Projet Calcul Scientifique - Analyse de Données : Deuxième Partie

Ayoub LOUDYI - Mohamed Hamza LOTFI - Mahmoud LAANAIYA

Département Sciences du Numérique - Première année
2020-2021

Table des matières

Table des figures

1 Limitations of the power method

Question 1 : Pour n = 200, on exécute le script testv11.m, et on obtient le résultat de la figure 1

```
>> test_v11

Matrice 200 x 200 - type 2

******* création de la matrice ******

Temps de création de la matrice = 1.700e-01

******* calcul avec eig ******

Temps eig = 2.000e-02
Qualité des valeurs propres (par rapport au spectre de la matrice) = [0.000e+00 , 9.059e-08]
Qualité des couple propres = [4.072e-16 , 1.160e-06]

Matrice 200 x 200 - type 2

******* calcul avec la méthode de la puissance itérée ******

Temps puissance itérée = 8.000e-02
Nombre de valeurs propres pour attendre le pourcentage = 5
Nombre d'itérations pour chaque couple propre
couple 1 : 88
couple 2 : 185
couple 3 : 45
couple 4 : 252
couple 5 : 510
Qualité des valeurs propres (par rapport au spectre de la matrice) = [1.460e-16 , 1.595e-15]
Qualité des couple propres = [9.767e-09 , 1.794e-08]
```

FIGURE 1 – Exécution de testv11.m pour la question 1 (eig.m et *power_v*11.*m*)

On remarque alors que la fonction eig.m est 4 fois plus rapide que la méthode de la puissance itérée proposée dans la fonction *power_v*11.*m*, avec un temps d'exécution de 0.02 secondes pour la première et de 0.08 secondes pour la deuxième.

On peut expliquer cette différence de temps par plusieurs facteurs :

1. n petit : Pour n = 200, les calculs de temps matlab ne sont pas toujours à point. Cependant en utilisant d'autre valeurs de n (500, 700, ...) on remarque que le temps d'exécution d'eig.m est toujours considérablement inférieur à celui de *power_v*11.*m*. Comme le montre la figure 1
2. Qualité des valeurs propres : On remarque que la qualité des valeurs propres des de la fonction eig.m est nulle seulement pour la première valeur, mais est déplorable pour les autres couples propres par rapport à la fonction *power_v*11.*m* dont la précision globale est plus petite que ce dernier (d'ordre 10^9 de différence pour la qualité des valeurs propres et d'ordre de (10^2) pour la qualité des vecteurs propres) par rapport à celle de eig.m. Ce qui peut expliquer cette disparité dans le temps d'exécution.

Question 2 : On remarque sur le code de *power_v*11.*m* qu'on a un vecteur z en dehors de la boucle qui initialise le produit de A par v, on élimine alors le vecteur y, et on ne procède plus qu'avec le vecteur z. Le principe de l'algorithme et de ne réaliser de produit matrice-vecteur qu'une fois par itération et de normaliser le vecteur z qui jouera à la fois le rôle du vecteur v et le vecteur z après et avant la normalisation respectivement. La figure 2 montre le code matlab utilisée pour la réduction des produits matrices par vecteurs dans le code de *power_v*12.*m*.

```
% méthode de la puissance itérée
v = randn(n,1);
z = A*v;
beta = v'*z;

% conv = || beta * v - A*v||/|beta| < eps
% voir section 2.1.2 du sujet
norme = norm(beta*v - z, 2)/norm(beta,2);
nb_it = 1;

while(norme > eps && nb_it < maxit)
  v = z/norm(z,2);
  z = A*v;
  beta = v'*z;
  norme = norm(beta*v - z, 2)/norm(beta,2);
  nb_it = nb_it + 1;
end
```

