Analyse Numérique
Systèmes linéaires

Simone Deparis

April 27, 2023

Contents

1 Résolution de systèmes linéaires

1.1 Méthodes Directes

D'abord quelques exemples d'utilisations de la factorisation LU et de Choleski, ensuite on passe aux methodes itéreatives

1.1.1 Exercice 1 série 8

On considère le système linéaire $A\mathbf{x} = \mathbf{b}$ où :

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

1. Calculez la factorisation LU de la matrice A avec Python à l'aide du code ci-dessous.
2. Résolvez le système linéaire $A\mathbf{x} = \mathbf{b}$ en utilisant la factorisation trouvée au point précédent (Ne plus utiliser Python. Mais ici on va le faire)

3. Calculez le déterminant de la matrice A en utilisant sa factorisation LU.

[1]:
```
# importing libraries used in this notebook
import numpy as np
import matplotlib.pyplot as plt

# scipy.linalg.lu : LU decomposition
from scipy.linalg import lu
# scipy.linalg.cholesky : Cholesky decomposition
from scipy.linalg import cholesky

import pprint

np.set_printoptions(precision=4, suppress=True, linewidth=120)
```

Partie 1

[2]:
```
A = np.array([[3, 6, 7],
              [1, 1, 4],
              [2, 4, 8] ])

# LU factorisation with pivoting
P, L, U = lu(A)

print("A = P L U")
pprint.pprint(P.dot(L.dot(U)) )

print("P:")
pprint.pprint(P)

print ("L:")
pprint.pprint(L)

print ("U:")
pprint.pprint(U)
```

```
A = P L U
array([[3., 6., 7.],
       [1., 1., 4.],
       [2., 4., 8.]])
P:
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
L:
array([[ 1.    ,  0.    ,  0.    ],
       [ 0.3333,  1.    ,  0.    ],
       [ 0.6667, -0.    ,  1.    ]])
U:
```

```
array([[ 3.    ,  6.    ,  7.    ],
       [ 0.    , -1.    ,  1.6667],
       [ 0.    ,  0.    ,  3.3333]])
```

Partie 2 Si P est l'identité, $A = LU$.

1. résolvez pour $\mathbf{y}$ tel que $L\mathbf{y} = \mathbf{b}$ par substitution progressive
2. résolvez pour $\mathbf{x}$ tel que $U\mathbf{x} = \mathbf{y}$ par substitution retrograde

Si P n'est pas l'identité, $A = PLU$.

Attention, une matrice de permutation est une matrice orthogonale car les colonnes sont orthonormées. Donc $P^{-1} = P^T$. Du coup : $P^T A = LU$ et

$$A\mathbf{x} = \mathbf{b} \Leftrightarrow P^T A\mathbf{x} = P^T\mathbf{b} \Leftrightarrow LU\mathbf{x} = P^T\mathbf{b}.$$

Alors il faut modifier les calculs précédants comme suit:

1. résolvez pour $\mathbf{y}$ tel que $L\mathbf{y} = P^T\mathbf{b}$ par substitution progressive
2. résolvez pour $\mathbf{x}$ tel que $U\mathbf{x} = \mathbf{y}$ par substitution retrograde

[3]:
```python
def subst_progressive(L, b):
    """
    substitution progressive: résout y, L*y = b
    Input:
    - L: matrice carrée nxn, triangulaire inférieure
    - b: vector de dimension n
    Output:
    - y: vector de dimension n
    """

    # Initialisation de la solution
    y = np.zeros(L.shape[1])

    # La première ligne de Ly = b est L_{11} y_1 = b_1
    # Ensuite pour la ligne k on connait y_1, ..., y_{k-1} et elle s'écrit
    # L_{kk} y_k = b_k - ( L_{k1} y_1 + ... L_{kk-1} y_{k-1} )
    for k in range(L.shape[0]):
        # sum_k est ( L_{k1} y_1 + ... L_{kk-1} y_{k-1} )
        sum_k = 0
        for j in range(k):
            sum_k += L[k,j]*y[j]
        y[k] = 1/L[k,k]*(b[k]-sum_k)
    return y

def subst_retrograde(U, y):
    """
    substitution retrograde: résout pour x, U*x = y
    Input:
    - U: matrice carrée nxn, triangulaire supérieure
```

```
    - y: vector de dimension n
    Output:
    - x: vector de dimension n
    """

    # Initialisation de la solution
    x = np.zeros(U.shape[1])

    # La dernière ligne de Yx = y est U_{nn} x_n = y_n
    # Ensuite pour la ligne k on connait x_n, ..., x_{k+1} et elle s'écrit
    # U_{kk} x_k = y_k - ( U_{kk+1} x_{k+1} + ... U_{kn} y_{n} )
    for k in reversed(range(U.shape[0])):
        # sum_k est ( U_{kk+1} x_{k+1} + ... U_{kn} y_{n} )
        sum_k = 0
        for j in range(k+1, U.shape[0]):
            sum_k += U[k,j]*x[j]
        x[k] = 1/U[k,k]*(y[k]-sum_k)
    return x
```

[4]:
```
b = np.array([4, 5, 6])
Ptb = P.T.dot(b)

y = subst_progressive(L, Ptb)
print("y =", y)

x = subst_retrograde(U, y)
print("x =", x)

# check the residual of the equation
print("residual =",b - A.dot(x))
```

```
y = [4.     3.6667 3.3333]
x = [ 3. -2.  1.]
residual = [0. 0. 0.]
```

Partie 3

$$\det A = \det P \det L \det U$$

[5]:
```
# scipy.linalg.det : determinant
from scipy.linalg import det

det(P)*det(L)*det(U)
```

[5]:
```
-10.0
```

1.1.2 Critère de Sylvester

Les mineurs principaux d'une matrice $A \in \mathbb{R}^{n\times n}$ sont les déterminants des matrices $A_p = (a_{i,j})_{1\le i,j\le p}$, $p = 1, ..., n$.

Critère de Sylvester: une matrice symétrique $A \in \mathbb{R}^{n\times n}$ est définie positive si et seulement si les mineurs principaux de A sont tous positifs.

1.1.3 Exercice 2 série 8

On considère le système linéaire $A\mathbf{x} = \mathbf{b}$ où

$$A = \begin{pmatrix} \varepsilon & 1 & 2 \\ 1 & 3 & 1 \\ 2 & 1 & 3 \end{pmatrix}.$$

1. Déterminez pour quelles valeurs du paramètre réel $\varepsilon \in \mathbb{R}$, la matrice A est symétrique définie positive.
2. Soit maintenant $\varepsilon = 0$. On veut résoudre le système $A\mathbf{x} = \mathbf{b}$ par une méthode directe; quelle factorisation de la matrice A envisageriez-vous? Justifiez votre réponse.
3. En considérant $\varepsilon = 2$, vérifier que dans ce cas la matrice A est définie positive et calculer sa factorisation de Cholesky $A = LL^T$.
4. En supposant que $\mathbf{b} = (1, 1, 1)^T$, résolvez le système linéaire $A\mathbf{x} = \mathbf{b}$ en utilisant la factorisation de Cholesky calculée au point c).

Référence Python pour la factorisation de Cholesky : scipy.linalg.cholesky

Partie 1 En appliquant le critère de Sylvester, il suffit d'imposer

$$\begin{cases} \varepsilon > 0, \\ \det \begin{pmatrix} \varepsilon & 1 \\ 1 & 3 \end{pmatrix} = 3\varepsilon - 1 > 0, \\ \det A = 8\varepsilon - 11 > 0, \end{cases} \qquad \Rightarrow \quad \varepsilon > \frac{11}{8}.$$

Partie 2 Si $\varepsilon = 0$ la matrice A est sym'etrique, mais elle n'est pas d'efinie positive; donc on ne peut pas calculer la factorisation de Cholesky. On utilisera la m'ethode d' 'elimination de Gauss avec changement de pivot, puisque $a_{11} = 0$; par exemple, on peut consid'erer la matrice de permutation P par lignes:

$$P = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

On peut facilement voir que $A = PLU$ avec

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 2 & -5 & 1 \end{pmatrix} \qquad \text{et} \qquad U = \begin{pmatrix} 1 & 3 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 11 \end{pmatrix}.$$

