TP3_MethodsIteratives

November 19, 2021

1 TP 3 - Systèmes linéaires

Ci-dessous des éléments de correction du TP 3 concernant la mise en œuvre et les observations menées sur des méthodes itératives pour la résolution de $Ax = b$.

On rappelle la CNS de convergence de telles méthodes pour $A = M - N$ inversible avec M inversible : $\rho(M^{-1}N) < 1$.

La convergence de ces méthodes est sous cette condition au plus géométrique, de raison $\|M^{-1}N\|$ où la norme subordonnée approche arbitrairement la quantité $\rho(M^{-1}N)$ (voir éléments de cours sur le rayon spectral d'une matrice).

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import numpy.linalg as npl
plt.rcParams["figure.figsize"] = (12,10)
```

1.1 Definir la matrice du laplacien / des splines

[ ]:
```
def Lapl(n):
    A = 2*np.diag(np.ones(n)) - np.diag(np.ones(n-1),1) - np.diag(np.
 ↪ones(n-1),-1)
    return A

print(Lapl(10))
```

```
[[ 2. -1.  0.  0.  0.  0.  0.  0.  0.  0.]
 [-1.  2. -1.  0.  0.  0.  0.  0.  0.  0.]
 [ 0. -1.  2. -1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0. -1.  2. -1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0. -1.  2. -1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0. -1.  2. -1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0. -1.  2. -1.  0.  0.]
 [ 0.  0.  0.  0.  0.  0. -1.  2. -1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0. -1.  2. -1.]
 [ 0.  0.  0.  0.  0.  0.  0.  0. -1.  2.]]
```

[ ]:
```
def Spli(n):
    A = 4*np.diag(np.ones(n)) + np.diag(np.ones(n-1),1) + np.diag(np.
 ↪ones(n-1),-1)
    return A

print(Spli(10))
```

```
[[4. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 4. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 4. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 4. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 4. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 4. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 4. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 4. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 4. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 4.]]
```

1.2 Methode de Jacobi

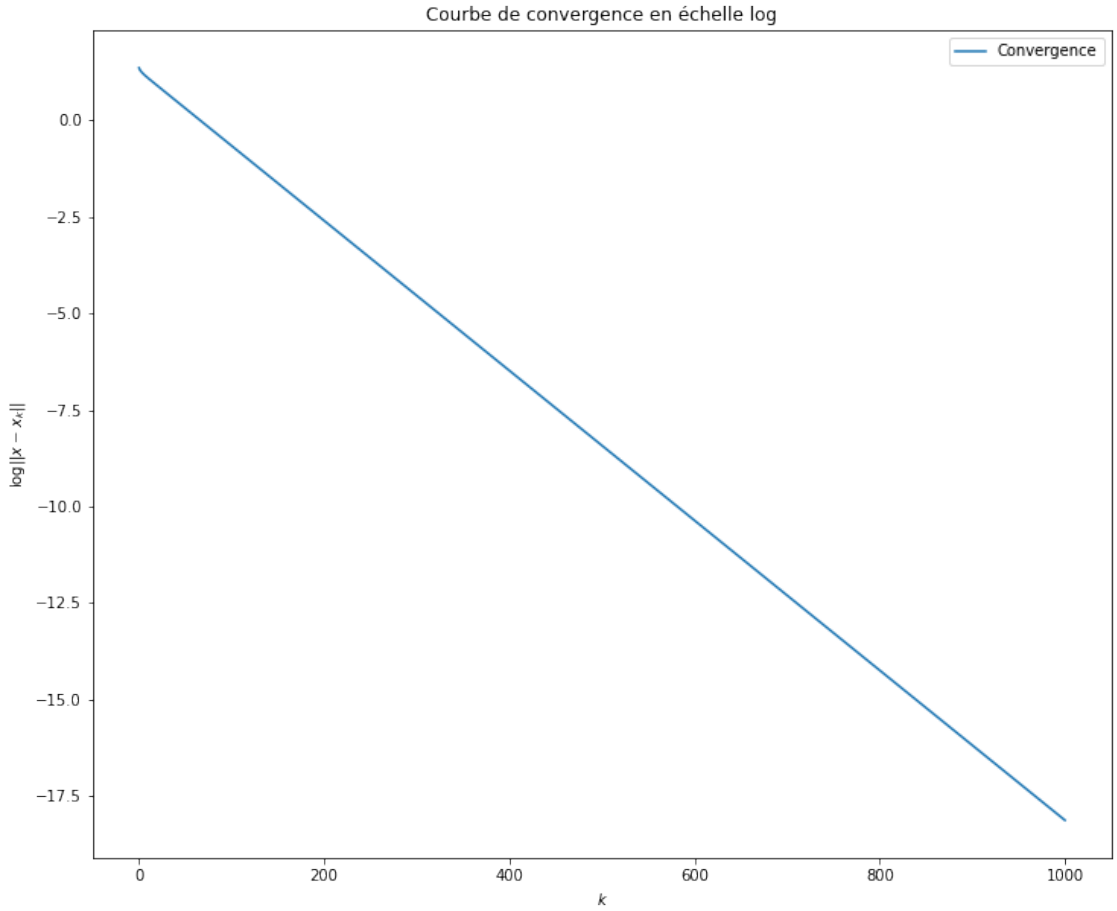
Le choix est $M = D$ et $N = M - A$ où D désigne la matrice diagonale extraite de A.

$M^{-1}N = I - D^{-1}A$

[ ]:
```
# Version rudimentaire sans stockage des itérées