FIGURE 2 – Boucle du produit A×v dans *power_v*12.*m*

Vérification de la réduction du temps d'exécution : Dans la figure 3 on constate effectivement (Pour n = 700) que le temps d'exécution est bel est bien largement réduit quand on élimine un des deux produits matrice-vecteur ce qui nous fait un gain considérable en coût de calcul, puisque cette fois dans *power_v*12.*m* on a déja le produit matrice-vecteur stocké dans un vecteur au lieu de le recalculer au début de chaque boucle.

```
Matrice 700 x 700 - type 2

******* calcul avec la méthode de la puissance itérée ******

Temps puissance itérée = 3.880e+00

Matrice 700 x 700 - type 2

******* calcul avec la méthode de la puissance itérée améliorée ******

Temps puissance itérée améliorée = 1.930e+00
```

FIGURE 3 – Temps d'exécution de *power_v*11.*m* et *power_v*12.*m*

L'algorithme utilisé : Après suppression du deuxième produit matriciel $(A \times v)$ on obtient l'algorithme suivant :

while ($convg \neq 0$ and $k < m$) do :
- k = k + 1
- $v \in \mathbb{R}^n$
- $z = A \times v$
- $\beta = v^\top \times z$
- $norme = \frac{|\beta \times v - z|}{|\beta|}$
- $nb\_it = 1$
- **while** ($convg > eps$ and $nb\_it < maxit$) do :
  - $v = \frac{z}{|z|}$
  - $z = A \times v$
  - $\beta = v^\top \times z$
  - $norme = \frac{|\beta \times v - z|}{|\beta|}$
  - $nb\_it = nb\_it + 1$
- **end while**
- ...
- ...
- ...

end while

Question 3 : Le majeur incovénient de la méthode de la puissance itérée avec déflation est qu'on effectue une itération dans laquelle on calcule à chaque fois un produit matriciel dans le but de déterminer juste un seul couple propre. Autrement dit le nombre d'itérations à faire pour calculer un seul couple propre est remarquablement élevé (surtout si la taille de la matrice est importante).

2 Extending the power method to compute dominant eigenspace vectors

2.1 subspace_iter_v0

2.1.1 Orthonormalisation

Question 4 : Proposition On remarque en utilisant plusieurs matrices A, que V converge vers une matrice de rang 1, or tout ces vecteurs colonnes sont liées. Ceci est d'autant plus clair en regardant les lignes 1 et 3 du vecteur V en utilisant la matrice A suivante (de ligne 1 et 3 identiques).

```
>> A = [ 1 0 1; 1 1 1; 1 0 1]

A =

     1     0     1
     1     1     1
     1     0     1
```

FIGURE 4 – Matrice symétrique A utilisée pour l'exemple

```
>> [ W, V, n_ev, itv, flag] = power_v11_extended( A, 2, percentage, eps, maxit )
hi1

W =

   2.000000017119218


V =

   0.227978446643161   0.338662812011069
   0.455956867964072   0.677325615793650
   0.227978446643161   0.338662812011069


n_ev =

     1


itv =

    26


flag =

     0
```

FIGURE 5 – Résultats pour une matrice A symetrique

Conclusion : On en conclut alors qu'effectivement, en utilisant la méthode de la puissance itérée sur m vecteurs directement on retrouve un résultat qui ne comprend qu'un seul eigenvector dominant au lieu de m. Ce qui valide la conjecture établie précédemment.

2.1.2 Rayleigh quotient

Question 5 : Le calcul de toute la décomposition spectrale de la matrice H ne pose pas problème car la matrice H est de taille (m) alors que la matrice A est de taille (n) avec $m << n$. Ceci est dû en effet au fait qu'on aie besoin que d'un nombre limité de vecteurs propres pour donner suffisamment d'informations sur les données à traiter. Or avec en selectionnant les m composantes principales d'un système on est capable d'obtenir de hauts pourcentages d'informations sur ce dernier.