Partie 3 Si $\varepsilon = 2$, la matrice A est symétrique définie positive. Ici on va utiliser $A = LL^T$. Les éléments de la matrice L de la factorisation de Cholesky de A sont:

$$\begin{aligned} l_{11} &= \sqrt{a_{11}} = \sqrt{2} \\ l_{21} &= \frac{1}{l_{11}} \cdot a_{21} = \frac{1}{\sqrt{2}} \\ l_{22} &= \sqrt{a_{22} - l_{21}^2} = \sqrt{\frac{5}{2}} \\ l_{31} &= \frac{1}{l_{11}} \cdot a_{31} = \sqrt{2} \\ l_{32} &= \frac{1}{l_{22}} \cdot (a_{32} - l_{31} l_{21}) = 0 \\ l_{33} &= \sqrt{a_{33} - (l_{31}^2 + l_{32}^2)} = 1 \end{aligned}$$

c'est-à-dire:

$$L = \begin{pmatrix} \sqrt{2} & 0 & 0 \\ \frac{1}{\sqrt{2}} & \sqrt{\frac{5}{2}} & 0 \\ \sqrt{2} & 0 & 1 \end{pmatrix}.$$

[6]:
```
# scipy.linalg.eigh : eigenvalues for symmetric matric
from scipy.linalg import eigh
# scipy.linalg.eig : eigenvalues of a matrix
from scipy.linalg import eig
```

[7]:
```
epsilon = 2
A = np.array([[epsilon, 1, 2],
              [1, 3, 1],
              [2, 1, 3] ])

# Is A symmetric ?
print(f'max(abs(A-AT)) = {np.max(np.abs(A-A.T))} ')

# What are the eigenvlues of $A$ ? (usually, use eig, but here A is symmetric,␣
 ↪we can use eigh )
lk, v = eigh(A)
print(f'The eigenvalues of A are {lk}')

# Cholesky factorisation: lower : return lower-triangular matrix, A = L L^T
L = cholesky(A, lower=True)
print(f"\n A = L^T L = {L.dot(L.T)}\n")

print (f"L = {L}")
```

```
max(abs(A-AT)) = 0
The eigenvalues of A are [0.4242 2.1873 5.3885]

 A = L^T L = [[2. 1. 2.]
```

```
 [1. 3. 1.]
 [2. 1. 3.]]

L = [[1.4142 0.     0.    ]
 [0.7071 1.5811 0.    ]
 [1.4142 0.     1.    ]]
```

Partie 4

[8]:
```
b = np.array([1,1, 1])

y = np.linalg.solve(L,b)
x = np.linalg.solve(L.T,y)

print(x)

# check the residual of the equation
print(b - A.dot(x))
```

```
[0.4 0.2 0. ]
[ 0.  0. -0.]
```

1.1.4 Problèmes de précision (Exercice 3 série 8)

Les erreurs d'arrondis peuvent causer des différences importantes entre la solution calculée par la méthode d'élimination de Gauss (MEG) et la solution exacte. Cela arrive si le *conditionnement* de la matrice du système est très grand.

La matrice de Hilbert de taille $n \times n$ est une matrice symétrique définie par

$$a_{ij} = \frac{1}{i+j-1}, \qquad i,j = 1,\dots,n$$

On peut construire une matrice de Hilbert de taille n quelconque en utilisant la commande `A = scipy.linalg.hilbert(n)`. Par exemple, pour $n = 4$, on a:

$$A = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \end{bmatrix}$$

On considère les systèmes linéaires $A_n \mathbf{x}_n = \mathbf{b}_n$ ou A_n est la matrice de Hilbert de taille n avec $n = 4, 6, 8, 10, 12, 14, \dots, 20$ tandis que $\mathbf{b}_n$ est choisi de sorte que la solution exacte soit $\mathbf{x}_n = (1, 1, \cdots, 1)^T$.

1. Pour chaque n, calculez le conditionnement de la matrice
2. Résolvez le système linéaire par la factorisation LU et notez $\vec{x}_n^{LU}$ la solution calculée.
3. Dessinez le graphique avec le conditionnement obtenu ainsi que l'erreur rélative $\|\mathbf{x}_n - \mathbf{x}_n^{LU}\|/\|\mathbf{x}_n\|$ (où $\|\cdot\|$ est la norme euclidienne d'un vecteur, $\|\mathbf{x}\| = \sqrt{\mathbf{x}^T \cdot \mathbf{x}}$). Utilisez une échelle logarithmique pour l'axe y.

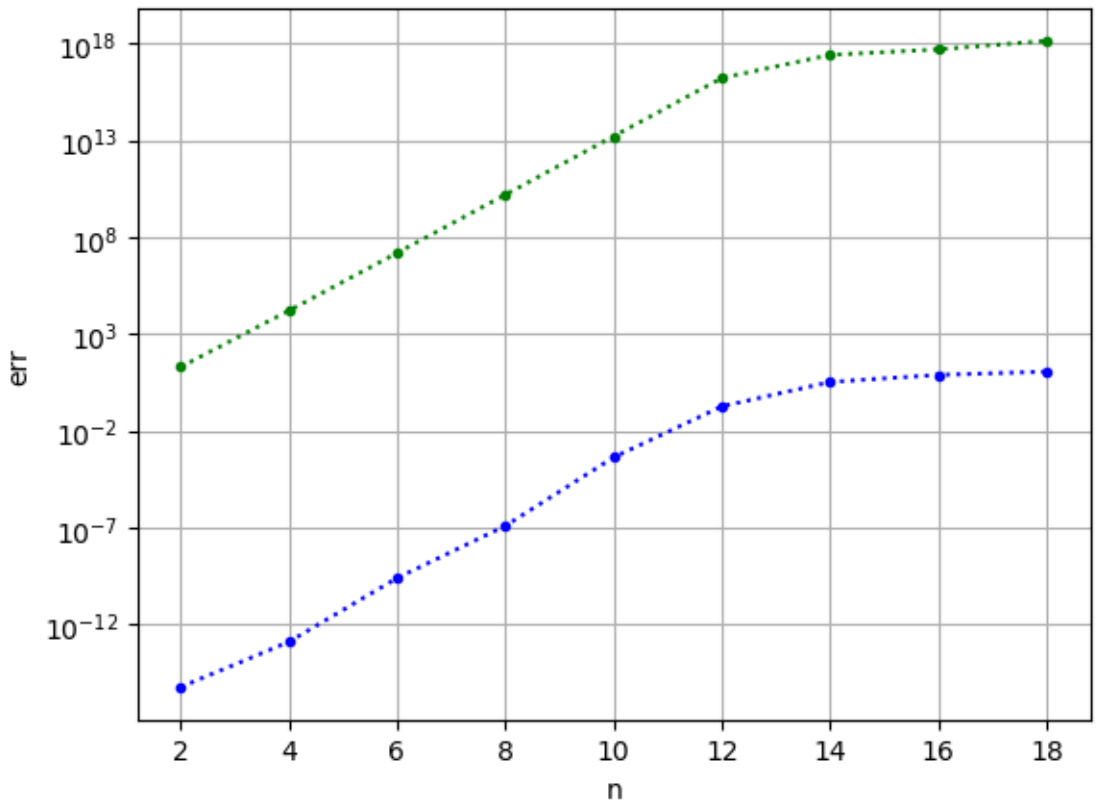
4. Sur le même graphique, reportez le conditionnement de la matrice A, `np.linalg.cond(A)`

Répétez la même chose avec la factorisation de Cholesky `L = cholesky(A, lower=True)` pour $n = 4, 6, 8, 10, 12$. Que se passe-t-il si $n = 14$?

[9]:
```
from scipy.linalg import hilbert
```

[10]:
```
Nrange = range(2,20,2)
err = []
cond = []

for n in Nrange :
    A = hilbert(n)
    P, L, U = lu(A)

    x = np.ones([n,1])

    b = A.dot(x)

    y = np.linalg.solve(L,P.T.dot(b))
    xLU = np.linalg.solve(U,y)

    err.append( np.linalg.norm(x-xLU) / np.linalg.norm(x) )
    cond.append( np.linalg.cond(A) )
```

