Sol-Assignment 4 - Systemes linéaires

May 7, 2025

1 Assignment 4: Méthodes itératives pour systèmes linéaires

Avant de voir le code disponible de ce test et avant de commencer à rédiger vos réponses, prenez le temps de réfléchir à la manière dont vous pouvez organiser le travail.

- **Pensez à quelles parties du test utiliseront des fonctions écrites/résultats obtenus dans les parties précédentes.**
- **Réfléchissez à la structure de votre code (vous pouvez faire un brouillon sur papier).**
- **Réfléchissez aux sections du cours qui vous seront utiles pour l'analyse de vos résultats.**

On considère les systèmes linéaires paramétrisés de dimension n de la forme $A_\beta \mathbf{x} = \mathbf{b}_\beta$, où

$$A_\beta = \begin{pmatrix} f(1)g(0) & f(\beta)g(0) & f(\beta^2)g(0) & 0 & \cdots & 0 \\ f(\beta)g(1) & f(1)g(1) & f(\beta)g(1) & f(\beta^2)g(1) & 0 & 0 \\ f(\beta^2)g(2) & f(\beta)g(2) & f(1)g(2) & f(\beta)g(2) & f(\beta^2)g(2) & 0 \\ \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\ 0 & \cdots & 0 & f(\beta^2)g(n-1) & f(\beta)g(n-1) & f(1)g(n-1) \end{pmatrix} \in \mathbb{R}^{n\times n};$$

$$\mathbf{b}_\beta \in \mathbb{R}^n \quad \text{tel que} \quad \mathbf{b}_\beta = \mathbf{A}_\beta\, \mathbf{1} \quad \text{(c-à-d } \mathbf{x} = \mathbf{1} \text{ est la solution exacte).}$$

On choisi $f(x) = 1 - \cos\left(\frac{\pi x}{2}\right)$, $g(x) = \frac{1}{n}\cosh\left(\frac{\pi x}{n}\right)$ et $\beta \in [0, 1]$.

Pour une taille `n` et une valeur `beta` de β données, on peut calculer A et b en utilisant la fonction `matrix(n, beta)` définie plus loin.

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

[2]:
```
def matrix(n, beta) :
    # Returns A_\beta and b given in the exercice.

    f = lambda x: 1.0 - np.cos(np.pi/2*x)
    g = lambda x: np.cosh(np.pi * x / n) / n
    vec = np.array([g(k) for k in range(n)])
```

```
    # diagonals of the matrix
    d = [f(1)*vec,
         f(beta**1)*vec[:-1],
         f(beta**2)*vec[:-2]]
    A = np.diag(d[0],k=0) + \
        np.diag(d[1],k=1) + np.diag(d[1],k=-1) + \
        np.diag(d[2],k=2) + np.diag(d[2],k=-2)

    b = A.dot(np.ones(n))

    return A,b
```

1.1 Partie 1

Ècrire une fonction `Richardson`, qui implemente la méthode de Richardson stationnaire préconditionné. La fonction doit avoir la structure suivante:

```
def Richardson(A, b, x0, P, alpha, max_iter, tol) :
    # Stationary Richardson method to approximate the solution of Ax=b
    #
    # INPUTS:
    # A               : system matrix
    # b               : system vector
    # x0              : initial guess
    # P               : preconditioner
    # alpha           : constant relaxation parameter
    # max_iter        : maximum number of iterations
    # tol             : tolerance on the relative residual
    #
    # OUTPUTS
    # xk              : approximate solution to the linear system
    # rk              : list of relative norms of the residuals
```

[3]:
```
def Richardson(A, b, x0, P, alpha, max_iter, tol) :
    # Stationary Richardson method to approximate the solution of Ax=b
    #
    # INPUTS:
    # A               : system matrix
    # b               : system vector
    # x0              : initial guess
    # P               : preconditioner
    # alpha           : constant relaxation parameter
    # max_iter        : maximum number of iterations
    # tol             : tolerance on the relative residual
    #
    # OUTPUTS
    # xk              : approximate solution to the linear system
```

```
    # res_norm       : list of relative norms of the residuals


    xk = np.copy(x0)
    rk = b - A.dot(xk)
    res_norm = [np.linalg.norm(rk)]
    rel_res = res_norm[-1] / np.linalg.norm(b)

    k = 0
    while k < max_iter and rel_res > tol:

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

        res_norm.append(np.linalg.norm(rk))
        rel_res = res_norm[-1] / np.linalg.norm(b)

        k += 1

    if k >= max_iter and rel_res >= tol:
        print(f'Richardson did not converge in {max_iter} iterations; '
              f'the relative residual is {rel_res:.3e}')
    else:
        print(f'Richardson method converged in {k+1} iterations '
                  f'with a relative residual of {rel_res:.3e}')

    return xk, res_norm
```

1.2 Partie 2

On sait que si la matrice A_β est diagonale dominante stricte par ligne, alors la méthode de Gauss-Seidel converge. Établir si c'est le cas lorsqu'on fixe $\beta = 0.4$, pour une taille `n=20`.

