Rapport du TP2 projet Calcul Scientifique et Analyse de données

Thierry Xu
Tom Bonetto
Mickaël Song

Département Sciences du Numérique - Première année
2021-2022

Table des matières

Table des figures

Liste des tableaux

1 Introduction

On a implémenté dans la première partie du projet, la méthode de la puissance itérée pour trouver les couples propres dominants. Or il n'est pas nécessaire d'avoir la décomposition spectrale de la matrice de variance/covariance symétrique dans sa globalité pour pouvoir réduire la dimension par l'ACP. En réalité, on a juste besoin des couples propres dominants fournissant suffisamment d'informations pour les données. Dans cette deuxième partie on va s'intéresser à l'implémentation de différentes méthodes plus efficaces en matière de performance comme "subsapce iteration" ou encore la puissance itérée avec déflation qui permettent de trouver les couples propres d'une matrice.

2 Limites de la méthode de la puissance itérée

2.1 Question 1

On va comparer le temps d'exécution pour le calcul des premiers couples propres entre la méthode de la puissance itérée et la fonction eig de Matlab.

Type de matrice	1		2	3	4	
	50x50	200x200			50x50	200x200
Puissance itérée	6e-02	5	2e-02	1e-02	5e-02	4.6
eig	1e-02	2e-02	1e-02	1e-02	1e-02	2e-02

TABLE 1 – Tableau de comparaison du temps d'exécution entre la méthode de la puissance itérée et la fonction eig

On remarque que pour des tailles de matrices importantes la fonction eig est bien plus efficace que la méthode de la puissance itérée, que soit en matière de temps d'exécution mais également de qualité des couples propres.

2.2 Question 2

En réarrangeant les opérations dans l'algorithme de la méthode de la puissance itérée, on peut faire en sorte qu'il y est qu'une seule muLtiplication matricielle A par v, ce qui réduit quasiment le temps d'exécution de moitité. (Voir power_v12.m, on passe de 4.5 sec à 2.5 sec).

Algorithme 1 : Méthode de puissance itérée améliorée

Input : $A \in \mathbb{R}^{nxn}$

Output : (λ_1, v_1) couple propre associé à la plus grande valeur propre

$v \in \mathbb{R}^n$ donné;
$\beta = v^T.A.v$;
$y = A.v$
repeat
  $v = y/\|y\|$
  $y = A.v$
  $\beta_{old} = \beta$
  $\beta = v^T.y$
until $|\beta - \beta_{old}| / |\beta_{old}| < \epsilon$

2.3 Question 3

Le problème avec la méthode de la puissance itérée par déflation c'est que son temps de calcul est très long du fait que l'on calcule le spectre en entier, il y a donc un grand nombre de produit matriciel ce qui affecte grandement le temps d'exécution.

3 Améliorer la méthode de la puissance itérée pour calculer les vecteurs propres dominants

3.1 Question 4

Si l'on applique la méthode de la puissance itérée sur m vecteurs, la matrice V va converger vers une matrice avec m fois le même vecteur (un vecteur propre associée à la valeur propre dominante). En modifiant notre script puissance_iteree.m de la partie 1 du projet, on vérifie effectivement cette conjecture.

3.2 Question 5

Il n'est à priori pas gênant de calculer l'entière décomposition spectrale de H, puisque H($m \times m$) est de taille inférieure à A($n \times n$), on a : m $\ll$ n. Donc dans tous les cas le calcul de la décomposition spectrale de H est négligeable par rapport à celui de A.

3.3 Question 6

Voir : subspace_iter_v0.m.

3.4 Question 7

On reprend l'algorithme 4 et on identifie les différentes étapes avec les lignes de code de subspace_iter_v1.m, voir figure 1, 2 et 6.

Lignes 22 à 49 : "Generate an initial set of m orthonormal vectors V $\in \mathbb{R}^{n \times m}$; k = 0 ; PercentReached = 0"
Ligne 56 : "Compute Y such that Y = A $\cdot$ V"
Ligne 58 : "V $\leftarrow$ orthonormalisation of the columns of Y"
Ligne 61 : "Rayleigh-Ritz projection applied on matrix A and orthonormal vectors V"
Lignes 64 à 68 : "Convergence analysis step : save eigenpairs that have converged"
Lignes 70 à fin : "Convergence analysis step : update PercentReached"

```
function [ W, V, n_ev, it, itv, flag ] = subspace_iter_v1( A, m, percentage, eps, maxit )

    % calcul de la norme de A (pour le critère de convergence d'un vecteur (gamma))
    normA = norm(A, 'fro');

    % trace de A
    traceA = trace(A);

    % valeur correspondnat au pourcentage de la trace à atteindre
    vtrace = percentage*traceA;

    n = size(A,1);
    W = zeros(m,1);
    itv = zeros(m,1);

    % numéro de l'itération courante
    k = 0;
    % somme courante des valeurs propres
    eigsum = 0.0;
    % nombre de vecteurs ayant convergés
    nb_c = 0;

    % indicateur de la convergence
    conv = 0;

    % on génère un ensemble initial de m vecteurs orthogonaux
    Vr = randn(n, m);
    Vr = mgs(Vr);

    % rappel : conv = (eigsum >= trace) | (nb_c == m)
    while (~conv & k < maxit),

        k = k+1;
        %% Y <- A*V
        Y = A*Vr;
        %% orthogonalisation
        Vr = mgs(Y);

        %% Projection de Rayleigh-Ritz
        [Wr, Vr] = rayleigh_ritz_projection(A, Vr);

        %% Quels vecteurs ont convergé à cette itération
        analyse_cvg_finie = 0;
```