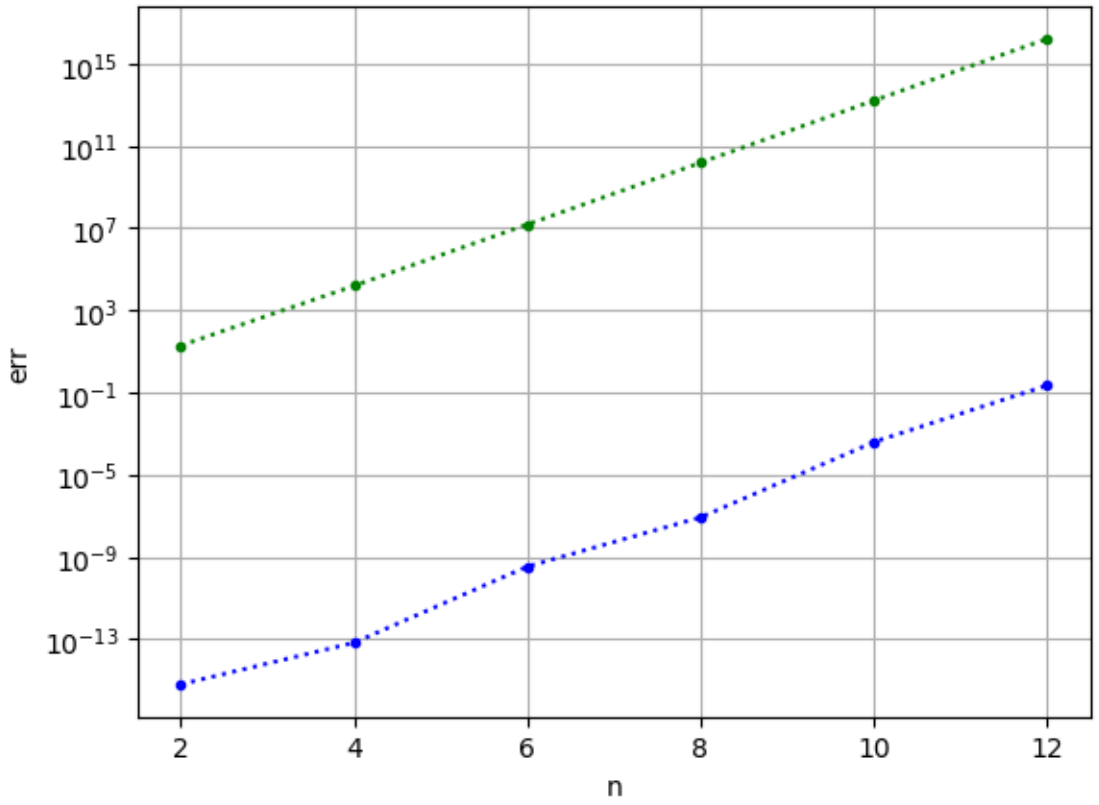
[11]:
```
plt.plot(Nrange, err, 'b:.',Nrange, cond, 'g:.')

plt.xlabel('n'); plt.ylabel('err');
# plt.xscale('log')
plt.yscale('log')
plt.grid(True)
plt.show()
```



[12]:
```
Nrange = range(2,13,2)
err = []
cond = []

for n in Nrange :
    A = hilbert(n)
    L = cholesky(A, lower=True)

    x = np.ones([n,1])

    b = A.dot(x)

    y = np.linalg.solve(L,b)
    xCho = np.linalg.solve(L.T,y)

    err.append( np.linalg.norm(x-xCho) / np.linalg.norm(x) )
    cond.append( np.linalg.cond(A) )
```

[13]:
```
plt.plot(Nrange, err, 'b:.',Nrange, cond, 'g:.')

plt.xlabel('n'); plt.ylabel('err');
```

```
# plt.xscale('log')
plt.yscale('log')
plt.grid(True)
plt.show()
```



[14]:
```
n = 14
A = hilbert(n)
# L = cholesky(A, lower=True)
lk, v = eigh(A)

print(f'lk[0] = {lk[0]:0.5e} , the matrix is not positive definite. Let''s␣
↪verify with the first eigenvector')
```

```
lk[0] = -9.13148e-18 , the matrix is not positive definite. Lets verify with the
first eigenvector
```

La première valeur propre est négative, cela signifie que $(\mathbf{v}, A\mathbf{v}) = (\mathbf{v}, \lambda\mathbf{v}) = \lambda(\mathbf{v}, \mathbf{v}) = \lambda\|\mathbf{v}\|^2 < 0$, où $\mathbf{v}$ est le vecteur propre associé à cette valeur propre λ négative.

Vérifions:

[15]:
```
print ( A.dot(v[0]).T.dot(v[0]) )
```

```
0.007555533559357492
```

Pourquoi ? En vérité, le calcul des valeurs et vecteurs propres de A a aussi un problème d'arrondis à cause du conditionnement. On peut en effet vérifier que le rapport entre les composantes de $\mathbf{v}$ et $A\mathbf{v}$ n'est ni égale à λ, ni à une autre constante :

[16]:
```
print(v[0]/A.dot(v[0]) )
```

```
[ -0.      -0.      -0.       0.0001  -0.0008   0.0052  -0.0292   0.1421
-0.6044  -2.2364   7.0956  18.675  -37.1359
  40.0708]
```

[ ]:

[ ]:

1.2 Méthodes itératives

[17]:
```
# importing libraries used in this notebook
import numpy as np
import matplotlib.pyplot as plt


# scipy.linalg.eig : eigenvalues of a  matrix
from scipy.linalg import eig
# scipy.linalg.lu : LU decomposition
from scipy.linalg import lu
# scipy.linalg.cholesky : Cholesky decomposition
from scipy.linalg import cholesky
# scipy.linalg.hilbert : Hilbert matrix
from scipy.linalg import hilbert

np.set_printoptions(precision=4, suppress=True, linewidth=120)
```

1.3 Méthode de Richardson

A et $\mathbf{b}$ donnés; on cherche à approximer la solution $\mathbf{x}$ de $A\mathbf{x} = \mathbf{b}$.

Soit $\mathbf{x}^{(0)}$ donné, $\mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)}$ pour $k = 0, 1, 2, \ldots$:

$$\begin{aligned} \text{trouvez } \mathbf{z}^{(k)} \text{ tel que} \qquad & P\mathbf{z}^{(k)} = \mathbf{r}^{(k)} \\ \text{choisissez } \alpha_k \qquad & \\ & \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{z}^{(k)} \\ & \mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha_k A\mathbf{z}^{(k)}. \end{aligned}$$

1.3.1 Exemple 1

On considère la matrice A et le terme de droite $\mathbf{b}$ suivants

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} \quad b = \begin{pmatrix} 1 \\ -1 \\ 1 \\ -1 \end{pmatrix}.$$

1.3.2 Méthode de Jacobi

La diagonale de A est

$$D = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 6 & 0 & 0 \\ 0 & 0 & 11 & 0 \\ 0 & 0 & 0 & 16 \end{pmatrix};$$

Prenond $\alpha = 1$ et $P = D$. La méthode de Richardson dans ce cas correspond à la méthode de Jacobi et s'écrit comme suit :

[18]:
```
A = np.array([[1,2,3,4],
              [5,6,7,8],
              [9,10,11,12],
              [13,14,15,16]])
b = np.array([1,-1,1,-1])

# Diagonale de A
P = np.diag(np.diag(A))
alpha = 1

# Initialisation, par exemple
k=0
xk = np.array([1,0,0,0])
```

[19]:
```
# Une itération de Richardson

# résidu
rk = b- A.dot(xk)
# résidu préconditionné :
zk = np.linalg.solve(P,rk)

# correction de la solution, x(k+1) :
xk1 = xk + alpha*zk

# Préparer la prochaine itération
xk = xk1
print (xk)
```

```
[ 1.     -1.     -0.7273 -0.875 ]
```

Si on execute plusieurs fois ces itérations, la méthode ne converge pas. Pourquoi ?

Dans ce cas, la matrice d'itération de la méthode de Richardson est égale à

$$B = D^{-1}(D - A) = I - D^{-1}A = \begin{pmatrix} 0 & -2 & -3 & -4 \\ -5/6 & 0 & -7/6 & -4/3 \\ -9/11 & -10/11 & 0 & -12/11 \\ -13/16 & -14/16 & -15/16 & 0 \end{pmatrix}.$$

[20]:
```
# Jacobi
# the first diag extracts the diagonal of A, the second one builds a diagonal␣
↪matrix starting from a vector
D = np.diag(np.diag(A))

# efficiently computing the Jacobi iteration matrix without explicitely␣
↪computing the inverse of D
Bj = np.linalg.solve(D,(D-A))

# What are the eigenvlues of Bj ?
lk, v = eig(Bj)
print(f'The eigenvalues of Bj are {lk}\n')
print(f'The spectral radius of Bj is {np.max(np.abs(lk)):.2f} > 1, therefore the␣
↪Jacobi method does not converge')
```

```
The eigenvalues of Bj are [-3.9362+0.j  1.9362+0.j  1.    +0.j  1.    +0.j]

The spectral radius of Bj is 3.94 > 1, therefore the Jacobi method does not
converge
```

1.3.3 Méthode de Gauss-Seidel

Prenond $\alpha = 1$ et P la partie triangulaire inférieure de A avec la diagonale. La méthode de Richardson dans ce cas correspond à la méthode de Gauss-Seidel et s'écrit comme suit :

[21]:
```
# Gauss-Seidel
# Return a copy of an array with elements above the k-th diagonal set to zero.
Pgs =  np.tril(A, k=0)

# efficiently computing the Gauss-Siedel iteration matrix without explicitely␣
↪computing the inverse of D
Bgs = np.linalg.solve(Pgs,(Pgs-A))

# What are the eigenvlues of Bgs ?
lk, v = eig(Bgs)
print(f'The eigenvalues of Bgs are {lk}\n')
print(f'The spectral radius of Bgs is {np.max(np.abs(lk)):.2f} > 1, therefore␣
↪the Gauss-Seidel method does not converge')
```

```
The eigenvalues of Bgs are [0.    +0.j 2.0682+0.j 1.    +0.j 1.    +0.j]
```

```
The spectral radius of Bgs is 2.07 > 1, therefore the Gauss-Seidel method does
not converge
```

La méthode de Gauss-Seidel appliqué à cette matrice ne converge pas.

