Algèbre Linéaire Numérique : Méthodes de réductions de modèles pour des problèmes PDE - Phase 2

Awa Dieng, Yvan Rameliarison, Victor Drouin Viallard

1er juin 2015

Table des matières

Chapitre 1

Introduction

1.1 Le problème étudié

Ce projet consiste à étudier une technique de réduction (approximation d'un espace à l'aide de sons sous-espace singulier dominant gauche) dans le but d'accélérer le calcul de l'évolution d'une vague atmosphérique. L'intérêt de cette étude vient du fait que la taille des problèmes étudiés rend le calcul direct des solutions très long et que de tels algorithmes de réduction permettent d'accélérer la recherche de solutions aux équations de vagues atmosphériques (mais aussi d'autres problèmes d'équations aux dérivées partielles).

1.2 Travail effectué

Au cours du projet, nous avons implémenté et utilisé divers algorithmes de calcul d'espace singulier dominant gauche dans le but de réduire la taille des problèmes, puis de ces sous-espaces réduits nous avons déterminé l'évolution d'une vague atmosphérique à partir de sa condition initiale (réduction) et déterminé l'appartenance d'une vague atmosphérique à telle ou telle catégorie (classification). En outre nous avons implémenté l'algorithme de la puissance itérée en matlab puis en Fortran dans le but d'observer dans quelles mesures il peut concurencer l'algorithme SVD de matlab.Les algorithmes développés s'appuient sur la méthode de la puissance itérée avec projection de Rayleigh-Ritz.

Dans une première partie nous avons implémenté en matlab l'algorithme de la puissance itérée appliqué de façon à calculer l'espace singulier dominant gauche d'une matrice rectangulaire. Puis nous avons défini comment obtenir une solution des équations de vague atmosphérique à partir du sous espace dominant gauche d'une matrice contenant elle-même des solutions de ces équations.

Dans une deuxième partie Fortran a été utilisé pour écrire à nouveau l'algorithme de la puissance itérée mais dans un langage de plus bas niveau, de façon à améliorer sensiblement le temps d'exécution.

Chapitre 2

Algorithme de la puissance itérée

2.1 Principe général

Le principe de base de l'algorithme de la puissance itérée est le suivant : si on prend un vecteur quelconque de l'espace puis qu'on le multiplie par une matrice et qu'on le normalise, ceci de manière répétée, alors ce dernier va tendre vers un vecteur propre de la matrice associé à sa plus haute valeur propre. En d'autres termes si $y \in \mathbb{R}^n$ est assez quelconque et si u est défini par $u_0 = y, u_{k+1} = \frac{A.u_k}{\|A.u_k\|}$ alors

$$\lim_{k\to\infty} u_k = u \in Spectre(A)$$

Ainsi comme pour une matrice carrée de $M_n(\mathbb{R})$ on ne désire que la partie dominante du spectre, on prend une matrice de m vecteurs orthogonaux V - qu'on espère voir converger vers une matrice de vecteurs propres de A -, on leur applique A (ou A^p, le principe reste le même), on orthonormalise le résultat, puis grace au théorème du quotient de Rayleigh on se réduit à calculer le spectre de la matrice $H = V^T.A.V$, dont la taille est moindre, et on obtient le "spectre" de A (ce n'est pas exactement le spectre puisque cela suppose que V ne contient que des vecteurs propres de A or ce n'est pas immédiatement le cas).

2.2 Modifications apportées

Dans l'algorithme qu'il nous a été demandé d'implémenter il s'agissait de calculer l'espace singulier dominant gauche d'une matrice Z rectangulaire. Dans ce cas précis l'algorithme précédent ne s'applique pas correctement.

Théorème : Si $A = U.\Sigma.V^T$ est une décomposition en valeurs singulières, alors les vecteurs colonnes orthogonaux de U sont des vecteurs propres de $B = A.A^T$.

A l'aide du théorème précédent on se réduit donc à calculer le spectre et des vecteurs propres associées de la matrice symétrique $Z.Z^T$, et on peut pour cela reprendre l'algorithme précédent.

