Programa: SEL (sistema de ecuaciones lineales)

por: Ricardo Santana, Gilberto Noguera

A. Introducción

El código proporcionado una clase que implementa métodos para resolver sistemas de ecuaciones lineales (SEL) utilizando diferentes enfoques, como métodos directos (eliminación gaussiana, factorización PALU y factorización de Cholesky) y métodos iterativos (Jacobi, Gauss-Seidel y SOR).

B. SEL

Entrada:

- Matriz $A \in \mathbb{R}^{n \times n}$
- Matriz $b \in \mathbb{R}^{n \times 1}$

Salida:

- Matriz resultado del sistema $Ax = b, x \in \mathbb{R}^{n \times 1}$
- Matriz resto $R \in \mathbb{R}^{n \times 1}$ (para metodos iterativos)

C. Programa

1. Importación de bibliotecas: El código comienza importando las bibliotecas necesarias, como numpy (para operaciones numéricas), sys (para interactuar con el sistema) y pandas (para el manejo de datos tabulares).
2. Definición de la clase SEL: Se define una clase llamada SEL, que contiene todos los métodos y herramientas necesarios para resolver SEL.
3. Inicialización de la clase SEL: El constructor **init** inicializa la clase SEL con los parámetros de entrada A (matriz de coeficientes), b (vector de términos independientes). También se inicializan otras variables como x0 (vector de soluciones iniciales aleatorias), D (matriz diagonal de A), Al (parte triangular inferior de A) y Au (parte triangular superior de A).
4. Métodos de menú: Los métodos metodo_directo y metodo_iterativo son métodos de menú que permiten seleccionar el método de resolución deseado (por ejemplo, Gauss, PALU, Cholesky, Jacobi, Seidel o SOR) y llamar al método correspondiente.

5. Herramientas: Los métodos sustitucion_progresiva y sustitucion_regresiva son herramientas auxiliares utilizadas en los métodos de resolución directa para realizar la sustitución progresiva y regresiva, respectivamente.
6. Verificación iterativa: Forma parte de las herramientas, el método verificacion_iterativa verifica si el sistema de ecuaciones es válido para los métodos iterativos. Comprueba si la matriz A es cuadrada, si el vector independiente tiene las dimensiones correctas y si el rango de la matriz A es igual a su tamaño.
7. Métodos directos: Los métodos resolucion_gaussiana, resolucion_palu y resolucion_cholesky son métodos de resolución directa que utilizan la eliminación gaussiana, la factorización PALU y la factorización de Cholesky, respectivamente, para resolver el SEL.
8. Métodos iterativos: Los métodos resolucion_jacobi, resolucion_seidel y resolucion_sor son métodos de resolución iterativa que utilizan los métodos de Jacobi, Gauss-Seidel y SOR, respectivamente, para resolver el SEL.
9. Cada método de resolución devuelve la solución del SEL o un mensaje de error si el sistema es incompatible, las matrices que conforman el sistema no son adecuadas o si no converge el método.

Observación:

Los metodos sustitucion_regresiva, resolucion_gaussiana, factorizacion_palu, factorizacion_cholesky, resolucion_jacobi, resolucion_seidel, resolucion_sor fueron definidos por el profesor Noguera en lenguaje Scilab.

