Université de Strasbourg
Cours Calcul Scientifique 2
Année 2019-2020

TP 2 : Méthodes GMRES et Gradient Conjugué

L'objectif de ce TP est de programmer les méthodes GMRES et Gradient Conjugué. Commencez par importer les modules python :

```
import numpy as np
import scipy as sp
import scipy.sparse as spsp
import scipy.sparse.linalg as spsplin
```

Pour la multiplication matricielle, utilisez la commande `@` : elle est compatible avec les structures creuses de scipy.

Partie 1. (GMRES)

1. Programmer une fonction `Arnoldi( A, V, H)`, qui à partir des matrices $A \in M_n(\mathbb{R})$, `V` $\in M_{n,k}(\mathbb{R})$ (contenant la base de l'espace de Krylov $K_{k-1}(A;r_0)$) et `H` $\in M_{k,k-1}(\mathbb{R})$, renvoie les matrices `Vp` $\in M_{n,k+1}(\mathbb{R})$ (contenant la base de l'espace de Krylov $K_k(A;r_0)$) et `Hp` $\in M_{k+1,k}(\mathbb{R})$. Nous rappelons que nous avons les relations suivantes :
$$w_k = Av_{k-1} - \sum_{j\leqslant k-1} \langle Av_{k-1},v_j\rangle v_j, \quad v_k = \frac{w_k}{\|w_k\|}, \quad Av_{k-1} = \|w_k\|\, v_k + \sum_{j\leqslant k-1} \langle Av_{k-1},v_j\rangle v_j$$
$$= h_{k,k-1}v_k + \sum_{j\leqslant k-1} h_{j,k-1}v_j.$$
en notant $V = [v_0,\dots,v_{k-1}] \in M_{n,k}(\mathbb{R})$. Au vu de ces relations, commencez par calculer les $h_{j,k-1}$ et le w_k, puis calculer v_k et $h_{k,k-1}$.
2. Programmer une fonction `gmres(A, b, xexact)` qui, à partir d'une matrice $A \in M_n(\mathbb{R})$, un vecteur $b \in \mathbb{R}^n$ et une solution exacte `xexact` $\in \mathbb{R}^n$ (si disponible), renvoie la solution x du système linéaire obtenue par l'algorithme GMRES, ainsi que la liste des erreurs relatives $\|x_{\text{exact}} - x_k\| \,/\, \|x_{\text{exact}}\|$, et la liste de la norme des résidus (relatifs) $\|r_k\| \,/\, \|r_0\|$.
Nous rappelons ci-dessous l'algorithme:

   x_0 donné
   $r_0 = b - Ax_0,$
   $v_0 = r_0/\|r_0\|$, $V_0 = [v_0]$, $\hat{H}_{-1} = []$
   Tant que condition non satisfaite
       calcul de $V_{k+1},\hat{H}_k$ à partir de $V_k,\hat{H}_{k-1}$
       $Q_kR_k = \hat{H}_k$
       $(R_k)_{0\leqslant i,j\leqslant k}\, y = \|r_0\|\,(Q_k^T e_0)_{0\leqslant j\leqslant k}$
       $x_{k+1} = V_k y$

   Vous pourrez utiliser les fonctions `sp.linalg.qr` pour la factorisation QR et `np.linalg.solve` pour la résolution du système triangulaire.
3. Tester votre programme sur la matrice suivante :
```
A = np.diag(2*np.ones(n)) + 0.5 * np.random.rand(n, n)/np.sqrt(n)
```
Prendre $x_0 = 0$. Afficher l'erreur et le résidu en fonction des itérations (en échelle logarithmique). On pourra calculer la solution exacte avec une fonction numpy.

4. (Facultatif) Programmer une fonction `gmres(A, b, xexact, p)` qui effectue un redémarrage de la fonction `gmres` toutes les `p` $\in \mathbb{N}^*$ itérations.