Petite modification cependant : le nombre de lignes n de la matrice Z dans notre cas étant considérable, on ne peut se permettre d'utiliser l'espace mémoire d'une matrice de $M_n\mathbb{R}$. On doit donc systématiquement diviser les calculs et transformer les opérations de la forme $y = A.u$ en $y = Z.(Z^T.u)$.

2.3 Processus de l'algorithme

Tant que le nombre de vecteurs singuliers est inférieur à la taille de l'espace de travail, qu'on n'a pas récupéré un spectre assez important, et que l'on n'a pas atteint le nombre maximal d'itérations, on effectue les opérations suivantes :

1. On calcule Y=Z*(tZ*U) (matrice m*l)
   (a) On calcule d'abord tZ*U (n*m par m*l)
   (b) Puis on multiplie à gauche par Z (m*n par n*l)
2. On calcule la nouvelle base orthogonale de l'espace de travail Y avec Gram-Schmidt

3. On recalcule Y=Z*tZ*U par la même méthode que précédemment
4. On calcule Y=tU*Y (matrice l*l) qui est symétrique (car H=t(tZ*U)*(tZ*U))
5. On calcule les éléments propres de H
6. On ordonne les vecteurs de H
7. On calcule Y=U*H (matrice m*l)
8. On s'assure que le nombre de vecteurs n'est pas plus grand que la taille de l'espace
9. On calcule le résidu pour chacun des vecteurs
10. On s'assure que les résidus calculés sont inférieurs au epsilon de précision que l'on s'est fixé
11. On accède aux valeurs singulières de Z qui sont les racines carrées de celles de Z*tZ

Remarque De façon à accélérer le calcul de V il aurait été judicieux - cela n'ayant pas été fait - d'utiliser l'algorithme d'exponentiation rapide employé à $Z^T.Z$ (de taille raisonnable) et $p-1$.

Chapitre 3

Tests et comparaisons

3.1 Tests effectués

Remarque préliminaire Les sources fournies présentent une coquille dont l'importance n'est pas négligeable si on veut comparer les différents algorithmes : la condition d'arrêt pour l'algorithme de la SVD est erroné et ce dernier "oublie" systématiquement une valeur singulière (donc un vecteur singulier). On peut s'assurer de cette erreur en affichant la variabilité qui dès lors ne dépasse jamais la valeur de percentInfo donnée. Dans la suite nous considérons l'erreur comme corrigée.