def Jacobi(A,b,x0,Nmax,tol):
    M = np.diag(np.diag(A))
    N = M-A
    x = x0
    k = 0
    err = npl.norm(np.dot(A,x)-b)
    while (k<Nmax and err>tol):
        k = k+1
        x = npl.solve(M,np.dot(N,x)+b) ## Partie optimisable en temps de calcul␣
 ↪(résoudre un système diagonal !)
        err = npl.norm(np.dot(A,x)-b)
    return x

n = 15
A = Lapl(n)
x = np.ones(n)
b = np.dot(A,x)
Jacobi(A,b,np.zeros(n),1000,1e-10)
```

[ ]:
```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

[ ]:
```
# Version avec stockage des itérées dans une liste de array permettant␣
↪l'analyse de convergence

def Jacobi(A,b,x0,Nmax,tol):
    M = np.diag(np.diag(A))
    N = M-A
    x = x0
    ListX = [x]
    k = 0
    err = npl.norm(np.dot(A,x)-b)
    while (k<Nmax and err>tol):
        k = k+1
        x = npl.solve(M,np.dot(N,x)+b)
        err = npl.norm(np.dot(A,x)-b)
        ListX.append(x)
    return ListX

n = 15
A = Lapl(n)
x = np.ones(n)
b = np.dot(A,x)
ListX = Jacobi(A,b,np.zeros(n),1000,1e-10)
ListE = [npl.norm(x-y) for y in ListX] # calcul de l'erreur (en norme 2)␣
↪utilisant la solution exacte x

plt.plot(np.log(ListE))
plt.legend(['Convergence'])
plt.xlabel('$k$')
plt.ylabel('$\log||x-x_k||$')
plt.title('Courbe de convergence en échelle log')
plt.show()
```



La courbe de convergence témoigne d'un comportement géométrique: $||x - x_k|| \leq C\alpha^k$ avec α que nous allons identifier par régression linéaire et comparer au facteur théorique $\rho(M^{-1}N)$ (celui-ci peut être calculé de façon numérique par la routine 'eigvals' ou à la main à partir des valeurs propres de la matrice du laplacien/celle des splines)

[ ]:
```
a1,a2 = np.polyfit(np.arange(0,len(ListE),1),np.log(ListE),1)
print('Facteur de convergence effectif:  ',np.exp(a1))

M = np.diag(np.diag(A))
S = npl.eigvals(npl.inv(M) @ (M-A))
rho = max(np.abs(S))
# rho = max([1-2*np.sin(np.pi/(2*n+2))**(2),np.abs(1-2*np.sin(n*np.pi/
 ↪(2*n+2))**(2))]) # Formule exacte alternative
print('Facteur de convergence theorique: ',rho)
```

```
Facteur de convergence effectif:   0.9807840112640708
Facteur de convergence theorique:  0.9807852804032312
```

1.3 Méthode de Jacobi relaxée

Le choix est $M = \frac{1}{\omega}D$ et $N = M - A$ où D désigne la matrice diagonale extraite de A.