Dans le cas affirmatif, calculer la solution du système par la commande `Richardson` en choisissant les méthodes de Jacobi et Gauss-Seidel. On fixe une tolérance `tol` $= 10^{-10}$ et le point de départ $\mathbf{x}^{(0)} = \mathbf{0}$.

Est-ce que les résultats obtenus sont en accord avec la théorie ? (*Commentaire 1*)

Aide: *la somme des valeurs (en valeur absolue) dans toutes les lignes d'une matrice `M` peut etre calculée de la manière suivante: `v = np.sum(np.abs(M), axis=1)`. L'output `v` est un `numpy.array`, dont longeur est égale au nombre des lignes de la matrice `M`.*

[4]:
```
n = 20
beta = 0.4
[A, b] = matrix(n, beta)
```

```python
# check if A is diagonally dominant
x1 = np.sum(np.abs(A - np.diag(np.diag(A))), axis=1)
x2 = np.abs(np.diag(A))
print(f'A is diagonally dominant: {all(x1 < x2)}')

x0 = np.zeros(n)
tol = 1e-10
nmax = 500

print('\nJacobi preconditioner')
P = np.diag(np.diag(A))  # Jacobi preconditioner
x, iters = Richardson(A, b, x0, P, 1, nmax, tol)
err = np.linalg.norm(x - np.ones(n))
print(f'The error is: {err:.3e}')

print('\nGauss-Seidel preconditioner')
P = np.tril(A, 0)  # Gauss-Seidel preconditioner
x, iters = Richardson(A, b, x0, P, 1, nmax, tol)
err = np.linalg.norm(x - np.ones(n))
print(f'The error is: {err:.3e}')
```

```
A is diagonally dominant: True

Jacobi preconditioner
Richardson method converged in 26 iterations with a relative residual of
9.899e-11
The error is: 8.369e-10

Gauss-Seidel preconditioner
Richardson method converged in 14 iterations with a relative residual of
3.384e-11
The error is: 6.420e-10
```

Commentaire 1 Dans le cas $\beta = 0.4$, on vérifie numeriquement que la matrice A_β est à diagonale dominante strict. Donc on déduit que les méthodes de Jacobi et Gauss-Seidel convergent. En fait, on observe que Jacobi converge en 26 itérations et Gauss-Seidel en 14 itérations.