1.3.4 Exercice

Réprenez ces deux méthodes avec la matrice

$$A = \begin{pmatrix} 3 & 2 & 1 & 0 \\ 3 & 4 & 3 & 2 \\ 3 & 2 & 5 & 2 \\ 1 & 2 & 3 & 4 \end{pmatrix}.$$

```
A = np.array([[3,2,1,0],[3,4,3,2],[3,2,5,2],[1,2,3,4]])
```

1. Vérifiez si Jacobi et Gauss-Seidel convergent à l'aide de la matrice d'itérations
2. Calculez les premier 10 itérations de ces méthodes.

1.4 Exemple 2

Considerez la matrice A et le vecteur b suivants

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

Exprimez (sans calculer) P, B, et $\rho(B)$ dans le cas de Jacobi et Gauss-Seidel.

[22]:
```
A = np.array([[3, 2, 1],
              [1, 4, 1],
              [2, 4, 8] ])
```

[23]:
```
# Jacobi
# first diag extract the diagonal of A, the second builds a diagonal matrix␣
 ↪starting from a vector
D = np.diag(np.diag(A))

Bj = np.linalg.solve(D,(D-A))
# What are the eigenvalues of $Bj$ ?
lk, v = eig(Bj)
print(f'The eigenvalues of Bj are {lk}\n')
print(f'The spectral radius of Bj is {np.max(np.abs(lk)):.2f} < 1, therefore␣
 ↪Jacobi converges')
```

```
The eigenvalues of Bj are [-0.7026+0.j  0.4205+0.j  0.2821+0.j]

The spectral radius of Bj is 0.70 < 1, therefore Jacobi converges
```

[24]:
```
# Gauss-Seidel
# Return a copy of an array with elements above the k-th diagonal zeroed.
Pgs =  np.tril(A,k=0)

Bgs = np.linalg.solve(Pgs,(Pgs-A))
# What are the eigenvlues of $Bgs$ ?
lk, v = eig(Bgs)
print(f'The eigenvalues of Bgs are {lk} whose moduli are {np.absolute(lk)} \n')
print(f'The spectral radius of Bgs is {np.max(np.absolute(lk)):.2f} < 1,␣
 ↪therefore Gauss-Seidel converges')
```

```
The eigenvalues of Bgs are [0.    +0.j     0.1667+0.1179j 0.1667-0.1179j] whose
moduli are [0.     0.2041 0.2041]

The spectral radius of Bgs is 0.20 < 1, therefore Gauss-Seidel converges
```

1.5 Exemple 3 - Jacobi et Gauss-Seidel avec relaxation

Nous avons vu que la méthode de Jacobi appliquée à la matrice

$$A = \begin{pmatrix} 3 & 2 & 1 & 0 \\ 3 & 4 & 3 & 2 \\ 3 & 2 & 5 & 2 \\ 1 & 2 & 3 & 4 \end{pmatrix}$$

ne converge pas, alors que celle de Gauss-Seidel converge.

Nous allons utiliser la méthode de Richardson stationnaire avec un paramètre de relaxation α constant pour voir si on peut améliorer la convergence.

On va utiliser la méthode de Richardson stationnaire avec un paramètre de relaxation α constant pour voir si on peut améliorer la convergence.

Partie 1 - Matrices d'itérations La matrice d'itérations pour la méthode de Richardson est $B(\alpha_k) = I - \alpha_k P^{-1} A$. En particulier, pour α constant, nous pouvons faire un graphique du rayon spectral $\rho(B(\alpha))$ dans les cas d'un préconditionneur égale à la diagonale de A (similaire à Jacobi) ou au triangle inférieur de A (similaire à Gauss-Seidel):

Voici comment voir que $B(\alpha_k) = I - \alpha_k P^{-1} A$:

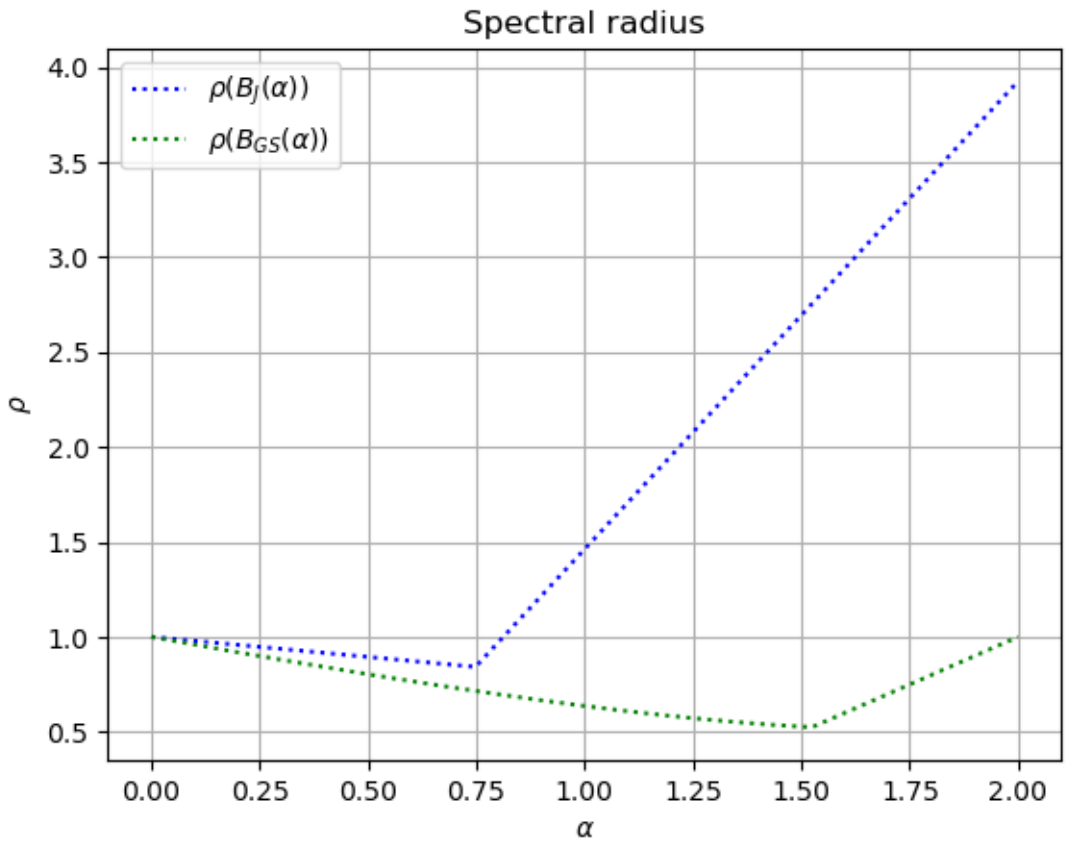
Pour Richardson préconditionné nous avons que

$$\begin{aligned} Px^{(k+1)} &= Px^{(k)} + \alpha_k (b - Ax^{(k)}) \\ Px^{(k+1)} &= \alpha_k b + (P - \alpha_k A) x^{(k)} \\ x^{(k+1)} &= \alpha_k P^{-1} b + P^{-1}(P - \alpha_k A) x^{(k)} \end{aligned}$$

On reconnait la matric d'itération en $B(\alpha_k) = P^{-1}(P - \alpha_k A) = I - \alpha_k P^{-1} A)$

[25]:
```
A = np.array([[3,2,1,0],
              [3,4,3,2],
```

```
              [3,2,5,2],
              [1,2,3,4]])
D = np.diag(np.diag(A))
Pgs =  np.tril(A,k=0)

Alphas = np.linspace(0, 2, 3001)

rhoBj  = []
rhoBgs = []

for alpha in Alphas :
    Bgs = np.linalg.solve(Pgs,(Pgs-alpha*A))
    Bj = np.linalg.solve(D,(D-alpha*A))

    lk, v = eig(Bj)
    rhoBj.append( np.max(np.abs(lk)) )

    lk, v = eig(Bgs)
    rhoBgs.append( np.max(np.abs(lk)) )

plt.plot(Alphas, rhoBj, 'b:', label=r'$\rho(B_J(\alpha))$')
plt.plot(Alphas, rhoBgs, 'g:', label=r'$\rho(B_{GS}(\alpha))$')

plt.xlabel(r'$\alpha$'); plt.ylabel(r'$\rho$');
plt.title('Spectral radius')
plt.grid(True)
plt.legend(['$\\rho(B_J(\\alpha))$','$\\rho(B_{GS}(\\alpha))$'])
plt.show()
```