FIGURE 1 – 1ère partie du code de subspace_iter_v1.m

```
% nombre de vecteurs ayant convergé à cette itération
nbc_k = 0;
% nb_c est le dernier vecteur à avoir convergé à l'itération précédente
i = nb_c + 1;

while(~analyse_cvg_finie),
    % tous les vecteurs de notre sous-espace ont convergé
    % on a fini (sans avoir obtenu le pourcentage)
    if(i > m)
        analyse_cvg_finie = 1;
    else
        % est-ce que le vecteur i a convergé

        % calcul de la norme du résidu
        aux = A*Vr(:,i) - Wr(i)*Vr(:,i);
        res = sqrt(aux'*aux);

        if(res >= eps*normA)
            % le vecteur i n'a pas convergé,
            % on sait que les vecteurs suivants n'auront pas convergé non plus
            % => itération finie
            analyse_cvg_finie = 1;
        else
            % le_vecteur i a convergé
            % un de plus
            nbc_k = nbc_k + 1;
            % on le stocke ainsi que sa valeur propre
            W(i) = Wr(i);

            itv(i) = k;

            % on met à jour la somme des valeurs propres
            eigsum = eigsum + W(i);

            % si cette valeur propre permet d'atteindre le pourcentage
            % on a fini
            if(eigsum >= vtrace)
                analyse_cvg_finie = 1;
            else
                % on passe au vecteur suivant
                i = i + 1;
            end
        end
```

FIGURE 2 – 2ème partie du code de subspace_iter_v1.m

```
                    if(eigsum >= vtrace)
                        analyse_cvg_finie = 1;
                    else
                        % on passe au vecteur suivant
                        i = i + 1;
                    end
                end
            end
        end

        % on met à jour le nombre de vecteurs ayant convergés
        nb_c = nb_c + nbc_k;

        % on a convergé dans l'un de ces deux cas
        conv = (nb_c == m) | (eigsum >= vtrace);

    end

    if(conv)
        % mise à jour des résultats
        n_ev = nb_c;
        V = Vr(:, 1:n_ev);
        W = W(1:n_ev);
        it = k;
    else
        % on n'a pas convergé
        W = zeros(1,1);
        V = zeros(1,1);
        n_ev = 0;
        it = k;
    end

    % on indique comment on a fini
    if(eigsum >= vtrace)
        flag = 0;
    else if (n_ev == m)
            flag = 1;
        else
            flag = -3;
        end
    end
end
```

FIGURE 3 – 3ème partie du code de subspace_iter_v1.m

4 subspace_iter_v2 and subspace_iter_v3 : vers une meilleure résolution

4.1 Approche par block (subspace_iter_v2)

4.1.1 Question 8

Calculons le coût de A^p et de $A^p * V$:

- Pour p = 2, chaque élément de A^2 nécéssite m multiplications et (m-1) additions (d'après la formule C = AB, $c_{ij} = \sum_{k=1}^{m}(a_{ik} + b_{kj})$), soit $2m - 1$ opérations.
Il y a m^2 élements dans la matrice A^2, donc il y a au total $(2m - 1) * m^2 \approx 2m^3$ flops.
- Puisque chaque produit matricielle de A^2 à A^p coûte $2m^3$ flops, il faut donc un total de $2 * (p - 1) * m^3*$ flops pour A^p et $2 * p * m^3*$ flops pour $A^p * V$.

On peux réduire les flops en calculant $Y = A^p * V$ avant l'entrée de la boucle while puis de le stocker dans une variable intermédiaire : cela permettrait de ne pas à avoir calculer la même matrice à chaque tour de boucle. De plus, étant donné que A est une matrice symétrique, le coût du calcul du produit matriciel peut être réduite en effectuant seulement les opérations sur la diagonnale supérieure (ou inférieure).

4.1.2 Question 9

Voir : subspace_iter_v2.m.