$M^{-1}N = I - \omega D^{-1}A$

[ ]:
```
# Version avec stockage des itérées dans une liste de array permettant l'analyse de convergence

def JacobiRelaxee(A,b,omega,x0,Nmax,tol):
    M = np.diag(np.diag(A))*1/omega
    N = M-A
    x = x0
    ListX = [x]
    k = 0
    err = npl.norm(np.dot(A,x)-b)
    while (k<Nmax and err>tol):
        k = k+1
        x = npl.solve(M,np.dot(N,x)+b) ## Partie optimisable en temps de calcul (résoudre un système diagonal !)
        err = npl.norm(np.dot(A,x)-b)
        ListX.append(x)
    return ListX


n = 15
A = Lapl(n)
x = np.ones(n)
b = np.dot(A,x)

omega = 1
ListX = JacobiRelaxee(A,b,omega,np.zeros(n),100,1e-16)
print('La méthode est d''abord validée avec $\omega=1$ qui correspond à la méthode de Jacobi précédente')
ListX[-1]
```

```
La méthode est dabord validée avec $\omega=1$ qui correspond à la méthode de
Jacobi précédente
```

[ ]:
```
array([0.96408019, 0.93089462, 0.89770905, 0.87230992, 0.84691078,
       0.83316487, 0.81941896, 0.81941896, 0.81941896, 0.83316487,
       0.84691078, 0.87230992, 0.89770905, 0.93089462, 0.96408019])
```

La CNS de convergence pour cette méthode dépend bien sûr du facteur ω qu'il faut choisir avec précaution.

Pour ce faire, sur les exemples considérés, on peut identifier le rayon spectral de la matrice d'itération $M^{-1}N$ en fonction de ω.

Les valeurs propres de A_{lapl} sont les $4\sin^2\left(\frac{k\pi}{2(n+1)}\right)$ pour $1 \leq k \leq n$ et donc on trouve les valeurs propres de $M^{-1}N$: $1 - 2\omega\sin^2\left(\frac{k\pi}{2(n+1)}\right)$ pour $1 \leq k \leq n$.

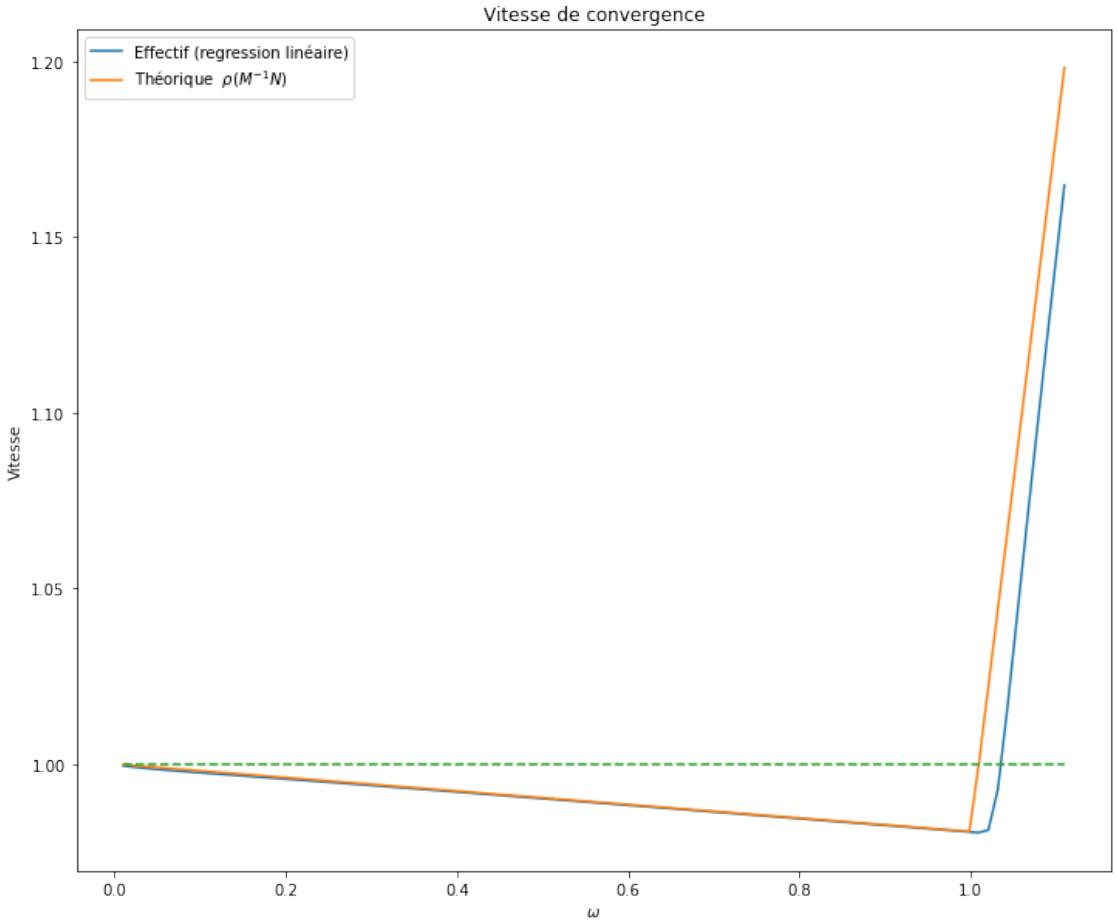
On en déduit facilement que $\rho(M^{-1}N) = \max(\left|1 - 2\omega\sin^2\left(\frac{\pi}{2(n+1)}\right)\right|, \left|1 - 2\omega\sin^2\left(\frac{n\pi}{2(n+1)}\right)\right|)$.