On constate que:

1. En utilisant $P = D$, Richardson converge pour $\alpha \lessapprox 0.8$ and the optimal one is for $\alpha \approx 0.75$
2. En utilisant P égal au triangle inférieur de A, Richardson converge pour $\alpha \lessapprox 2$; le paramètre optimale est $\alpha \approx 1.5$

Partie 2 - Implémentation de la Méthode de Richardson Définissez une fonction Python qui implémente la méthode de Richardson stationnaire avec la structure suivante:

```
def Richardson(A, b, x0, P, alpha, maxIterations, tolerance) :
    # Stationary Richardson method to approximate the solution of Ax=b
    # INPUT          :
    # x0             : initial guess
    # P              : preconditioner
    # alpha          : constant relaxation parameter
    # maxIterations  : maximum number of iterations
    # tolerance      : tolerance for relative residual
    # OUTPUT         :
    # xk             : approximate solution to the linear system
    # rk             : vector of relative norm of the residuals
```

La méthode de Richardson, comme toute méthode iterative a besoin d'un critère d'arrêt. Dans ce

cas, le plus simple est de poser les critères suivants:

1. On fixe le nombre maximal d'itérations
2. Le résidu relatif est plus petit qu'une tolérance ϵ :

$$\frac{\|\mathbf{b} - A\mathbf{x}^{(k)}\|}{\|\mathbf{b}\|} < \epsilon$$

On s'arrête dès que l'un des ces critières est rempli.

[26]:
```python
def Richardson(A, b, x0, P, alpha, maxIterations, tolerance) :
    # Stationary Richardson method to approximate the solution of Ax=b
    # INPUT          :
    # x0             : initial guess
    # P              : preconditioner
    # alpha          : constant relaxation parameter
    # maxIterations  : maximum number of iterations
    # tolerance      : tolerance for relative residual
    # OUTPUT         :
    # xk             : approximate solution to the linear system
    # rk             : vector of relative norm of the residuals

    # we do not keep track of all the sequence, just the last two entries
    xk = x0
    rk = b - A.dot(xk)

    RelativeResidualNorm = []
    for k in range(maxIterations) :

        zk = np.linalg.solve(P,rk)
        xk = xk + alpha*zk
        rk = b - A.dot(xk)

        # you can verify that this is equivalent to
        # rk = rk - alpha*A.dot(zk)
        RelativeResidualNorm.append(np.linalg.norm(rk)/np.linalg.norm(b))
        if ( RelativeResidualNorm[-1] < tolerance ) :
            print(f'Richardson converged in {k+1} iterations with a relative␣
↪residual of {RelativeResidualNorm[-1]:1.3e}')
            return xk, RelativeResidualNorm

    print(f'Richardson did not converge in {maxIterations} iterations, the␣
↪relative residual is {np.linalg.norm(rk)/np.linalg.norm(b):1.3e}')
    return xk, RelativeResidualNorm
```

Partie 3 - Utilisation de la Méthode de Richardson Utilisez la fonction `Richardson` pour approcher la solution de $A\mathbf{x} = \mathbf{b}$ pour $\mathbf{b} = (0, 1, -1, 1)^T$ avec une tolérance sur le résidu relatif de 10^{-6} et le vecteur nul comme point initial et un nombre maximum d'itérations de 200

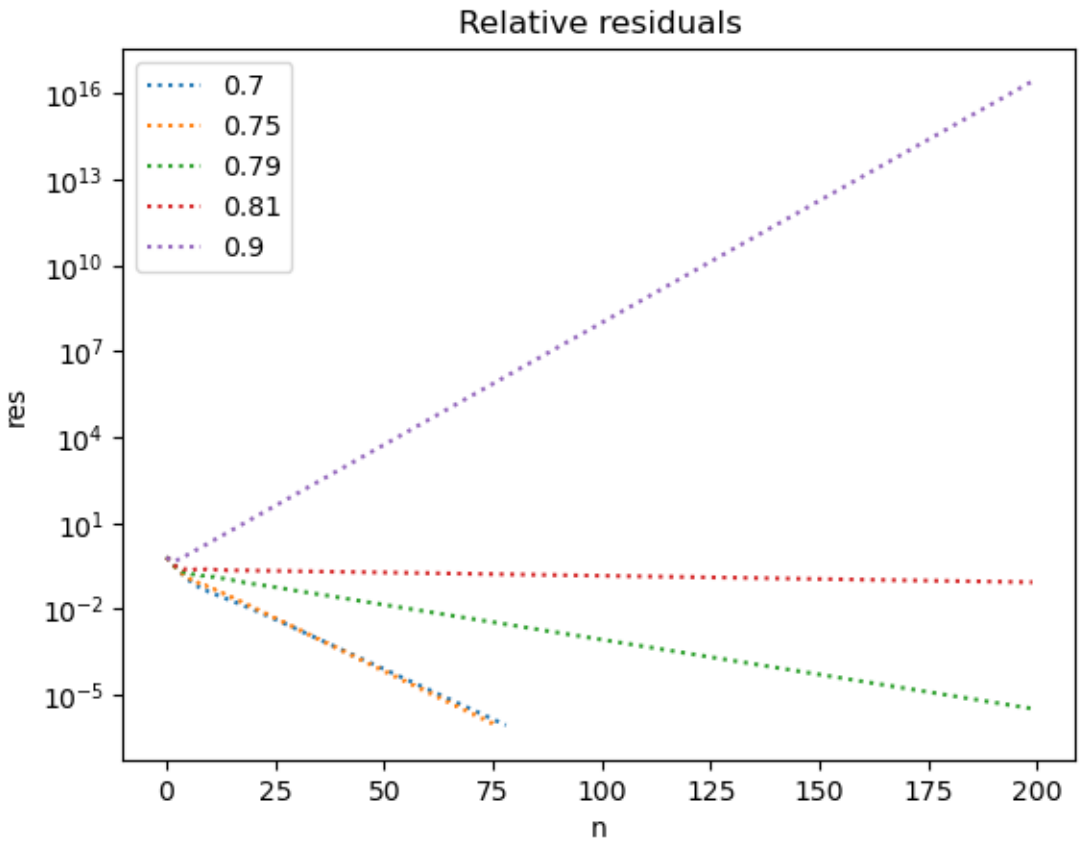
1. En utilisant $P = D$, avec : $\alpha = 0.7, 0.75, 0.79, 0.81, 0.9$
2. En utilisant P égal au triangle inférieur de A, avec : $\alpha = 1, 1.5, 1.95, 2.05$

Comment interprétez-vous ces résultats ?

[27]:
```
b = np.array([0,1,-1,1])
# Define the initial guess
x0 = 0*b # trick to have the right size for x0

tolerance = 1e-6
maxIter = 200
```

[28]:
```
# Jacobi-like preconditioner
P = np.diag(np.diag(A))
legend = []

for alpha in [0.7, 0.75, 0.79,0.81, 0.9] :
    x,relRes = Richardson(A,b,x0,P,alpha,maxIter,tolerance)
    print(relRes[-1])
    plt.plot( relRes, ':')
    legend.append(str(alpha))

plt.legend(legend)

plt.xlabel('n'); plt.ylabel('res');
plt.title('Relative residuals')
plt.yscale('log')
plt.show()
```