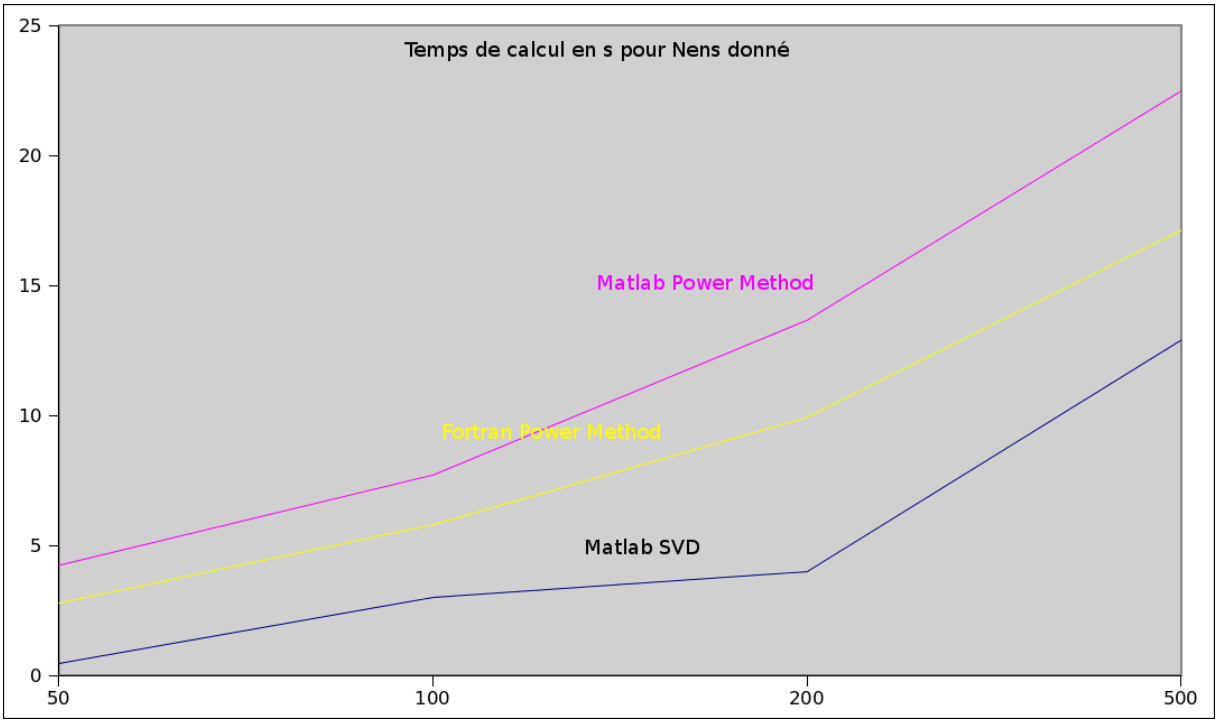
Contenu des tests Afin de comparer au mieux les algorithmes, en prenant en compte le fait que la fonction "svd" de matlab retourne systématiquement - pour des tailles de matrice raisonnables - tout l'espace singulier gauche, nous avons fait le choix de toujours laisser les algorithmes de la puissance itérée (celui de matlab et celui de Fortran) atteindre la valeur de "percentInfo". Le sujet évoquait une comparaison en terme d'erreur entre l'algorithme utilisant la SVD et l'algorithme de la puissance itérée implémenté en Fortran mais celle-ci nous a paru peu pertinente compte tenu du fait que si on laissait à Fortran le temps d'atteindre percentInfo alors l'erreur était toujours la même entre les deux algorithmes*. Nous aurions aussi pu faire le choix de ne pas laisser obligatoirement les algorithmes de la puissance itérée atteindre percentInfo, par exemple en fixant maxIter ou m de façon à tous leur laisser le même temps d'exécution, mais il aurait alors fallu bricoler sur leur valeur entière ce qui n'eut pas été scientifiquement correct pour effectuer des comparaisons.

Nous avons donc effectué des tests à ϵ constant de 10^{-8} et en nous assurant que le choix de maxIter et m (la dimension maximale du sous espace singulier calculé par les algorithmes de la puissance itérée) permettait aux algorithmes de toujours atteindre percentInfo. Nous avons alors enregistré les différences de temps relatives entre les algorithmes - en effectuant plusieurs tests pour chaque jeu d'entrées puis en moyennant - tout en constatant la constance, d'un algorithme à l'autre, de l'erreur, de la variabilité, et de la dimmension du sous-espace singulier gauche permettant d'atteindre la variabilité demandée (percentInfo).

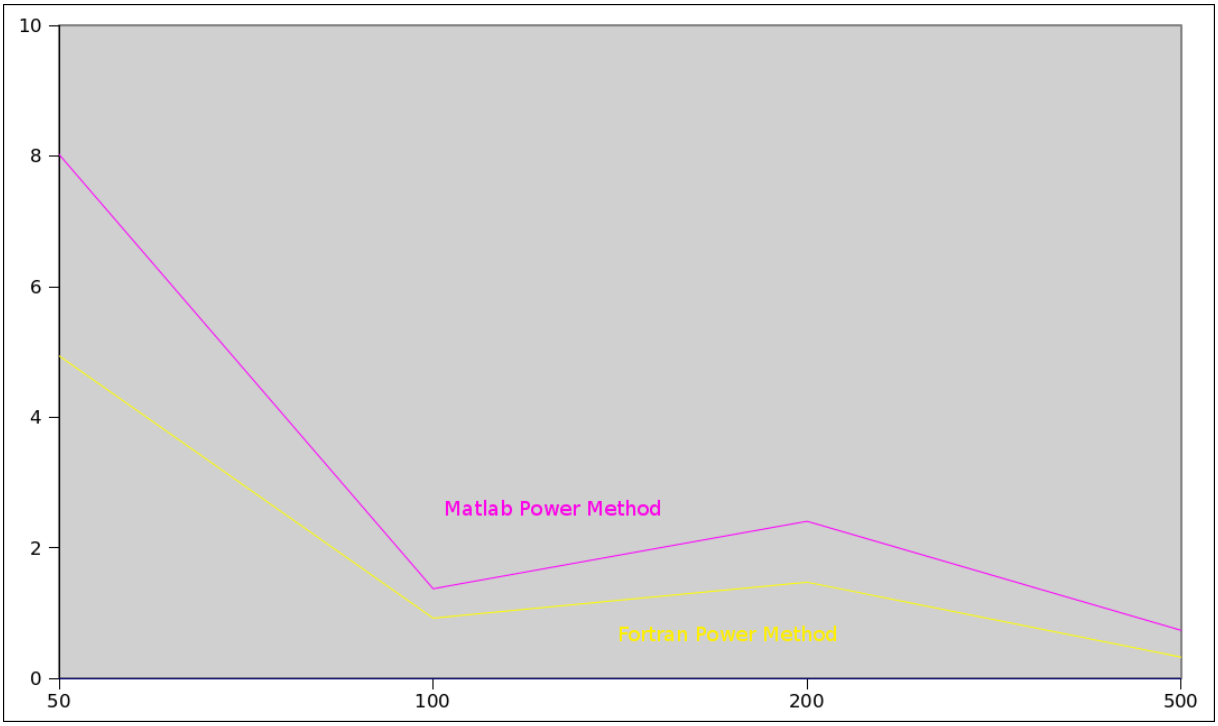
***** L'erreur est en effet toujours sensible la même dès lors qu'on laisse les algorithmes atteindre percentInfo car ils trouvent tous les mêmes valeurs et vecteurs singuliers - à une différence négligeable près.