D. Código

```
import numpy as np
import sys
import pandas as pd

class SEL():
    def __init__(self, A, b):
        self.A = np.array(A, dtype='f')
        self.b = np.reshape(np.array(b, dtype='f'), (len(b),1))
        self.x0 = np.reshape(np.random.randint(10, size=(len(A),1)), (len(b),1))
        self.D = np.diag(np.diagonal(A))
        self.Al = -np.tril(A, -1)
        self.Au = -np.triu(A, 1)

    #menu de metodos directos
```

```
def metodo_directo(self,nombre):
    nombre = nombre.lower()
    if nombre == 'gauss':
        return self.resolucion_gaussiana()
    elif nombre == 'palu':
        return self.resolucion_palu()
    elif nombre == 'cholesky':
        return self.resolucion_cholesky()

#menu de metodos iterativos
def metodo_iterativo(self, nombre, iteraciones, tolerancia, omega):
    nombre = nombre.lower()
    if nombre == 'jacobi':
        return self.resolucion_jacobi(iteraciones, tolerancia)
    elif nombre == 'seidel':
        return self.resolucion_seidel(iteraciones, tolerancia)
    elif nombre == 'sor':
        return self.resolucion_sor(iteraciones, tolerancia, omega)

#herramientas
def sustitucion_progresiva(self,triangular_inferior,vector):

    n = len(vector)
    sol = np.mat(np.zeros((n,1)))
    sol[0] = vector[0] / triangular_inferior[0, 0]
    for kk in range(1, n):
        sol[kk] = (vector[kk] - np.dot(triangular_inferior[kk, :kk], sol[:kk])) / triar
    return sol

def sustitucion_regresiva(self,triangular_superior,vector):

    n = len(vector)
    sol = np.mat(np.zeros((n,1)))
    sol[n-1] = vector[n-1]/triangular_superior[n-1,n-1]
    for kk in range(n-2,-1,-1):
        sol[kk] = (vector[kk] - sum(triangular_superior[kk,kk+1:n]*sol[kk+1:n]).T)/tri

    return sol

def verificacion_iterativa(self):
    n, m = self.A.shape
    if m != n:
        return 'La Matriz A no es cuadrada'

    n1, m1 = self.b.shape
    if n1 != n or m1 != 1:
        return 'Medida del vector independiente no valida'


    if np.linalg.matrix_rank(self.A) != n or np.linalg.matrix_rank(self.A) != m:
        return 'Sistema incompatible'
```

```
    return True

#-----metodos directos-----

#metodo de eliminacion gaussiana por pivoto parcial
def resolucion_gaussiana(self):
    #copias
    A, b = np.copy(self.A), np.copy(self.b)

    (n,m) = A.shape
    if (m!=n):
        print('La matriz no es cuadrada')
        return

    (n1,m1) = b.shape
    if (n1!=n or m1!=1):
        print('Medida del vector independiente no valida')
        return

    if (np.linalg.matrix_rank(A) != m):
        print('Sistema incompatible o con infinitas soluciones')
        return

    for j in range(0,n-1):
        k = np.argmax(np.abs(A[j:n,j]))
        k = k+j
        faux = np.copy(A[j,:])
        A[j,:] = np.copy(A[k,:])
        A[k,:] = np.copy(faux)
        baux = np.copy(b[j])
        b[j] = np.copy(b[k])
        b[k] = np.copy(baux)
        for i in range(j+1,n):
            mu = A[i,j]/A[j,j]
            A[i,:] = A[i,:] - mu*A[j,:]
            b[i] = b[i] - mu*b[j]

    sol = self.sustitucion_regresiva(A, b)

    return sol

#metodo por factorifacion PALU
def factorizacion_palu(self):
    A = self.A

    n, m = A.shape
    if m != n:
        raise ValueError('La Matriz A no es cuadrada')

    L = np.eye(n)
```

```
    P = np.eye(n)
    U = np.copy(A)

    for j in range(n-1):
        k = np.argmax(np.abs(U[j:n, j]))+j

        U[[j, k], :] = U[[k, j], :]
        P[[j, k], :] = P[[k, j], :]
        L[[j, k], :j] = L[[k, j], :j]

        for i in range(j+1, n):
            L[i, j] = U[i, j] / U[j, j]
            U[i, j+1:n] = U[i, j+1:n] - L[i, j] * U[j, j+1:n]
            U[i, j] = 0

    return {'P':P, 'A':self.A, 'L':L, 'U':U}

def resolucion_palu(self):
    A, b = np.copy(self.A), np.copy(self.b)

    palu = self.factorizacion_palu()
    w = np.matmul(palu['P'], b)

    #resolviendo Ly=w
    y = self.sustitucion_progresiva(palu['L'], w)

    #resolviendo Ux=y
    x = self.sustitucion_regresiva(palu['U'], y)
    return x

#metodo por factorizacion cholesky
def factorizacion_cholesky_palu(self):