4.1.3 Question 10

On a lancé l'algorithme avec des valeurs de p différentes. On a fixé la taille du sous-espace à 70 et le pourcentage de la trace que l'on veut atteindre à 50%.
On observe que plus p augmente, plus le temps d'itérations des sous-espaces et le nombre d'itérations sont faibles.

```
******* calcul avec subspace iteration v2 ******

Temps subspace iteration v2 = 1.040e+00
Nombre d'itérations : 141
Nombre de valeurs propres pour attendre le pourcentage = 59
Nombre d'itérations pour chaque couple propre
```

FIGURE 4 – Test du code de subspace_iter_v2.m pour p = 1

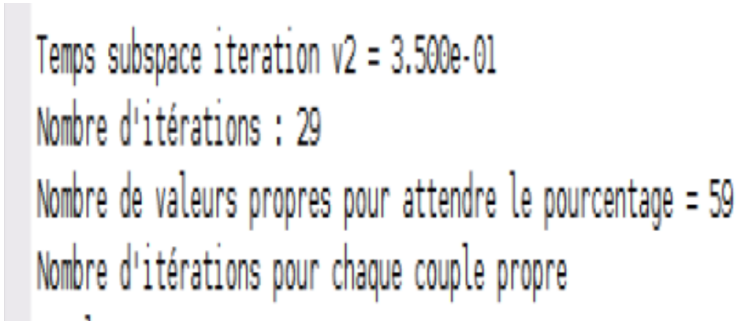

FIGURE 5 – Test du code de subspace_iter_v2.m pour p = 5

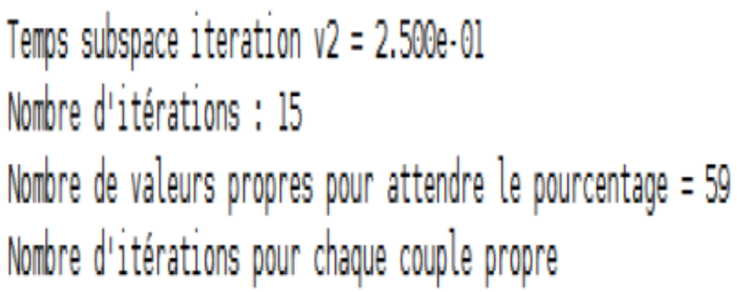

FIGURE 6 – Test du code de subspace_iter_v2.m pour p = 10

4.2 Méthode par déflation (subspace_iter_v3)

4.2.1 Question 11

Pour subspace_iter_v1 on remarque que la précision pour les différents vecteurs varie, ce qui est normal puisque ces vecteurs ne sont que des approximations obtenues grâce à la projection de Rayleigh. C'est grâce à un seuil d'erreur epsilon que l'on tente de trouver des valeurs approchées aux vecteurs que l'on cherche.

4.2.2 Question 12

Avec la méthode subspace_space_v3, les vecteurs de V qui ont déjà convergé vers les vecteurs propres seront gelés et les opérations seront donc effectuées sur ceux qui n'ont pas encore convergé.

4.2.3 Question 13

5 Expériences numériques

5.1 Question 15

Pour comparer les performances des différents algorithmes, nous les avons testés avec différentes tailles de matrices et différents nombres de valeurs propres recherchées.

Sur ce premier test, à l'aide de la table 2, on constate que la fonction eig est plus efficace pour calculer les valeurs propres peu importe le type de la matrice.

Type de matrice	1	2	3	4
eig	2.000e-02	1.000e-02	1.000e-02	1.000e-02
subspace_iter_v0	5.300e-01	1.000e-01	3.000e-02	1.300e-01
subspace_iter_v1	7.000e-02	1.000e-02	2.000e-02	6.000e-02
subspace_iter_v2	2.000e-02	2.000e-02	1.000e-02	1.000e-02
subspace_iter_v3	5.000e-02	3.000e-02	3.000e-02	5.000e-02

TABLE 2 – Tableau de comparaison du temps d'exécution entre les différents algorithmes pour une matrice de taille 50x50 et pour 20 valeurs propres recherchées

Pour le deuxième test, dont les résultats sont contenus dans la table 2 on remarque que l'algorithme subspace_iter_v2 est plus performante que les autres.

Type de matrice	1	2	3	4
eig	1.000e-02	2.000e-02	2.700e-01	1.000e-02
subspace_iter_v0	4.010e+00	1.345e+01	5.500e-01	3.990e+00
subspace_iter_v1	1.370e+00	2.000e-02	4.000e-02	1.390e+00
subspace_iter_v2	4.100e-01	1.000e-02	3.000e-02	3.900e-01
subspace_iter_v3	5.500e-01	8.000e-02	6.000e-02	5.900e-01

TABLE 3 – Tableau de comparaison du temps d'exécution entre les différents algorithmes pour une matrice de taille 200x200 et pour 50 valeurs propres recherchées

Pour le dernier test avec une matrice de grande taille et une grande quantité de valeurs propres recherchés, nosu remarquons que l'algorithme subspace_iter_v3 est plus efficace.

Type de matrice	1	2	3	4
eig	2.000e-02	3.000e-02	1.890e+00	3.000e-02
subspace_iter_v0	4.087e+01	4.820e+00	5.840e+00	4.162e+01
subspace_iter_v1	2.620e+00	1.200e-01	2.300e-01	2.420e+00
subspace_iter_v2	7.900e-01	1.900e-01	1.000e-01	7.500e-01
subspace_iter_v3	9.200e-01	2.900e-01	2.400e-01	9.600e-01

TABLE 4 – Tableau de comparaison du temps d'exécution entre les différents algorithmes pour une matrice de taille 400x400 et pour 150 valeurs propres recherchées