3.2 Résultats des tests

Variation de N_{ens} Pour $p = 5$ et $percentInfo = 0.95$ nous avons calculé les différents temps d'exécution des algorithmes sur une même matrice et pour des valeurs de N_{ens} variables.



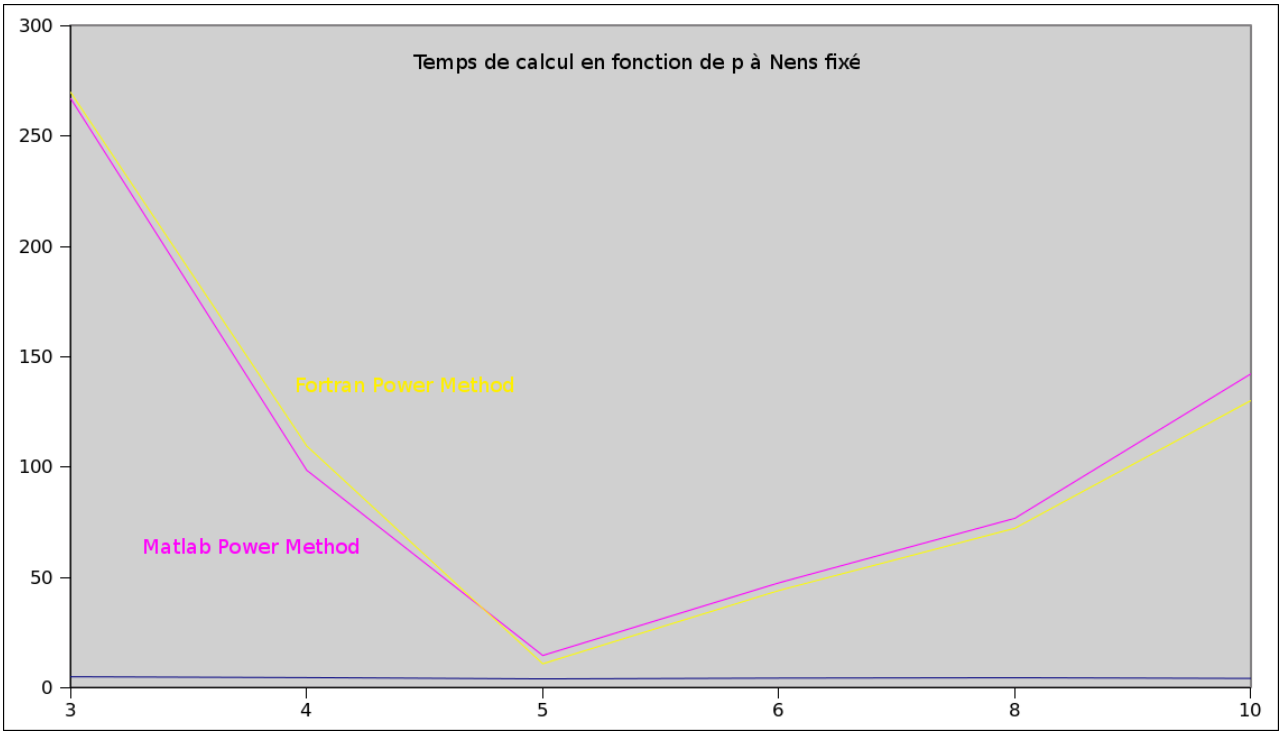
En calculant la différence relative entre les temps de calcul des algorithmes et celui de la SVD de matlab on obtient le graphique suivant :