En particulier, $\rho(M^{-1}N) < 1$ si et seulement si ω est dans l'intervalle ouvert $]0, \sin^{-2}\left(\frac{n\pi}{2(n+1)}\right)[$, dont la borne supérieure est légèrement supérieure à 1 lorsque n est grand.

Sans cette analyse, on peut tester au hasard des valeurs de ω est constater à regret que de nombreuses valeurs occasionnent la divergence de la méthode, avec souvent des ***inf*** *ou* ***nan***

[ ]:
```
omeg_max = np.sin(n*np.pi/(2*n+2))**(-2)
print('Omega maximum: ',omeg_max)
ListOmega = np.linspace(0.01,omeg_max+0.1,100)

Facteur_Effectif = []
Facteur_Theorique = []

for omega in ListOmega:
    x0 = np.zeros(n)
    ListX = JacobiRelaxee(A,b,omega,x0,100,1e-16)
    ListE = [npl.norm(x-y)+1e-20 for y in ListX] # Ici, un coeff artificiel de très petite taille est ajouté pour rendre plus robuste les regressions en cas de calcul log(0)
    a1,a2 = np.polyfit(np.arange(0,len(ListE),1),np.log(ListE),1)
    Facteur_Effectif.append(np.exp(a1))
    Facteur_Theorique.append(max([1-2*omega*np.sin(np.pi/(2*n+2))**(2),2*omega*np.sin(n*np.pi/(2*n+2))**(2)-1]))

plt.plot(ListOmega,Facteur_Effectif,ListOmega,Facteur_Theorique)
plt.plot(ListOmega,np.ones_like(ListOmega),'--')
plt.legend(['Effectif (regression linéaire)','Théorique  'r'$\rho(M^{-1}N)$'])
plt.xlabel('$\omega$')
plt.ylabel('Vitesse')
plt.title('Vitesse de convergence')
plt.show()
```

```
Omega maximum:  1.0097005565352637
```