    # Prueba para verificar si la matriz es simétrica
    if not np.allclose(A, A.T):
        return 'La matriz A no es simétrica'

    # Prueba para determinar si la matriz es definida positiva
    auto_valores = np.linalg.eigvalsh(A)
    if np.any(auto_valores <= 0):
        return 'La matriz A no es definida positiva'

    palu = self.factorizacion_palu()
    D = np.sqrt(np.diag(np.diagonal(palu['U'])))
    L_chlk = np.matmul(palu['L'],D)

    return L_chlk, L_chlk.T

def factorizacion_cholesky(self):
    A = np.copy(self.A)

    n, m = A.shape
```

```
    U = np.zeros((n, m))

    # Prueba para determinar si la matriz es cuadrada
    if m != n:
        return 'La matriz A no es cuadrada'

    # Prueba para verificar si la matriz es simétrica
    if not np.allclose(A, A.T):
        return 'La matriz A no es simétrica'

    # Prueba para determinar si la matriz es definida positiva
    auto_valores = np.linalg.eigvalsh(A)
    if np.any(auto_valores <= 0):
        return 'La matriz A no es definida positiva'
    else:
        U = np.triu(A)

    for k in range(m):
        for j in range(k+1, m):
            U[j, j:m] = U[j, j:m] - U[k, j:m] * (U[k, j] / U[k, k])
        U[k, k:m] = U[k, k:m] / np.sqrt(U[k, k])

    L = U.T
    return L, U

def resolucion_cholesky(self):
    A, b = np.copy(self.A), np.copy(self.b)

    aux = self.factorizacion_cholesky()

    #L, Lt = self.factorizacion_cholesky_palu()
    if not type(aux) == str:
        L, Lt = aux
    else:
        return aux

    #resolviendo Ly=b
    y = self.sustitucion_progresiva(L, b)

    #resolviendo (Lt)x=y
    x = self.sustitucion_regresiva(Lt, y)
    return x

#-----metodos iterativos-----

#metodo de jacobi
def resolucion_jacobi(self, max_iteraciones, tolerancia):

    if self.verificacion_iterativa() != True:
        return self.verificacion_iterativa()
```

```
    M = np.matmul(np.linalg.inv(self.D), (self.Al + self.Au))

    rho = np.max(np.abs(np.linalg.eigvals(M)))
    if rho >= 1:
        return 'El metodo no converge'

    v = np.matmul((np.linalg.inv(self.D)), self.b)
    xk = self.x0
    rk = np.matmul(self.A, xk) - self.b
    Nrk = np.linalg.norm(rk, 1)
    k = 0
    res = np.array([k, Nrk])

    while True:
        if Nrk <= tolerancia:
            return xk, res
        if k > max_iteraciones:
            return xk, res

        res = np.append(res, Nrk)
        xk = np.matmul(M, xk) + v
        rk = np.matmul(self.A, xk) - self.b
        Nrk = np.linalg.norm(rk, 1)
        k += 1

#metodo de GaussSeidel
def resolucion_seidel(self, max_iteraciones, tolerancia):
    if self.verificacion_iterativa() != True:
        return self.verificacion_iterativa()

    inv_D_Al = np.linalg.inv(self.D - self.Al)
    M = np.matmul(inv_D_Al, self.Au)

    rho = np.max(np.abs(np.linalg.eigvals(M)))
    if rho >= 1:
        return 'El metodo no converge'

    v = np.matmul(inv_D_Al, self.b)
    xk = self.x0
    rk = np.matmul(self.A, xk) - self.b
    Nrk = np.linalg.norm(rk, 1)
    k = 0
    res = np.array([k, Nrk])

    while True:
        if Nrk <= tolerancia:
            return xk, res
        if k > max_iteraciones:
            return xk, res

        res = np.append(res, [Nrk], axis=0)
```

```
        xk = np.matmul(M, xk) + v
        rk = np.matmul(self.A, xk) - self.b
        Nrk = np.linalg.norm(rk, 1)
        k += 1

#metodo SOR
def resolucion_sor(self, max_iteraciones, tolerancia, omega):
    if self.verificacion_iterativa() != True:
        return self.verificacion_iterativa()

    w = omega
    Inv_D_wAl = np.linalg.inv(self.D - w * self.Al)
    M = np.matmul(Inv_D_wAl, (w * self.Au + (1 - w) * self.D))

    rho = np.max(np.abs(np.linalg.eigvals(M)))
    if rho >= 1:
        return 'El metodo no converge'

    v = np.matmul(Inv_D_wAl, w * self.b)
    xk = self.x0
    rk = np.matmul(self.A, xk) - self.b
    Nrk = np.linalg.norm(rk, 1)
    k = 0
    res = np.array([k, Nrk])

    while True:
        if Nrk <= tolerancia:
            return xk, res
        if k > max_iteraciones:
            return xk, res
```