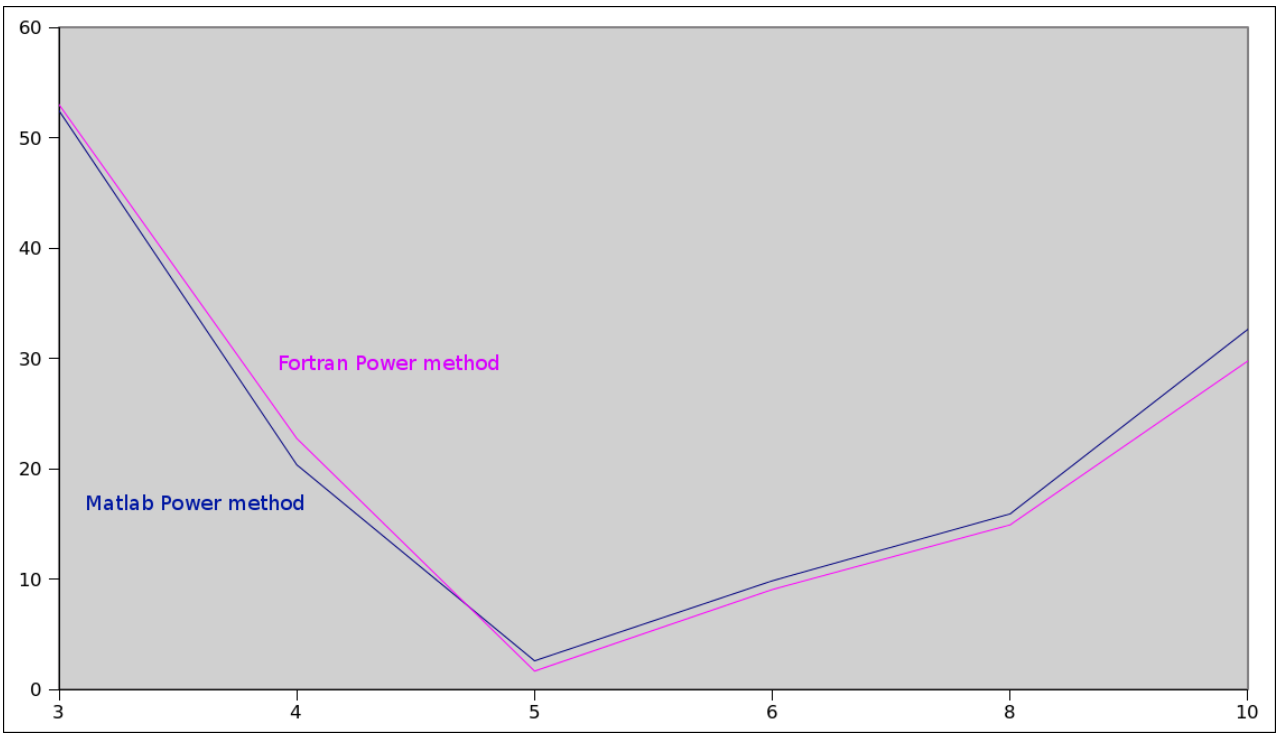
On observe alors que pour des valeurs de N_{ens} grandissantes l'algorithme de la puissance itérée tend à devenir meilleur que celui de la SVD.

Remarque On n'a cependant pas pu vérifier cette conjecture puisque matlab a refusé de calculer des modèles pour des valeurs de Nens très supérieures à 500.

Variation de p Pour $N_{ens} = 200$ et pour $percentInfo = 0.95$ nous avons calculé les différents temps d'exécution des algorithme sur une même matrice et pour des valeurs de p variables.



En calculant la différence relative entre les temps de calcul des algorithmes et celui de la SVD de matlab on obtient le graphique suivant :



Cette fois ci l'algorithme de la puissance itérée ne s'améliore pas proportionnellement à l'augmentation de la variable p puisqu'il semble qu'il existe une valeur optimale qui permette d'optimiser les calculs, ici atteinte pour 5.

Chapitre 4

Analyse des tests

4.1 SVD plus rapide sur de petites tailles

Comme on peut le voir sur les tests effectués sur des matrice de faible largeur, la fonction "svd" de matlab calcule l'espace singulier gauche (entier) de manière bien plus efficace que notre algorithme de puissance itérée qui lui se restreint à sa partie dominante. Cela est certainement dû à la complexité de cette fonction qui certainement adapte son calcul de manière très poussée en analysant préalablement la matrice à laquelle elle a affaire.

A côté nos algorithmes de puissance itérée sont bien plus lents avec une grande variation selon la valeur de p. L'algorithme de la puissance itérée peut en effet prendre beaucoup de temps pour converger, d'autant plus que la base orthonormale U est intialement choisie aléatoirement.

Inversement quand la largeur de la matrice dont il faut calculer l'espace dominant gauche augmente, la SVD commence à perdre son avantage (on aurait aimé pouvoir calculer des ensembles de solutions de taille plus grande que 1000 mais matlab refusait) et l'algorithme de la puissance itérée à la surpasser. Cela n'a rien d'étonant puisque connaissant la "physique" du problème nous savons que nous pouvons nous restreindre à appliquer la méthode de la puissance itérée à un nombre faible de vecteurs (une dizaine pour une précision de 95%) tandis que la SVD s'oblige à calculer tout l'espace dominant gauche ce qui prend alors plus de temps malgré ses algorithmes de calcul plus performants.

4.2 maxIter et les lacunes de la "Power Method"

Lorsque l'on augmente la précision requise ($percentInfo$) l'algorithme de la puissance itérée commence à rapidement s'enliser dans les calculs puisque la convergence des vecteurs propres est de plus en plus lente à mesure que les valeurs propres sont petites. Il faut alors augmenter considérablement la taille maximale du sous-espace dominant gauche à calculer - ce qui augmente considérablement les calculs - et aussi le nombre maximal d'itérations.

Néanmoins il devient alors intéressant de jouer sur le paramètre p pour diminuer ce nombre d'itérations et amoindrir cette perte de temps. On observe d'ailleurs sur la deuxième série de tests que ce paramètre est très important puisqu'un mauvais choix peut multiplier par plus de 10 le temps de calcul. L'optimisation du temps de calcul pour une valeur de p donnée vient du fait qu'augmenter p augmente les calculs préalables à chaque tour de boucle mais que dans le même temps cela permet de diminuer le nombre de tour et donc d'éviter plusieurs calculs d'espaces propres de H.

Chapitre 5

Conclusion

L'aspect "réduction" du problème nous a paru très intéressant tant il est d'actualité (gestion de grandes bases de données en informatique, prévisions météorologique alors que le climat semble se dégrader, etc.), et on voit rapidement combien il peut être aisément adapté à tout un large panel de problèmes.

Cependant l'utilisation de matlab, dont le fonctionnement semble aléatoire (pas identique d'un ordinateur à l'autre, nombreux problèmes de mémoire impossible à appréhender puisqu'un même test peut d'une fois sur l'autre échouer puis réussir, etc.), et l'emploi du Fortran, vieux langage accumulant les lacunes syntaxiques, ont considérablement atténué notre plaisir à travailler sur ce problème de réduction.

On regrettera aussi l'emploi de "codes à trous" car même si cela permet d'"aller plus loin" que si on avait du tout écrire nous même, cela empêche d'avoir une certaine "liberté" dans le traitement du problème et nous contraint à devoir passer énormément de temps à comprendre ce que l'auteur du code à voulu dire alors qu'une simple explication du fonctionnement nous aurait suffit pour travailler et rendre un travail fonctionnel.