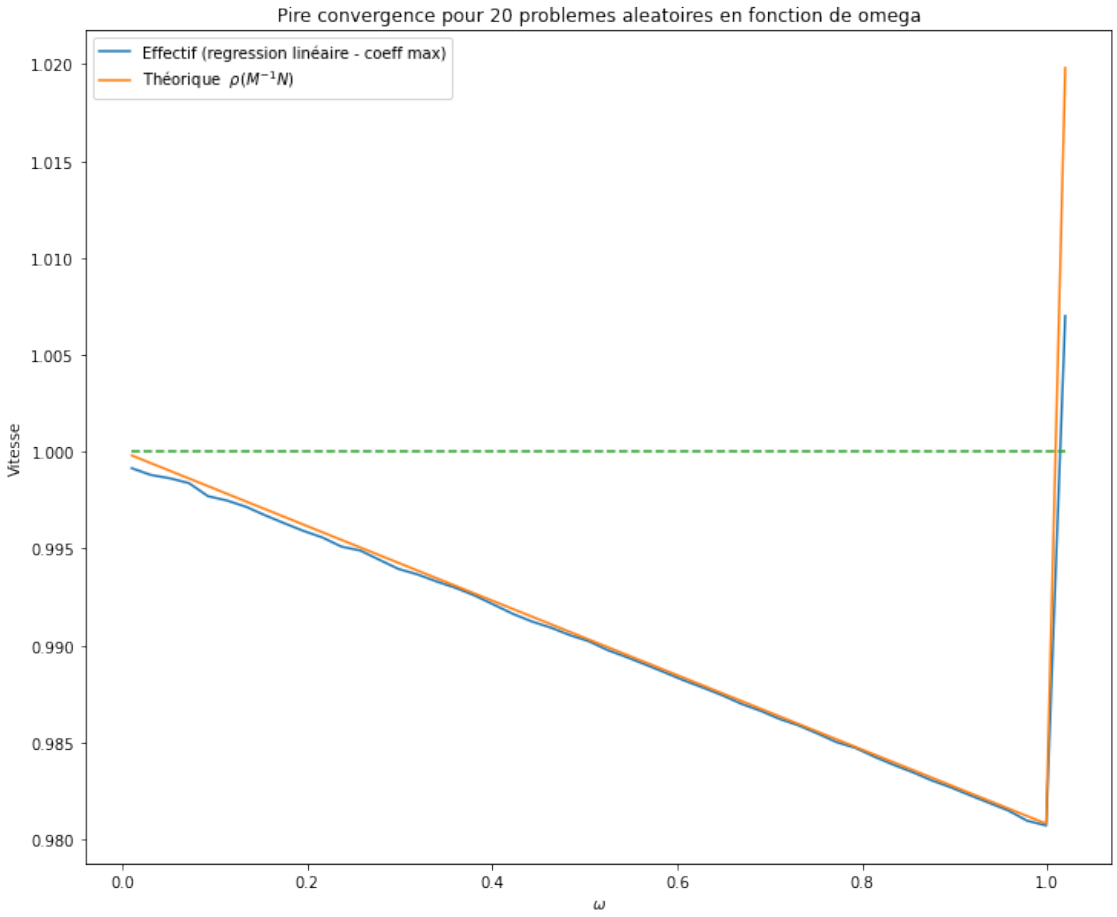
1.3.1 Utilisation de plusieurs données aléatoires

[ ]:

```
n = 15
A = Lapl(n)
m = 20

omeg_max = np.sin(n*np.pi/(2*n+2))**(-2)
print('Omega maximum: ',omeg_max)
ListOmega = np.linspace(0.01,omeg_max+0.01,50)

Facteur_Effectif = []
Facteur_Theorique = []

for omega in ListOmega:
    Effectif = []
    for k in range(m):
```

```
        x = np.random.rand(n,1)
        b = np.dot(A,x)
        x0 = np.zeros(n)
        ListX = JacobiRelaxee(A,b,omega,x0,100,1e-16)
        ListE = [npl.norm(x-y)+1e-20 for y in ListX]
        a1,a2 = np.polyfit(np.arange(0,len(ListE),1),np.log(ListE),1)
        Effectif.append(np.exp(a1))
    Facteur_Effectif.append(max(Effectif))
    Facteur_Theorique.append(max([1-2*omega*np.sin(np.pi/
↪(2*n+2))**(2),2*omega*np.sin(n*np.pi/(2*n+2))**(2)-1]))

plt.plot(ListOmega,Facteur_Effectif,ListOmega,Facteur_Theorique)
plt.plot(ListOmega,np.ones_like(ListOmega),'--')
plt.legend(['Effectif (regression linéaire - coeff max)','Théorique ␣
↪'r'$\rho(M^{-1}N)$'])
plt.title('Pire convergence pour '+str(m)+' problemes aleatoires en fonction de␣
↪omega')
plt.xlabel('$\omega$')
plt.ylabel('Vitesse')
plt.show()
```