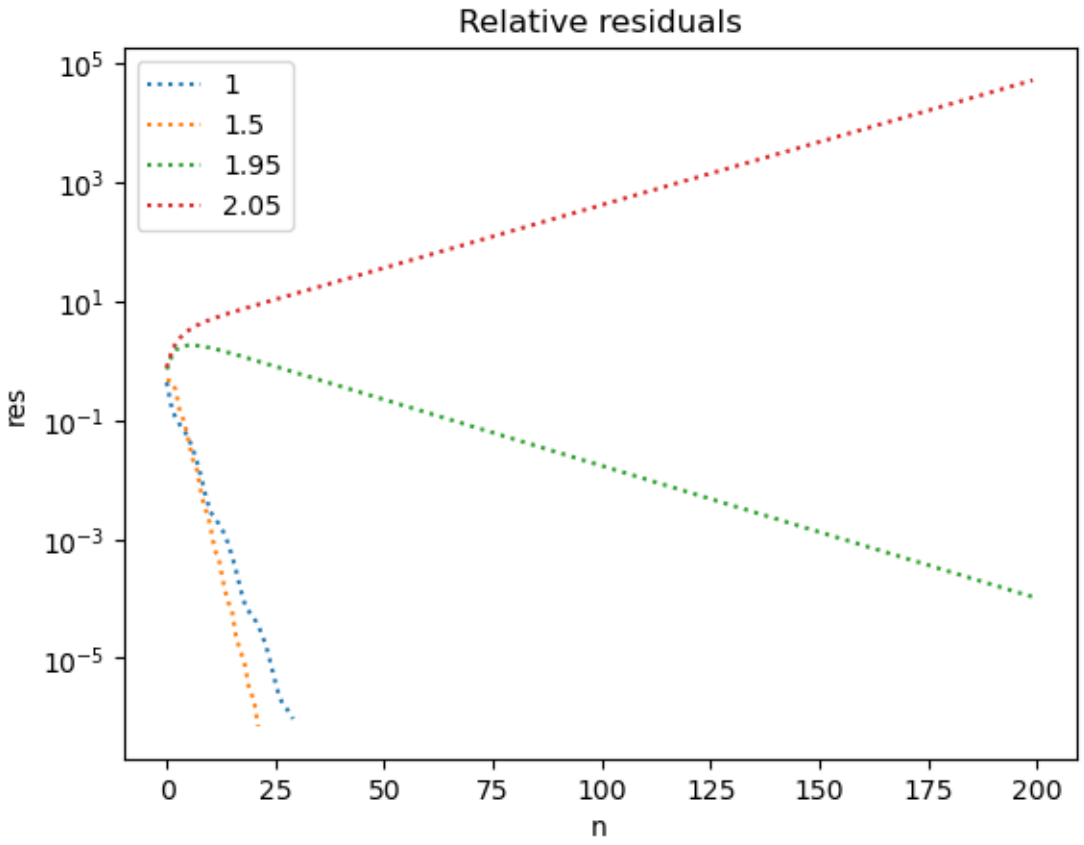
```
Richardson converged in 79 iterations with a relative residual of 8.984e-07
8.984498815385382e-07
Richardson converged in 77 iterations with a relative residual of 7.593e-07
7.592562065358359e-07
Richardson did not converge in 200 iterations, the relative residual is
3.379e-06
3.378918951868571e-06
Richardson did not converge in 200 iterations, the relative residual is
8.699e-02
0.08698843985290847
Richardson did not converge in 200 iterations, the relative residual is
2.576e+16
2.575559592693911e+16
```



[29]:
```
# Gauss-Seidel-like preconditioner
P = np.tril(A,k=0)
legend = []

for alpha in [1, 1.5, 1.95,2.05] :
    x,relRes = Richardson(A,b,x0,P,alpha,maxIter,tolerance)
    print(relRes[-1])
    plt.plot( relRes, ':')
    legend.append(str(alpha))

plt.legend(legend)

plt.xlabel('n'); plt.ylabel('res');
plt.title('Relative residuals')
plt.yscale('log')
plt.show()
```

```
Richardson converged in 30 iterations with a relative residual of 9.521e-07
9.520753485686893e-07
Richardson converged in 22 iterations with a relative residual of 7.144e-07
```

```
7.144137957248418e-07
Richardson did not converge in 200 iterations, the relative residual is
1.071e-04
0.00010708766426426366904
Richardson did not converge in 200 iterations, the relative residual is
5.283e+04
52829.70703107408
```



Partie 4 - Implémentation de la Méthode du Gradient Préconditionné Définissez une fonction Python qui implémente la méthode de Gradient Préconditionné et qui a la structure

```
def PrecGradient(A, b, x0, P, maxIterations, tolerance) :
    # Preconditioned Gradient method to approximate the solution of Ax=b
    # INPUT          :
    # x0             : initial guess
    # P              : preconditioner
    # maxIterations  : maximum number of iterations
    # tolerance      : tolerance for relative residual
    # OUTPUTS        :
    # xk             : approximate solution to the linear system
    # rk             : vector of relative norm of the residuals
```

[30]:
```
def PrecGradient(A, b, x0, P, maxIterations, tolerance) :
    # Preconditioned Gradient method to approximate the solution of Ax=b
    # INPUT          :
    # x0             : initial guess
    # P              : preconditioner
    # maxIterations  : maximum number of iterations
    # tolerance      : tolerance for relative residual
    # OUTPUTS        :
    # xk             : approximate solution to the linear system
    # rk             : vector of relative norm of the residuals

    # we do not keep track of all the sequence, just the last two entries
    xk = x0
    rk = b - A.dot(xk)

    RelativeResidualNorm = []
    for k in range(maxIterations) :

        zk = np.linalg.solve(P,rk)
        Azk = A.dot(zk)

        alphak = zk.dot(rk) / zk.dot(Azk)

        xk = xk + alphak*zk

        rk = b - A.dot(xk)
        # you can verify that this is equivalent to
        # rk = rk - alphak*A.dot(zk)
        RelativeResidualNorm.append(np.linalg.norm(rk)/np.linalg.norm(b))
        if ( RelativeResidualNorm[-1] < tolerance ) :
            print(f'Gradient converged in {k+1} iterations with a relative␣
↪residual of {RelativeResidualNorm[-1]:1.3e}')
            return xk, RelativeResidualNorm

    print(f'Graident did not converge in {maxIterations} iterations, the␣
↪relative residual is {np.linalg.norm(rk)/np.linalg.norm(b):1.3e}')
    return xk, RelativeResidualNorm
```

Partie 5 - Implémentation de la Méthode du Gradient Conjugué Préconditionné
Définissez une fonction Python qui implémente la méthode du Gradient Conjugué Préconditionné avec la structure suivante:

```
def PrecConjugateGradient(A, b, x0, P, maxIterations, tolerance) :
    # Preconditionate Conjugate Gradient method to approximate the solution of Ax=b
    # INPUT          :
    # x0             : initial guess
```

```
    # P             : preconditioner
    # maxIterations : maximum number of iterations
    # tolerance     : tolerance for relative residual
    # OUTPUTS       :
    # xk            : approximate solution to the linear system
    # rk            : vector of relative norm of the residuals
```

[31]:
```
def PrecConjugateGradient(A, b, x0, P, maxIterations, tolerance) :
    # Preconditionate Conjugate Gradient method to approximate the solution of␣
↪Ax=b
    # INPUT         :
    # x0            : initial guess
    # P             : preconditioner
    # maxIterations : maximum number of iterations
    # tolerance     : tolerance for relative residual
    # OUTPUTS       :
    # xk            : approximate solution to the linear system
    # rk            : vector of relative norm of the residuals

    # we do not keep track of all the sequence, just the last two entries
    xk = x0
    rk = b - A.dot(xk)
    zk = np.linalg.solve(P,rk)
    pk = zk

    RelativeResidualNorm = []
    for k in range(maxIterations) :

        Apk = A.dot(pk)
        alphak = pk.dot(rk) / pk.dot(Apk)

        xk = xk + alphak*pk

        rk = rk - alphak*Apk
        # you can verify that this is equivalent to
        # rk = b - A.dot(xk)

        zk = np.linalg.solve(P,rk)

        betak = Apk.dot(zk) / pk.dot(Apk)
        pk = zk - betak*pk


        RelativeResidualNorm.append(np.linalg.norm(rk)/np.linalg.norm(b))
        if ( RelativeResidualNorm[-1] < tolerance ) :
            print(f'Conjugate Gradient converged in {k+1} iterations with a␣
↪relative residual of {RelativeResidualNorm[-1]:1.3e}')
```

```
            return xk, RelativeResidualNorm

    print(f'Conjugate Gradient did not converge in {maxIterations} iterations,␣
↪the relative residual is {np.linalg.norm(rk)/np.linalg.norm(b):1.3e}')
    return xk, RelativeResidualNorm
```

Partie 6 - Utilisation de la Méthode du Gradient Preconditioné et du Gradient Conjugué Préconditionné Utilisez les fonctions `PrecGradient` et `PrecConjugateGradient` pour approcher la solution de $A\mathbf{x} = \mathbf{b}$ pour $\mathbf{b} = (0, 1, -1, 1)^T$ avec une tolérance sur le résidu relatif de 10^{-6} et le vecteur nul comme point initial, et un nombre maximum d'itérations de 200