Question 6 : On a généré un ensemble initial de m vecteurs orthogonaux.
En effet on a généré un ensemble aléatoire avec rand(n, m), puis on applique la fonction mgs à cet ensemble pour l'orthonormaliser.
Ainsi on calcule A.V, puis le quotient de Rayleigth et enfin on vérifie la convergence avec l'invariance du sous-espace V lorsque :

$$\frac{norm(AV - VH)}{norm(A)} < eps \tag{1}$$

2.2 subspace_iter_v1

2.2.1 Convergence analysis step

Question 7 : Les deux figures 6 et 7 montrent la code du fichier **subspace_iter_v1.m** dans lequel figurent toutes les étapes de l'algorithme 4 qu'on détaillera juste après :

```
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
    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;

        %% orthonormalisation
        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;
        % 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)
```

FIGURE 6 – Première partie de subspace_iter_v1.m

```
74 -        while(~analyse_cvg_finie)
75              % tous les vecteurs de notre sous-espace ont convergé on a fini (sans avoir obtenu le pourcentage)
76 -            if(i > m)
77 -                analyse_cvg_finie = 1;
78 -            else
79                  % est-ce que le vecteur i a convergé
80
81                  % calcul de la norme du résidu
82 -                aux = A*Vr(:,i) - Wr(i)*Vr(:,i);
83                  %res = sqrt(aux'*aux);
84 -                res = norm(aux)/normA;
85
86 -                if(res > eps)
87                      % le vecteur i n'a pas convergé, on sait que les vecteurs suivants ne convergeront pas également
88                      % => itération finie
89 -                    analyse_cvg_finie = 1;
90 -                else
91                      % le_vecteur i a convergé
92                      % un de plus
93 -                    nbc_k = nbc_k + 1;
94                      % on le stocke ainsi que sa valeur propre
95 -                    W(i) = Wr(i);
96
97 -                    itv(i) = k;
98
99                      % on met à jour la somme des valeurs propres
100 -                   eigsum = eigsum + W(i);
101
102                     % si cette valeur propre permet d'atteindre le pourcentage
103                     % on a fini
104 -                   if(eigsum/traceA >= percentage)
105 -                       analyse_cvg_finie = 1;
106 -                   else
107                         % on passe au vecteur suivant
108 -                       i = i + 1;
109 -                   end
110 -               end
111 -           end
112 -       end
113
114         % on met à jour le nombre de vecteurs ayant convergés
115 -       nb_c = nb_c + nbc_k;
116
117         % on a convergé dans l'un de ces deux cas : soit on a atteint le nombre de couples propores maximal, soit on a eu le pourcentage
118         % souhaité
119 -       conv = ((nb_c == m) | (eigsum/traceA >= percentage));
```

FIGURE 7 – Deuxième partie de subspace_iter_v1.m

La figure 8 représente le code de la projection de Raleigh-Ritz utilisé dans l'algorithme ci-dessous :

```
1    % projection de Rayleigh-Ritz
2    % Données
3    % A : matrice dont on cherche des couples propres
4    % V : ensemble de m vecteurs orthonormés
5
6    % Résultats
7    % W : vecteur contenant les approximations des valeurs propres
8    % V : matrice des vecteurs propres correspondant
9    function [ W, V ] = rayleigh_ritz_projection( A, V )
10
11 -  H = V'*(A*V);
12   % Décomposition spectrale de H
13 -  [VH, DH] = eig(H);
14   % Classement dans l'ordre décroissant
15 -  [W, indice] = sort(diag(DH), 'descend');
16   % Résultat final
17 -  V = V*VH(:, indice);
18
19 -  end
```

FIGURE 8 – Projection de Raleigh-Ritz

Les étapes de l'algorithme 4 élaborées dans le code de **subspace_iter_v1.m** par ligne :

1. Generate an initial set of m orthonormal vectors V : Lignes 42 à 43.
2. Repeat jusqu'à until : Lignes 46 à 112.
3. Compute Y such that Y = A · V : Ligne 51.
4. V ← orthonormalisation of the columns of Y : Ligne 54.
5. Rayleigh-Ritz projection applied on matrix A ans orthonormal vectors V : Ligne 57. (Elle est détaillée dans la figure 8)
6. Convergence analysis step : save eigenpairs that have converged and update PercentReached :
   Lignes 66 à 111.

2.3 subspace_iter_v2 and subspace_iter_v3

2.3.1 Block approach (subspace_iter_v2)

Question 8 : De façon générale, si on prend deux matrices A de taille $(m \times n)$ et B de taille $(n \times p)$ alors les composantes de la matrice produit C sont calculées de la façon suivante : $c_{ij} = \sum_{k=0}^{n} a_{ik} \times b_{kn}$. Donc le coût du calcul de C est : 2mnp flops. Pour notre cas la matrice A est une matrice carrée de taille $(n \times n)$, donc le calcul de A^2 est de coût $2n^3$ flops. On peut montrer alors par récurrence que le coût du calcul de A^p est de $2(p-1)n^3$ flops. D'autre part, la matrice V est de taille $(n \times m)$, alors le coût du calcul de AV est $2mn^2$ flops. On peut conclure alors que le coût total du calcul de A^pV est : $2(mn^2 + (p-1)n^3)$ flops.

Une façon d'organiser le calcul pour réduire le coût est de remplacer le calcul de A^pV par un calcul de AV p fois, c-à-d (**Pour p fois faire** V = AV). Ceci permettra de réduire le coût du calcul à $2pmn^2$.

Question 10 : La figure 9 contient des test de **subspace_iter_v2.m** pour différentes valeurs de p. Ce test est présent dans le code **test_sub_v2.m**

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

******* création de la matrice ******

Temps de création de la matrice = 2.360e+00

Test pour p = 1

Temps subspace iteration v2 = 1.100e-01
Nombre d'itérations : 44
Nombre de valeurs propres pour attendre le pourcentage = 13

Test pour p = 5

Temps subspace iteration v2 = 1.309e+03
Nombre d'itérations : 10
Nombre de valeurs propres pour attendre le pourcentage = 13

Test pour p = 10

Temps subspace iteration v2 = 2.200e-01
Nombre d'itérations : 4
Nombre de valeurs propres pour attendre le pourcentage = 13

Test pour p = 20

Temps subspace iteration v2 = 1.309e+03
Nombre d'itérations : 3
Nombre de valeurs propres pour attendre le pourcentage = 13

Test pour p = 40

Temps subspace iteration v2 = 3.400e-01
Nombre d'itérations : 3
Nombre de valeurs propres pour attendre le pourcentage = 13

Test pour p = 100
subspace iteration v2 : convergence non atteinte: 10000
```

FIGURE 9 – Résultats de test de subspace_iter_v2.m pour p = 1,5,10,20,40,100

En observant ce test avec différentes valeurs de p, on remarque que plus la valeur de p augmente, plus le nombre d'itérations nécéssaires diminue et plus la qualité des couples propres augmente. Cependant, on ne doit pas augmenter la valeur de p plus qu'il faut, car sinon on n'aura plus de convergence des vecteurs propres.

2.3.2 Deflation method (subspace_iter_v3)

Question 11 : Les précisions des vecteur propres calculées diffèrent comme le montre la figure 10, elle se détériore plus on a de vecteurs propres calculés. Ceci est dû à l'approximation des vecteurs propres par la projection de Rayleigh-Ritz. Or le critère de convergence d'un vecteur dans ce cas est lié au calcul de la norme du résidu explicité dans **subspace_iter_v1.m** par la variable res (ligne 80). Or à une certaine itération k, le nombre de colonnes convergente nbc est plus grand que celui à l'itération k-1, et donc le résidu (aux) dans la figure 11 a des éléments diagonaux sur les nbc premières colonnes qui tendent vers 0, et donc la norme a ses plus grandes valeurs sur les autres colonnes restantes. Ainsi plus on réitère cette projection plus la valeur de la norme du résidu diminue, et puisque le critère d'arrêt à une borne inférieur (eps*normA) fixe, donc sa précision aussi.

```
qv1 =

   1.0e-07 *

   0.000000006686369
   0.000000041239016
   0.000000011586153
   0.000000051855287
   0.000002980050638
   0.000000845273825
   0.000660256006536
   0.057110879621853
   0.082340991182967
   0.104245228757894
   0.435218366489843
```

FIGURE 10 – Précisions des vecteurs propres calculés par subspace_iter_v1.m

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

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

FIGURE 11 – Partie du code subspace_iter_v1.m qui porte sur la convergence des approximations

Question 12 : On anticipe que dans subspace_iter_v3.m, qu'on aura un gain en temps d'execution en terme de convergence des vecteurs propres, dû à la réduction du "block approach", et de la sauvegarde des vecteurs orthonormés qui ont convergé et en n'orthonormalisant que ceux qui ne le sont pas, par rapport à ceux qui le sont et entre eux dans la projection de Rayleigh-Ritz.

Question 13 : Le problème dans les versions précédantes c'est le fait qu'on perd du temps avec des colonnes inutiles. Pour cela, on cherche dans cette dernière version à ignorer les colonnes inutiles.
On prend un entier noté nb_c qui présente le nombre de vecteurs ayant convergés. et on ne prend que les éléments dont l'indice est supérieur à cet entier.
Même dans l'orthonormalisation, on a utilisé la fonction mgs_{block} qui orthonomalise juste à partir d'un certain rang.

3 TO DO : Numerical Experiments

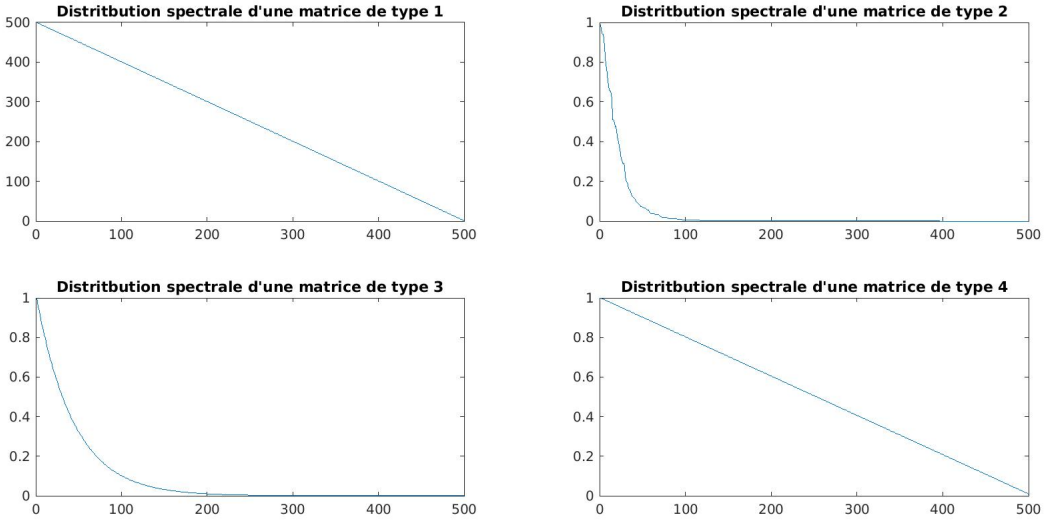

FIGURE 12 – Comparaison de la distribution spectrale des 4 types de matrices

Question 14 :

On remarque que les matrices de types 1 et 4 sont de distribution affine, et donc prendrons plus de temps à converger pour des pourcentages grands. Contrairement aux matrices de types 2 et 3 qui ont l'amplitude dominante de leurs spectres sur leurs 100 et 200 premières valeurs propres respectivement. Ainsi si on devait anticiper quel type de matrice convergerait plus vite par rapport à un pourcentage à atteindre, les matrices de type 2 seront les premières, suivies des matrices de type 3 et les matrices de type 1 et 4 sont ex-aequo dû au fait qu'ils ont la même pente.

Question 15 : On constate après quelques expérimentations que eig.m surclasse toutes nos méthode en terme de qualité des couples propres ainsi qu'en terme de temps d'éxecution.
D'autre part les méthodes de puissances itérées ont les pires qualités de couples/valeurs propres typiquement compris entre $[10^{-9}, 10^{-8}]$, 10^7 fois moins précis que la qualité des méthodes subspace pour le min et 10^2 fois pour le max, typiquement la précision des méthode subspace est entre $[10^{-16}, 10^{-6}]$.
En terme de temps d'execution, les méthodes subspace (d'ordre 10^{-1} secondes) sont plus rapides que les méthodes de puissances itérées (d'ordre 10^0 secondes).