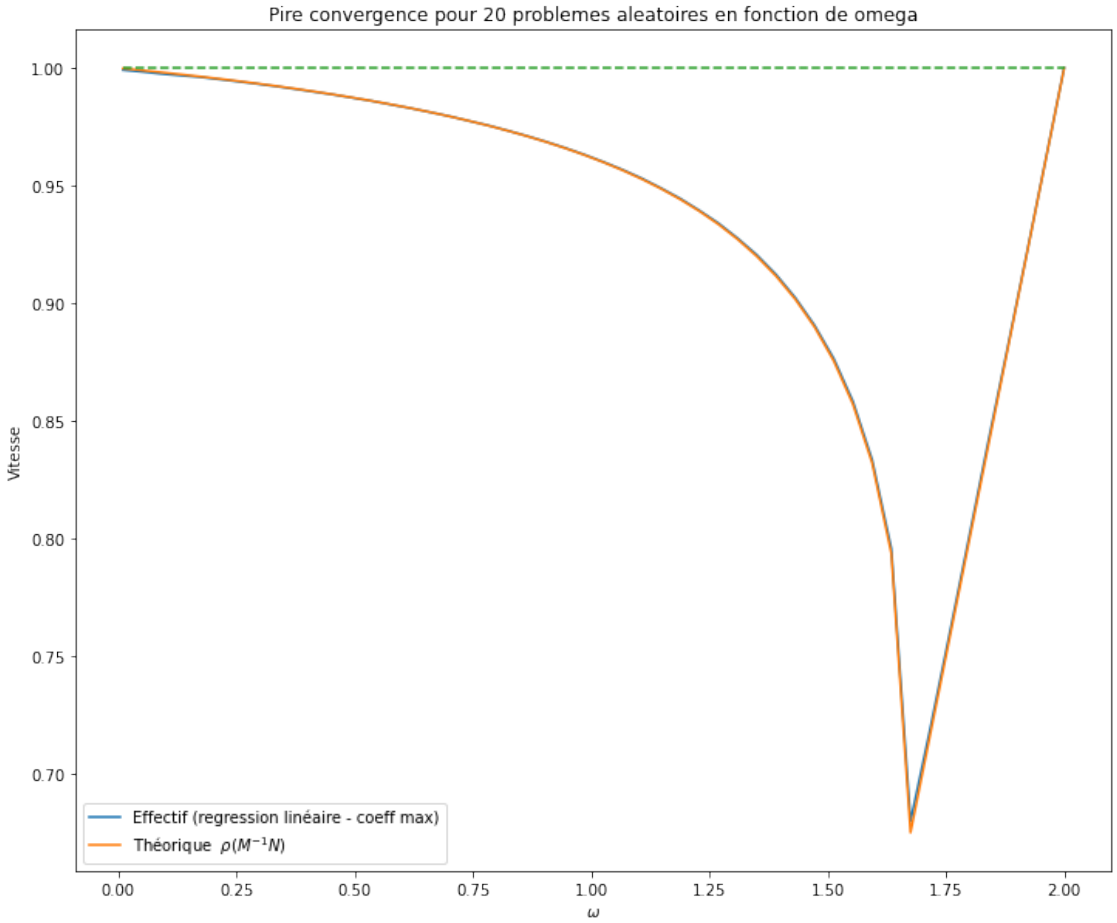
```
Omega maximum:  1.0097005565352637
```