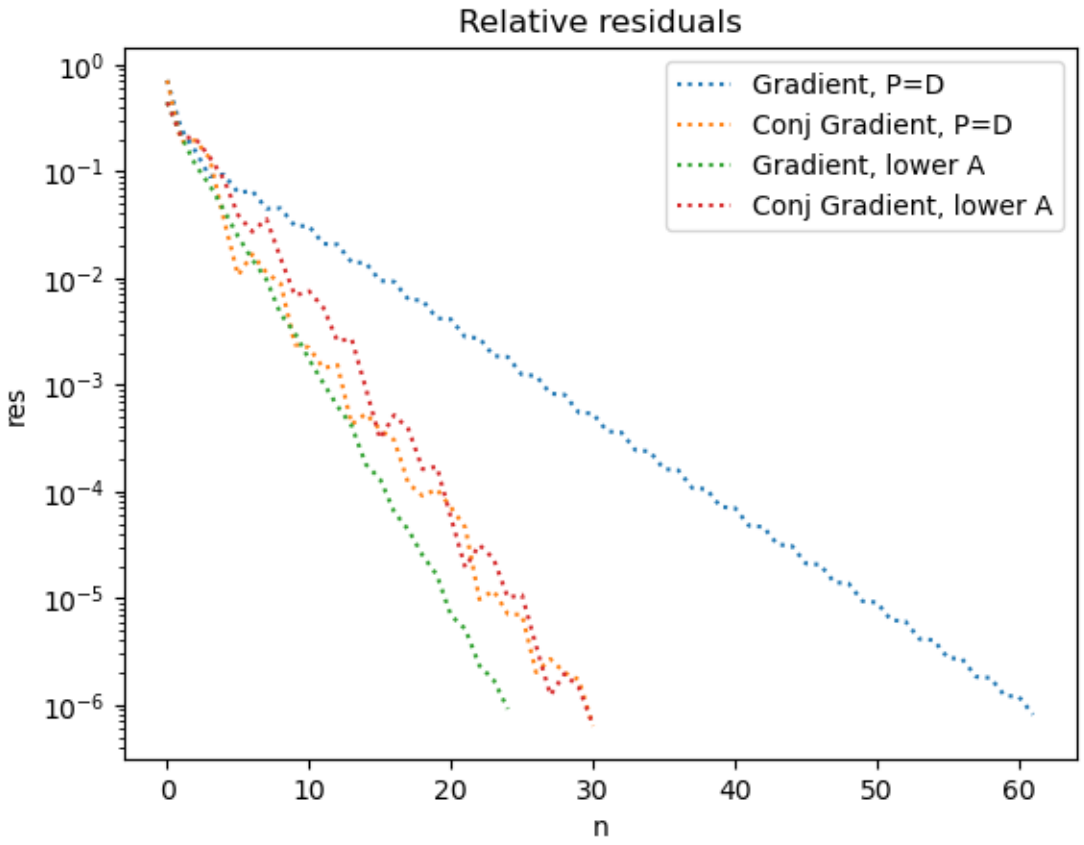
1. En utilisant $P = D$
2. En utilisant P égal au triangle inférieur de A

Comment interprétez-vous ces résultats ?

[32]:
```
legend = []

# Jacobi-like precondtioner
P = np.diag(np.diag(A))

x,relRes = PrecGradient(A,b,x0,P,maxIter,tolerance)
print(relRes[-1])
plt.plot( relRes, ':')
legend.append('Gradient, P=D')

x,relRes = PrecConjugateGradient(A,b,x0,P,maxIter,tolerance)
print(relRes[-1])
plt.plot( relRes, ':')
legend.append('Conj Gradient, P=D')

# Gauss-Seidel-like precondtioner
P = np.tril(A,k=0)

x,relRes = PrecGradient(A,b,x0,P,maxIter,tolerance)
print(relRes[-1])
plt.plot( relRes, ':')
legend.append('Gradient, lower A')

x,relRes = PrecConjugateGradient(A,b,x0,P,maxIter,tolerance)
print(relRes[-1])
plt.plot( relRes, ':')
legend.append('Conj Gradient, lower A')

plt.legend(legend)

plt.xlabel('n'); plt.ylabel('res');
```

```
plt.title('Relative residuals')
plt.yscale('log')
plt.show()
```

```
Gradient converged in 62 iterations with a relative residual of 8.146e-07
8.145842032913786e-07
Conjugate Gradient converged in 31 iterations with a relative residual of
6.323e-07
6.323181670011519e-07
Gradient converged in 25 iterations with a relative residual of 9.177e-07
9.177277863663321e-07
Conjugate Gradient converged in 31 iterations with a relative residual of
6.529e-07
6.529042663207426e-07
```



1.6 Autres Exemples

Répétez les expériences numériques ci-dessous pour approcher les solutions de

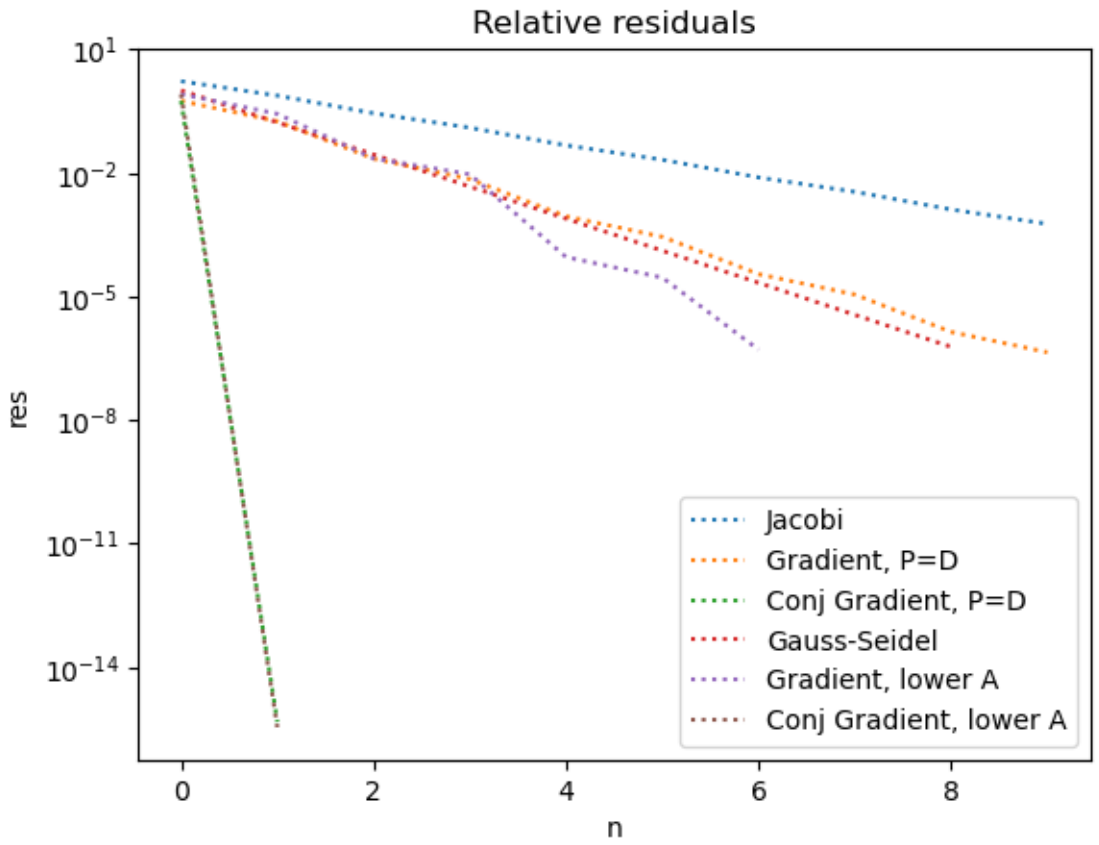
$$\begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad , \quad \begin{pmatrix} 2 & 2 \\ -1 & 3 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \text{et} \quad \begin{pmatrix} 5 & 7 \\ 7 & 10 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

Pour quelles matrices et méthodes vous attendez-vous à une convergence ? Pour la dernière matrice, le résultat n'est pas celui attendu, pourquoi ?