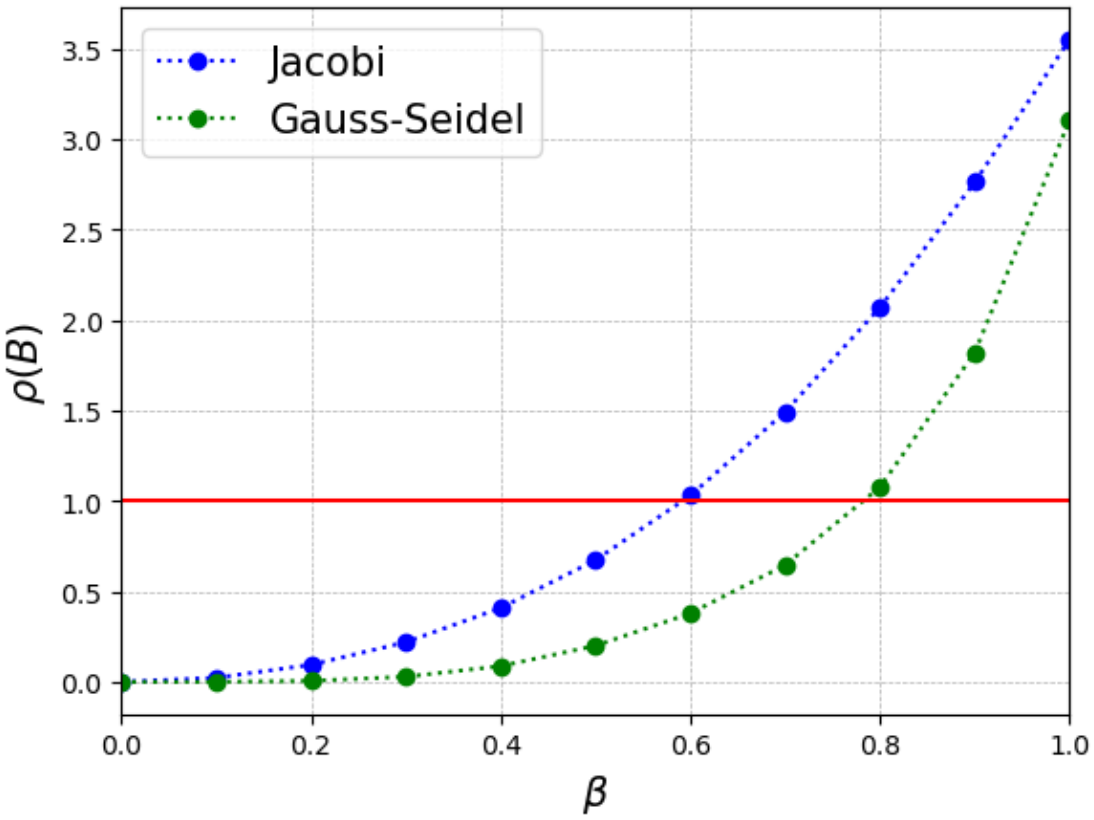
1.3 Partie 3

Tracer le graphe du rayon spectral $\rho(\beta) = \max |\lambda_{B(\beta)}|$ (utiliser la commande `np.linalg.eig` pour les valeures propes) des matrices d'itération de Jacobi $B_J(\beta)$ et de Gauss-Seidel $B_{GS}(\beta)$ en fonction de β, pour $\beta = 0, 0.1, 0.2, \cdots, 1$.

D'après ce graphe, que peut-on dire sur la convergence des deux méthodes en fonction du paramètre β? Quelle méthode utiliseriez-vous pour résoudre le système dans le cas $\beta = 0.7$? Pourquoi ? Est-ce qu'on s'attend au même nombre d'itérations que dans le cas $\beta = 0.4$? (*Commentaire 2*)

[5]:
```
n = 20
betas = np.linspace(0,1,11)

rho_j, rho_gs = [], []
for beta in betas :
    [A, b] = matrix(n,beta)

    # Jacobi
    Pj = np.diag(np.diag(A))
    Bj = np.eye(n) - np.linalg.solve(Pj, A)
    rho_j.append(np.max(np.abs(np.linalg.eig(Bj)[0])))

    # Gauss-Siedel
    Pgs = np.tril(A,0)
    Bgs = np.eye(n) - np.linalg.solve(Pgs,A)
    rho_gs.append(np.max(np.abs(np.linalg.eig(Bgs)[0])))
```

[6]:
```
plt.figure()

plt.plot(betas, rho_j, 'b:o', label='Jacobi')
plt.plot(betas, rho_gs, 'g:o', label='Gauss-Seidel')

plt.plot(np.linspace(0,1,50), np.ones(50), 'r-');

plt.xlabel(r'$\beta$', fontsize=15)
plt.ylabel(r'$\rho(B)$', fontsize=15)
plt.xlim([0,1])
plt.legend(fontsize=15)
plt.grid(linestyle='--', linewidth=.5)
plt.show()
```



Commentaire 2 On voit que le rayon spectral de la matrice d'itération de la méthode de Gauss-Seidel est toujours plus petit que celui de la matrice d'itération de la méthode de Jacobi, ce qui signifie que la méthode de Gauss-Seidel converge plus vite. En outre, on voit que la méthode de Jacobi converge seulement pour β plus petit qu'une valeur proche à 0.6, alors que celle de Gauss-Seidel demeure convergente jusqu'à $\beta = 0.8$ à peu près.

Si $\beta = 0.7$, on doit forcemment utiliser la méthode de Gauss-Seidel pour garantir la convergence des itérations. Avec reférence au cas $\beta = 0.4$, on s'attend un nombre plus élevé d'itérations, car le rayon spectral de la matrice B est plus haut.

1.4 Partie 4

Maintenant, on considère la matrice A_β et le vecteur $\mathbf{b}_\beta$ que l'on obtient pour `n=100`. Dans ce cas, pour $\beta = 0.25$ et $\beta = 0.50$, tracer et comparer les graphes `semilogy` de la norme du résidu $\mathbf{r}^{(k)}$ en fonction du nombre d'itérations pour la méthode de Jacobi.