1.4 Methode de Gauss-Seidel relaxée

[ ]:
```
def GaussSeidelRelaxee(A,b,omega,x0,Nmax,tol):
    M = np.diag(np.diag(A))*1/omega + np.tril(A) - np.diag(np.diag(A))
    N = M-A
    x = x0
    ListX = [x]
    k = 0
    err = npl.norm(np.dot(A,x)-b)
    while (k<Nmax and err>tol):
        k = k+1
        x = npl.solve(M,np.dot(N,x)+b) ## Partie optimisable en temps de calcul␣
↪(résoudre un système triangulaire !)
        err = npl.norm(np.dot(A,x)-b)
        ListX.append(x)
    return ListX
```

```
n = 15
A = Lapl(n)
m = 20

ListOmega = np.linspace(0.01,2,50)

Facteur_Effectif = []
Facteur_Theorique = []

for omega in ListOmega:
    Effectif = []
    for k in range(m):
        x = np.random.rand(n,1)
        b = np.dot(A,x)
        x0 = np.zeros(n)
        ListX = GaussSeidelRelaxee(A,b,omega,x0,100,1e-16)
        ListE = [npl.norm(x-y)+1e-20 for y in ListX]
        a1,a2 = np.polyfit(np.arange(0,len(ListE),1),np.log(ListE),1)
        Effectif.append(np.exp(a1))
    Facteur_Effectif.append(max(Effectif))

    # Pour s'épargner le calcul des valeurs propres dans ce cas, on utilise une␣
 ↪routine numérique eiguals
    M = np.diag(np.diag(A))*1/omega + np.tril(A) - np.diag(np.diag(A))
    S = npl.eigvals(npl.inv(M) @ (M-A))
    rho = max(np.abs(S))
    Facteur_Theorique.append(rho)

plt.plot(ListOmega,Facteur_Effectif,ListOmega,Facteur_Theorique)
plt.plot(ListOmega,np.ones_like(ListOmega),'--')
plt.legend(['Effectif (regression linéaire - coeff max)','Théorique ␣
 ↪'r'$\rho(M^{-1}N)$'])
plt.title('Pire convergence pour '+str(m)+' problemes aleatoires en fonction de␣
 ↪omega')
plt.xlabel('$\omega$')
plt.ylabel('Vitesse')
plt.show()
```



On pourra jeter un œil dans le livre de Ciarlet (Introduction à l'analyse matricielle et à l'optimisation) p.106 et 107 où une formule explicite du rayon spectral est donnée.