Partie 2. (Gradient Conjugué)

5. Programmer une fonction `gradient_conjugue(A, b, xexact)` qui, à partir d'une matrice $A \in M_n(\mathbb{R})$, un vecteur $b \in \mathbb{R}^n$ et une solution exacte `xexact` $\in \mathbb{R}^n$ (si disponible), renvoie la solution x du système linéaire obtenue par l'algorithme du Gradient conjugué, ainsi que la liste des erreurs relatives, et la liste de la norme des résidus (relatifs). Nous rappelons ci-dessous l'algorithme (à gauche):

$$\begin{cases} \text{(Gradient Conjugué)} \\ x_0 \text{ donné} \\ r_0 = b - Ax_0, d_0 = r_0 \\ \text{Tant que condition non satisfaite} \\ \quad s_k = (r_k, r_k)/(Ad_k, d_k) \\ \quad x_{k+1} = x_k + s_k d_k \\ \quad r_{k+1} = r_k - s_k A d_k \\ \quad \beta_k = (r_{k+1}, r_{k+1})/(r_k, r_k) \\ \quad d_{k+1} = r_{k+1} - \beta_k d_k \end{cases} \qquad \begin{cases} \text{(Gradient Conjugué Préconditionné)} \\ x_0 \text{ donné} \\ r_0 = b - Ax_0, \\ Mz_0 = r_0, d_0 = z_0 \\ \text{Tant que condition non satisfaite} \\ \quad s_k = (r_k, z_k)/(Ad_k, d_k) \\ \quad x_{k+1} = x_k + s_k d_k \\ \quad r_{k+1} = r_k - s_k A d_k \\ \quad M z_{k+1} = r_{k+1} \\ \quad \beta_k = (r_{k+1}, z_{k+1})/(r_k, z_k) \\ \quad d_{k+1} = z_{k+1} + \beta_k d_k \end{cases}$$

6. Testez votre méthode sur la matrice suivante :
```
B = spsp.diags([[4.]*n,[-1]*(n-1),[-1] *(n-1),[-1] *(n-d),[-1] *(n-d)],
  [0,1,-1,d,-d])
```
avec $n = d^2$. Afficher l'erreur et le résidu en fonction des itérations. On pourra calculer la solution exacte avec une fonction scipy. Comparer avec ce que vous obtenez avec la méthode GMRES ainsi que les temps de calcul avec le module `time`. Commentez. Pourquoi ces algorithmes son bien adapté aux structures creuses ?

Partie 3 (Préconditionnement).

7. Appliquer la méthode GMRES aux matrices
```
C = np.diag(2+np.arange(n)) - np.diag(np.ones(n-1),1) - np.diag(np.ones(n-1),-1)
```
Comparer les résultats avec le système préconditionné $M^{-1}Cx = M^{-1}b$ où M est la partie diagonale de C (préconditionnnement diagonal ou Jacobi). Afficher le conditionnement des matrices C et $M^{-1}C$ grâce à la fonction `np.linalg.cond`. Comparer les temps de calcul. Observez les résultats pour différentes tailles de matrice n. Commentez.
8. Reprendre la question précédente avec
```
D = np.diag(2*np.ones(n)) - np.diag(np.ones(n-1),1) - np.diag(np.ones(n-1),-1)
```
Commentez.
9. Programmer une fonction `gradient_conjugue_precond(A, b, M, xexact)` qui, à partir d'une matrice $A \in M_n(\mathbb{R})$, un vecteur $b \in \mathbb{R}^n$, une matrice de préconditionnement, et une solution exacte `xexact` $\in \mathbb{R}^n$ (si disponible), renvoie la solution x du système linéaire obtenue par l'algorithme du Gradient conjugué préconditionné, ainsi que la liste des erreurs relatives, et la liste de la norme des résidus (relatifs). L'algorithme est rappelé ci-dessus (à droite).
10. Tester votre fonction `gradient_conjugue_precond` sur la matrice `B` (question 6), en utilisant comme matrice de préconditionnement la factorisation LU incomplète donnée par `spsplin.spilu`. Comparer les temps de calcul avec la méthode de gradient conjugué non-préconditionné. Commentez.