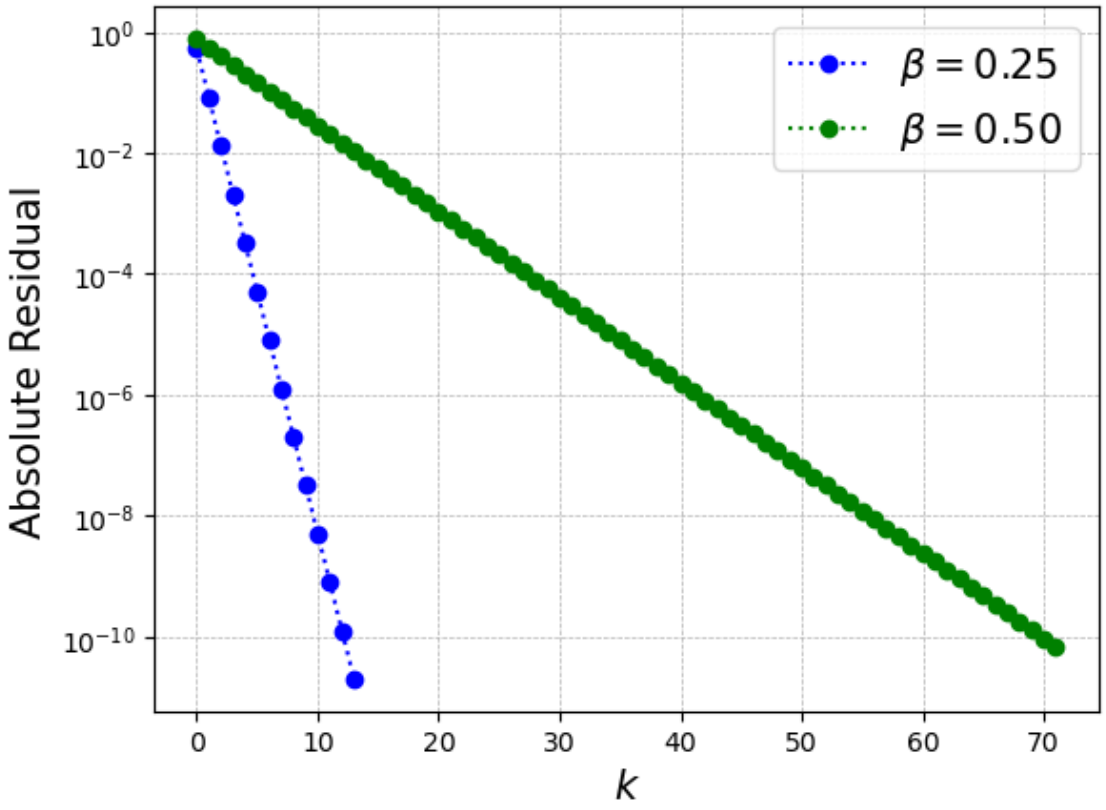
Donner un commentaire, en considerant les pentes des courbes obtenues. Est-ce qu'on s'attend de résultats similaires pour la méthode de Gauss-Seidel ? (*Commentaire 3*)

[7]:
```
n = 100
x0 = np.zeros(n)
tol = 1e-10
```

```
nmax = 500

beta = 0.25
[A, b] = matrix(n,beta)
P = np.diag(np.diag(A))
_, res_beta25 = Richardson(A, b, x0, P, 1, nmax, tol)

beta = 0.5
[A, b] = matrix(n,beta)
P = np.diag(np.diag(A))
_, res_beta50 = Richardson(A, b, x0, P, 1, nmax, tol)
```

```
Richardson method converged in 14 iterations with a relative residual of
3.789e-11
Richardson method converged in 72 iterations with a relative residual of
8.737e-11
```

[8]:
```
plt.figure()

plt.semilogy(res_beta25, 'b:o', label=r'$\beta = 0.25$')
plt.semilogy(res_beta50, 'g:o', label=r'$\beta = 0.50$')

plt.legend(fontsize=15)
plt.xlabel(r'$k$', fontsize=15)
plt.ylabel('Absolute Residual', fontsize=15);
plt.grid(linestyle='--', linewidth=.5)
plt.show()
```



Commentaire 3 On voit très clairement que, si on prend $\beta = 0.50$, la convergence est beaucoup plus lente (la pente du graphe est inférieure). Ceci nous confirme que le rayon spectral de la matrice d'itération de la méthode de Jacobi croît lorsque β augmente. Les conséquences sont une vitesse de convergence réduite et, pour β plus grand qu'une certaine valeur critique (que l'on peut estimer être ~ 0.6), la non-convergence de la méthode.

Sur la base de resultats obtenus dans la *Partie 3*, on sait que le rayon spectral de la matrice d'iteration de Gauss-Seidel a la meme comportement (par rapport a β) que ce de la matrice de Jacobi, mais c'est plus petit. Par conseuquent, avec Gauss-Seidel on aurait encore une convergence plus lente si $\beta = 0.50$, mais le nombre d'iterations serait inferieur que dans le cas de Jacobi.

1.5 Partie 5

D'aprés la théorie vue en classe, est-il possible d'ameliorer la convergence de la méthode de Richardson stationnaire avec le préconditionneur de Jacobi ? Si oui, comment ?

Verifiez votre hypothése numeriquement, en choisissant `n=100` et $\beta = 0.5$. Donner un commentaire sur les résultats obtenus. (*Commentaire 4*)

[9]:
```
beta = 0.5
[A, b] = matrix(n,beta)
```

```
P = np.diag(np.diag(A))
lambdas, _ = np.linalg.eig(np.linalg.solve(P, A))
alpha_opt = 2 / (lambdas[0] + lambdas[-1])
_ = Richardson(A, b, x0, P, alpha_opt, nmax, tol)
```

```
Richardson method converged in 34 iterations with a relative residual of
9.405e-11
```

Commentaire 4 La matrice A est symètrique et definie positive. Comme le préconditionneur de Jacobi est aussi symètrique et defini positif, on peut mimimiser le rayon spectral de la matrice d'itération en choisissant

$$\alpha = \frac{2}{\lambda_{\min}(P^{-1}A) + \lambda_{\max}(P^{-1}A)} .$$

Les résultats numériques confirment les attentes théoriques, car la méthode maintenant converge en seulement 34 itérations, au lieu qu'en 72 itérations.

1.6 Partie 6

La convergence peut être encore ameliorée si on considère des méthodes de Richardson non-stationnaires, dans lesquels la valeur du paramètre α change au cours des itérations. L'example le plus simple est donné par la méthode du *Gradient Preconditionee*.

Écrire une nouvelle fonction `PrecGradient`, en modifiant de manière appropriée la fonction `Richardson` créée dans la *Partie 1*. Vérifier la convergence de la méthode en prenant `n=100` et $\beta = 0.5$, et en considerant: * le cas pas preconditionné (c-à-d `P = np.eye(n)`); * le préconditionneur de Jacobi.

Donner un commentaire sur la base des résultats obtenus. (*Commentaire 5*)

[10]:
```
def PrecGradient(A, b, x0, P, max_iter, tol) :
    # Preconditioned gradient method to approximate the solution of Ax=b
    #
    # INPUTS:
    # A             : system matrix
    # b             : system vector
    # x0            : initial guess
    # P             : preconditioner
    # max_iter      : maximum number of iterations
    # tol           : tolerance on the relative residual
    #
    # OUTPUTS
    # xk            : approximate solution to the linear system
    # res_norm      : list of relative norms of the residuals

    xk = np.copy(x0)
    rk = b - A.dot(xk)
    res_norm = [np.linalg.norm(rk)]
    rel_res = res_norm[-1] / np.linalg.norm(b)
```

```
    k = 0
    while k < max_iter and rel_res > tol:

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

        alpha = zk.dot(rk) / zk.dot(A.dot(zk))  # line to be added !!

        xk += alpha*zk
        rk = b - A.dot(xk)

        res_norm.append(np.linalg.norm(rk))
        rel_res = res_norm[-1] / np.linalg.norm(b)

        k += 1

    if k >= max_iter and rel_res >= tol:
        print(f'Precondioned Gradient did not converge in {max_iter} iterations;
↪ '
              f'the relative residual is {rel_res:.3e}')
    else:
        print(f'Precondioned Gradient method converged in {k+1} iterations '
              f'with a relative residual of {rel_res:.3e}')

    return xk, res_norm
```

[11]:
```
n = 100
beta = 0.5
[A, b] = matrix(n, beta)

x0 = np.zeros(n)
tol = 1e-10
nmax = 500

print("Without preconditioning")
P = np.eye(n)
_ = PrecGradient(A, b, x0, P, nmax, tol)
print("\n")

print("Using Jacobi preconditioner")
P = np.diag(np.diag(A))
_ = PrecGradient(A, b, x0, P, nmax, tol)
```

```
Without preconditioning
Precondioned Gradient method converged in 228 iterations with a relative
residual of 9.796e-11
```

```
Using Jacobi preconditioner
Precondioned Gradient method converged in 27 iterations with a relative residual
of 7.290e-11
```

Commentaire 5 Comme on pouvait s'y attendre, l'utilise de la méthode du Gradient permet de réduire encore le nombre d'itérations (27 au lieu de 34), par rapport au cas stationnaire avec la valeur de α optimale.

Pourtant, on remarque que l'utilise d'un préconditionneur est vital, car le nombre d'itérations explose dans le cas non preconditionné.