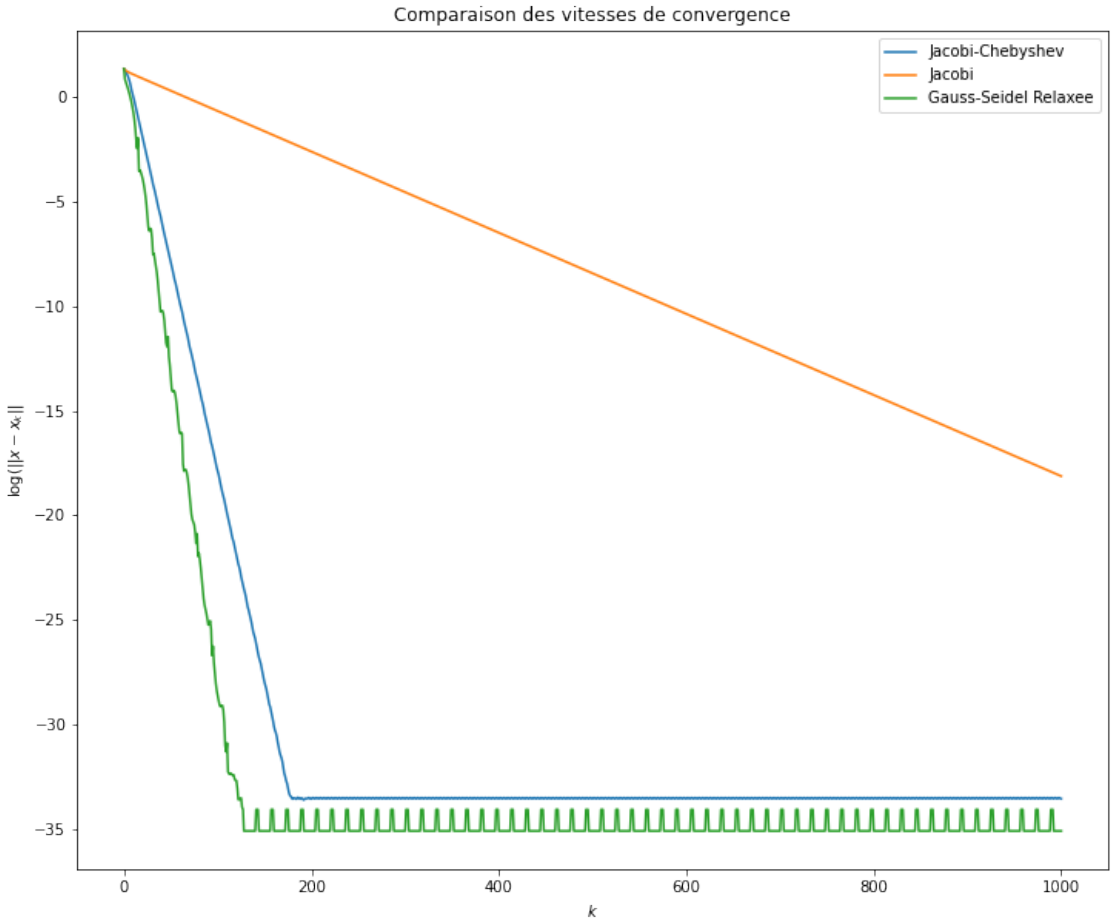
1.5 Jacobi-Chebyshev

[ ]:
```
def JacobiChebyshev(A,b,alpha,beta,x0,Nmax,tol):
    D = np.diag(np.diag(A))
    lam = 2/(2-alpha-beta)
    kap = (1-alpha)/(1-beta)
    tau2 = (kap-1)**2/(kap+1)**2
    sig = 2
    x = x0
    ListX = [x]
    xp = x - lam*npl.solve(D,np.dot(A,x)-b)
    ListX.append(xp)
    k = 1
    err = npl.norm(np.dot(A,xp)-b)
```

```
    while (k<Nmax and err>tol):
        k = k+1
        sig = 4/(4-tau2*sig)
        xpp = sig*(xp-lam*npl.solve(D,np.dot(A,xp)-b)) + (1-sig)*x
        err = npl.norm(np.dot(A,xpp)-b)
        ListX.append(xpp)
        x  = xp
        xp = xpp
    return ListX

n = 15
A = Lapl(n)
x = np.ones(n)
b = np.dot(A,x)


M = np.diag(np.diag(A))
S = npl.eigvals(npl.inv(M) @ (M-A))

print('Valeurs propres de M^{-1}N: \n',np.sort(S))
```

```
Valeurs propres de M^{-1}N:
 [-9.80785280e-01 -9.23879533e-01 -8.31469612e-01 -7.07106781e-01
 -5.55570233e-01 -3.82683432e-01 -1.95090322e-01 -1.56078444e-16
  1.95090322e-01  3.82683432e-01  5.55570233e-01  7.07106781e-01
  8.31469612e-01  9.23879533e-01  9.80785280e-01]
```

[ ]:
```
ListX = JacobiChebyshev(A,b,-0.9808,0.9808,np.zeros(n),1000,1e-20)
ListE = [npl.norm(x-y) for y in ListX]

ListXJ = Jacobi(A,b,np.zeros(n),1000,1e-20)
ListEJ = [npl.norm(x-y) for y in ListXJ]

ListXGSO = GaussSeidelRelaxee(A,b,1.74,np.zeros(n),1000,1e-20)
ListEGSO = [npl.norm(x-y) for y in ListXGSO]

plt.plot(np.log(ListE))
plt.plot(np.log(ListEJ))
plt.plot(np.log(ListEGSO))
plt.legend(['Jacobi-Chebyshev','Jacobi','Gauss-Seidel Relaxee'])
plt.xlabel('$k$')
plt.ylabel('$\log(||x-x_k||$')
plt.title('Comparaison des vitesses de convergence')
plt.show()
```


Comparaison des vitesses de convergence
Jacobi-Chebyshev
Jacobi
Gauss-Seidel Relaxee
0
−5
−10
−15
−20
−25
−30
−35
$\log(\|x - x_*\|)$
0
200
400
600
800
1000
k

[ ]: