5 0.  1.5 0.  2.5 0.  1.5 0.  0.5]
```

Método de Gauss-Seidel

Si tenemos un sistema 4×4

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 &= b_3 \\ a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 &= b_4 \end{aligned}$$

y despejamos la primera incógnita en la primera ecuación, la segunda incógnita en la segunda y así sucesivamente

$$\begin{aligned} x_1 &= \frac{b_1 - a_{12}x_2 - a_{13}x_3 - a_{14}x_4}{a_{11}} \\ x_2 &= \frac{b_2 - a_{21}x_1 - a_{23}x_3 - a_{24}x_4}{a_{22}} \\ x_3 &= \frac{b_3 - a_{31}x_1 - a_{32}x_2 - a_{34}x_4}{a_{33}} \\ x_4 &= \frac{b_4 - a_{41}x_1 - a_{42}x_2 - a_{43}x_3}{a_{44}} \end{aligned}$$

Y obtenemos los valores de la primera iteración a partir de los valores iniciales, pero conforme vamos teniendo resultados nuevos de una variable, los actualizamos

$$\begin{aligned} x_1^{(1)} &= \frac{b_1 - a_{12}x_2^{(0)} - a_{13}x_3^{(0)} - a_{14}x_4^{(0)}}{a_{11}} \\ x_2^{(1)} &= \frac{b_2 - a_{21}x_1^{(1)} - a_{23}x_3^{(0)} - a_{24}x_4^{(0)}}{a_{22}} \\ x_3^{(1)} &= \frac{b_3 - a_{31}x_1^{(1)} - a_{32}x_2^{(1)} - a_{34}x_4^{(0)}}{a_{33}} \\ x_4^{(1)} &= \frac{b_4 - a_{41}x_1^{(1)} - a_{42}x_2^{(1)} - a_{43}x_3^{(1)}}{a_{44}} \end{aligned}$$

Si generalizamos tenemos el siguiente algoritmo

Algoritmo

El algoritmo que hemos aplicado se expresa formalmente

- Elegir una aproximación inicial $\mathbf{x}^{(0)}$
- Para $k = 1, 2, \ldots$,MaxIter
  - Para $i = 1, 2, \ldots, n$, calcular

$$x_i^{(k)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k-1)} \right)$$

  - Si se cumple el criterio de parada, tomar $\mathbf{x}^{(k)}$ como aproximación a la solución.

Ejercicio 2

Escribir una función **`gauss_seidel(A,b,tol,maxiter=1000)`** para resolver $Ax = b$ por el método de Gauss-Seidel.

- Resolver los mismos sistemas del ejercicio anterior usando Gauss-Seidel con la misma tolerancia y criterio de parada.
- Escribir las mismos argumentos de salida que en el ejercicio anterior y comparar el número de iteraciones.

```
%run Ejercicio2.py
```

```
------------- Sistema 1 -------------

 8  iteraciones

x aproximada
[ 0.09375005  0.24999998 -0.09375003 -0.1875    ]

x exacta
[ 0.09375  0.25    -0.09375 -0.1875 ]

------------- Sistema 2 -------------

 125  iteraciones

x aproximada
[0.49999855 0.00000262 1.49999658 0.00000383 2.49999617 0.00000346
 1.4999972  0.00000194 0.49999903]

x exacta
[0.5 0.  1.5 0.  2.5 0.  1.5 0.  0.5]
```

Método de relajación

Algoritmo

- Elegir el punto inicial $\mathbf{x}^{(0)}$
- Para $k = 1, 2, \ldots, MaxIter$

  - Para $i = 1, 2, \ldots, n$ calcular

$$x_i^{(k)} = \frac{\omega}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k-1)} \right) + (1 - \omega)\, x_i^{(k-1)}$$

  - Si se satisface el criterio de parada, tomar $\mathbf{x}^{(k)}$ como solución aproximada

Si converge, entonces $\omega \in (0, 2)$.

Ejercicio 3

Escribir una función **`relajacion(A,b,w,tol,maxiter=1000)`** para resolver $Ax = b$ por el método de relajación.

- Resolver el segundo sistema del ejercicio anterior usando el método de relajación con $\omega = 1.5$ y la misma tolerancia y criterio de parada.
- Escribir las mismos argumentos de salida que en el ejercicio anterior y comparar el número de iteraciones.

```
%run Ejercicio3.py
```

```
------------- Sistema 2 -------------

 35  iteraciones

x aproximada
[0.49999957 0.00000066 1.49999926 0.0000007  2.49999941 0.00000046
 1.49999969 0.00000018 0.49999992]

x exacta
[0.5 0.  1.5 0.  2.5 0.  1.5 0.  0.5]
```

Ejercicios propuestos

Relajación: estudio del parámetro ω

Ejercicio 4

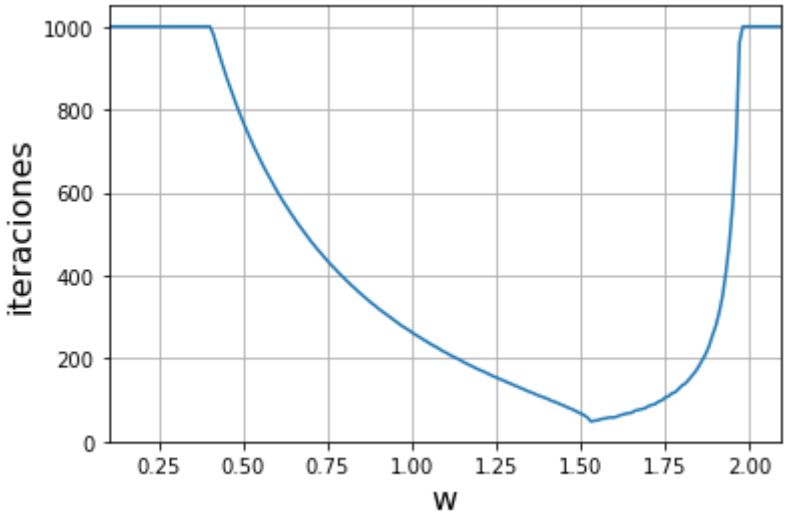
Escribir un programa que calcule el valor de ω óptimo para encontrar la solución del sistema del ejercicio 3 con el método de sobrerrelajación. Usar **`maxiter = 1000`** y **`tol = 1.e-12`** . Calcular y dibujar el número de iteraciones para los valores de ω **`w = np.arange(0.1,2.1,0.01)`** .

Nota

- Usar **`k = np.argmin(num_iter)`** para encontrar el índice del valor óptimo.

```
%matplotlib inline
%run Ejercicio4.py
```

```
Óptimo w = 1.53  num_iter = 48
```



Sistemas tridiagonales

En el segundo sistema, la matriz de coeficientes es tridiagonal. Este tipo de matrices dispersas (con muchos ceros) son habituales en problemas de la física (ecuación del calor, ecuación de onda) que se resuelven con diferencias finitas.

Por todo esto, se adaptan algoritmos ya existentes a estos casos particulares. Para ello, por ejemplo, en lugar de almacenar la matriz A del segundo sistema como una matriz cuadrada, podemos almacenar sólo las diagonales en una matriz rectangular A_r de la siguiente forma

```
n = 9

Ar = np.zeros((n,3))
Ar[:,0] = np.concatenate((np.array([0]),np.ones((n-1),)))
Ar[:,1] = np.ones((n),)*2
Ar[:,2] = np.concatenate((np.ones((n-1),),np.array([0])))

b = np.concatenate((np.arange(1,6),np.arange(4,0,-1)))*1.
```

Ejercicio 5

Modificar el código de las funciones de los tres ejercicios anteriores de forma que admitan la matriz A en la forma A_r (matriz rectangular).

- Resolver el sistema con la misma tolerancia y criterio de parada que en los ejercicios anteriores.
- Escribir los mismos argumentos de salida que en los ejercicios anteriores y comparar el número de iteraciones.
- Tomar, para el método de relajación, $\omega = 1.5$.

```
%run Ejercicio5.py
```

```
------------- Sistema 2 -------------

          ---- Datos ----
Ar
[[0. 2. 1.]
 [1. 2. 1.]
 [1. 2. 1.]
 [1. 2. 1.]
 [1. 2. 1.]
 [1. 2. 1.]
 [1. 2. 1.]
 [1. 2. 1.]
 [1. 2. 0.]]
b
[1. 2. 3. 4. 5. 4. 3. 2. 1.]

          ---- Jacobi ----

 307  iteraciones

x
[0.5        0.00000025 1.5        0.00000041 2.5        0.00000041
 1.5        0.00000025 0.5       ]

          ---- Gauss-Seidel ----

 125  iteraciones

x
[0.49999855 0.00000262 1.49999658 0.00000383 2.49999617 0.00000346
 1.4999972  0.00000194 0.49999903]

          ---- Relajación ----

 35  iteraciones

x
[0.49999957 0.00000066 1.49999926 0.0000007  2.49999941 0.00000046
 1.49999969 0.00000018 0.49999992]

          ---- Solución exacta ----

[0.5 0.  1.5 0.  2.5 0.  1.5 0.  0.5]
```