[33]:
```
b = np.array([1,0])
# Define the initial guess
x0 = np.array([1,1])

tolerance = 1e-6
maxIter = 10
A = np.array([[2,1],[1,3]])
# A = np.array([[2,2],[-1,3]])
# A = np.array([[5,7],[7,10]])

legend = []

# To complete: similarly to previous example:
# 1) Jacobi, Gauss-Seidel (simple)
# 2) PrecGradient and PrecConjGradient, with
# both Jacobi-like and Gauss-Seidel-like preconditioning

# 1) Jacobi-like precondtioner
P = np.diag(np.diag(A))

x,relRes = Richardson(A,b,x0,P,1,maxIter,tolerance)
print(relRes[-1])
plt.plot( relRes, ':')
legend.append('Jacobi')

x,relRes = PrecGradient(A,b,x0,P,maxIter,tolerance)
print(relRes[-1])
plt.plot( relRes, ':')
legend.append('Gradient, P=D')

x,relRes = PrecConjugateGradient(A,b,x0,P,maxIter,tolerance)
print(relRes[-1])
plt.plot( relRes, ':')
legend.append('Conj Gradient, P=D')

# Gauss-Seidel-like precondtioner
P = np.tril(A,k=0)

x,relRes = Richardson(A,b,x0,P,1,maxIter,tolerance)
print(relRes[-1])
```

```
plt.plot( relRes, ':')
legend.append('Gauss-Seidel')

x,relRes = PrecGradient(A,b,x0,P,maxIter,tolerance)
print(relRes[-1])
plt.plot( relRes, ':')
legend.append('Gradient, lower A')

x,relRes = PrecConjugateGradient(A,b,x0,P,maxIter,tolerance)
print(relRes[-1])
plt.plot( relRes, ':')
legend.append('Conj Gradient, lower A')

plt.legend(legend)

plt.xlabel('n'); plt.ylabel('res');
plt.title('Relative residuals')
plt.yscale('log')
plt.show()
```

```
Richardson did not converge in 10 iterations, the relative residual is 5.751e-04
0.0005751203645832735
Gradient converged in 10 iterations with a relative residual of 4.401e-07
4.40060404871371e-07
Conjugate Gradient converged in 2 iterations with a relative residual of
4.710e-16
4.710277376051325e-16
Richardson converged in 9 iterations with a relative residual of 5.954e-07
5.953741807340762e-07
Gradient converged in 7 iterations with a relative residual of 5.098e-07
5.098498742819304e-07
Conjugate Gradient converged in 2 iterations with a relative residual of
3.511e-16
3.510833468576701e-16
```



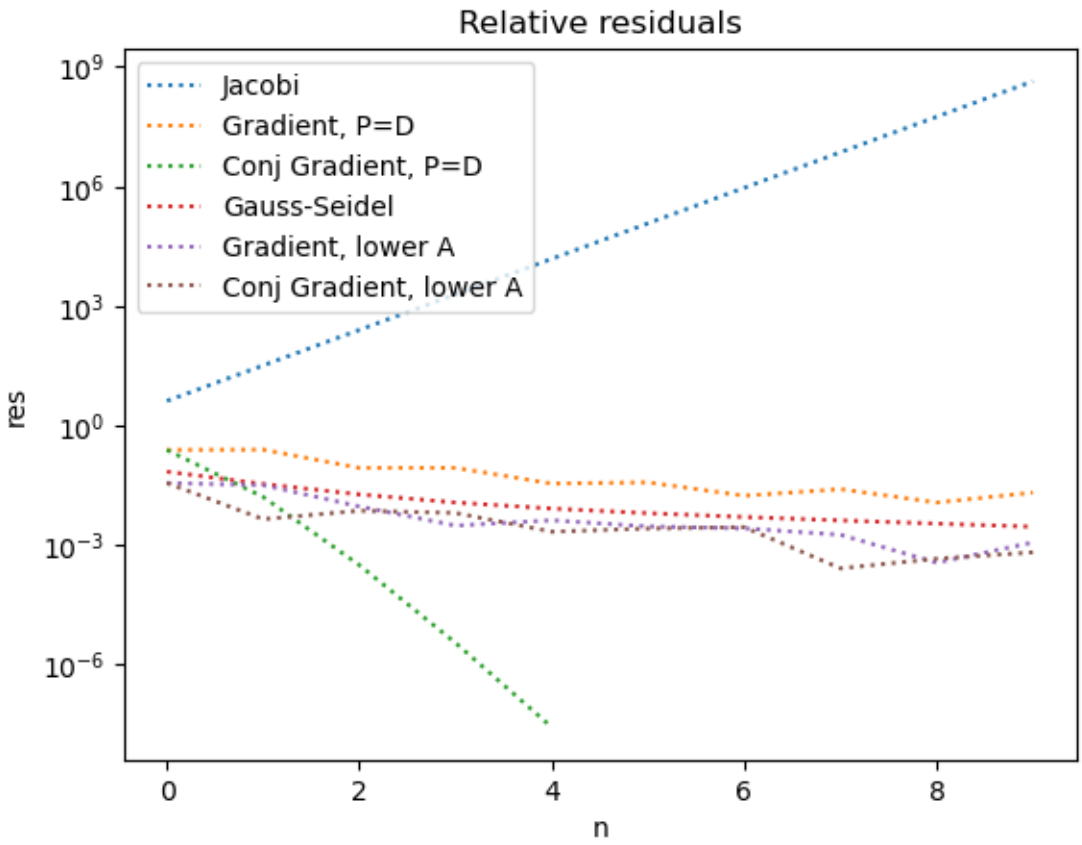
La matrice de Hilbert Peut-on utiliser une méthode de Richardson pour résoudre le problème du mauvais conditionnement de la matrice de Hilbert ?

[34]:
```
n = 10
A = hilbert(n)
xexact = np.ones(n)
b = A.dot(xexact)

# Define the initial guess
x0 = b

tolerance = 1e-6
maxIter = 10

legend = []

# To complete: similarly to previous example:
# 1) Jacobi, Gauss-Seidel (simple)
# 2) PrecGradient and PrecConjGradient, with
# both Jacobi-like and Gauss-Seidel-like preconditioning
```

```
# 1) Jacobi-like precondtioner
P = np.diag(np.diag(A))

x,relRes = Richardson(A,b,x0,P,1,maxIter,tolerance)
print(f'The relatove residual and absolute error are : \t {relRes[-1]:1.3e} \t␣
 ↪{np.linalg.norm(x-xexact):1.3e}\n')
plt.plot( relRes, ':')
legend.append('Jacobi')

x,relRes = PrecGradient(A,b,x0,P,maxIter,tolerance)
print(f'The relatove residual and absolute error are : \t {relRes[-1]:1.3e} \t␣
 ↪{np.linalg.norm(x-xexact):1.3e}\n')
plt.plot( relRes, ':')
legend.append('Gradient, P=D')

x,relRes = PrecConjugateGradient(A,b,x0,P,maxIter,tolerance)
print(f'The relatove residual and absolute error are : \t {relRes[-1]:1.3e} \t␣
 ↪{np.linalg.norm(x-xexact):1.3e}\n')
plt.plot( relRes, ':')
legend.append('Conj Gradient, P=D')

# Gauss-Seidel-like precondtioner
P = np.tril(A,k=0)

x,relRes = Richardson(A,b,x0,P,1,maxIter,tolerance)
print(f'The relatove residual and absolute error are : \t {relRes[-1]:1.3e} \t␣
 ↪{np.linalg.norm(x-xexact):1.3e}\n')
plt.plot( relRes, ':')
legend.append('Gauss-Seidel')

x,relRes = PrecGradient(A,b,x0,P,maxIter,tolerance)
print(f'The relatove residual and absolute error are : \t {relRes[-1]:1.3e} \t␣
 ↪{np.linalg.norm(x-xexact):1.3e}\n')
plt.plot( relRes, ':')
legend.append('Gradient, lower A')

x,relRes = PrecConjugateGradient(A,b,x0,P,maxIter,tolerance)
print(f'The relatove residual and absolute error are : \t {relRes[-1]:1.3e} \t␣
 ↪{np.linalg.norm(x-xexact):1.3e}\n')
plt.plot( relRes, ':')
legend.append('Conj Gradient, lower A')

plt.legend(legend)

plt.xlabel('n'); plt.ylabel('res');
```

```
plt.title('Relative residuals')
plt.yscale('log')
plt.show()
```

```
Richardson did not converge in 10 iterations, the relative residual is 4.305e+08
The relatove residual and absolute error are :   4.305e+08       1.847e+09

Graident did not converge in 10 iterations, the relative residual is 2.066e-02
The relatove residual and absolute error are :   2.066e-02       8.555e-01

Conjugate Gradient converged in 5 iterations with a relative residual of
2.692e-08
The relatove residual and absolute error are :   2.692e-08       2.280e-02

Richardson did not converge in 10 iterations, the relative residual is 2.883e-03
The relatove residual and absolute error are :   2.883e-03       4.243e-01

Graident did not converge in 10 iterations, the relative residual is 1.158e-03
The relatove residual and absolute error are :   1.158e-03       4.149e-01

Conjugate Gradient did not converge in 10 iterations, the relative residual is
6.569e-04
The relatove residual and absolute error are :   6.569e-04       4.205e-01
```


Relative residuals
Jacobi
Gradient, P=D
Conj Gradient, P=D
Gauss-Seidel
Gradient, lower A
Conj Gradient, lower A
res
n
10^{9}
10^{6}
10^{3}
10^{0}
10^{-3}
10^{-6}
0
2
4
6
8

[ ]: