,

Rapport Projet Calcul Scientifique

Auteurs :

VIGNAUX Adrien **BLANCHARD Enzo**

Professeurs :

GUIVARCH Ronan **ELOUARD Simon**

Projet de première année de SN année : 2024-2025

Table des matières

1. Introduction

Durant ce module de Calcul Scientifique, il nous a été demandé de réaliser un projet en binôme en utilisant Matlab. Ce dernier s'est déroulé sur deux séances : la première sur l'algorithme de la méthode de la puissance itérée avec différentes améliorations telles que l'approche en blocs et la déflation, puis la seconde sur une application de ces fonctions dans le cas de compression d'images.

2. Première Partie : Power Method Algorithm and Deflation

2.1. Introduction et limitations de la méthode de la puissance itérée

L'objectif de cette première partie est de tester différentes versions de la puissance itérée entre elles et avec la fonction "eig" de Matlab, afin de pouvoir implémenter la méthode la plus efficace lors de la seconde partie.

D'abord, nous avons commencé par comparer le programme "power_v11.m" qui nous était fourni avec la fonction "eig" de Matlab : on remarque que cette dernière est bien plus performante que notre programme. Nous allons donc l'améliorer.

Les coûts de calcul principaux sont les produits vectoriels : réduire leur usage nous permettrait de réduire notablement la durée d'exécution de notre programme. Nous remarquons dans notre programme que nous avons deux produits matriciels A.v qui peuvent être réarrangés de façon à n'en garder qu'un seul tout en ayant les mêmes résultats. En effet, la ligne y = A.v n'est plus utile puisque ce calcul est déjà enregistré lors de l'itération précédente (grâce à la ligne z = A.v), et pour la 1ère itération z est déjà initialisé. Ainsi, nous avons bel et bien des temps d'exécution jusqu'à plus de deux fois plus rapides qu'auparavant, c'est un bon départ. Voici la nouvelle version de ce programme :

POWER_V12.M ():
1 Input: Matrix A $\in R^{n\times n}$
2 Output: (λ_1, v_1) eigenpair associated **to** the largest (**in** module) eigenvalue.
3 v $\in R^n$ given
4
5 $z = A * v$
6 $\beta = v^T * z$
7 **repeat**
8 $\quad v = \frac{z}{\|z\|_2}$
9 $\quad z = A * v$
10 $\quad \beta_{\text{old}} = \beta$
11 $\quad \beta = v^T * z$
12 $\quad$ **until** $|\beta - \beta_{\text{old}}| / |\beta_{\text{old}}| < \varepsilon$
13 $\lambda_1 = \beta \wedge v_1 = v$

Fig. 1. – Algorythme de la puissance itérée avec deflation

Cependant, le principal défaut de cette méthode est sa vitesse de convergence car le nombre de flops de l'algorithme reste conséquent.

Notre objectif est maintenant d'étendre cette méthode à des blocs de couples propres dits dominants.

2.2. Extension de la méthode de la puissance itérée à un espace propre dominant

Cette fois nous allons devoir implémenter cette nouvelle méthode dans un nouveau fichier "subspace_iter_v0.m". La différence ici, c'est que notre algorithme va tendre vers une matrice V contenant les m différents vecteurs propres de A, en passant par une décomposition spectrale et une orthonormalisation de celle-ci.

L'algorithme ressemble grandement au précédent, si ce n'est que les conditions d'arrêt diffèrent, et que l'on doit orthonormaliser. Voici le résultat :

SUBSPACE_ITER_V0.M ():
Input: Symmetric matrix A $\in R^{n\times n}$, number of required eigenpairs m, tolerance ε **and** *MaxtIter* (max nb of iterations)
Output: m dominant eigenvectors V_{out} **and** the corresponding eigenvalues Λ_{out}.

Generate a set of m orthonormal vectors V $\in R^{n\times m}$;

$k = 0$
repeat
    $k = k + 1$
    $Y = A * V$
    $H = VT * A * V$ {ou $H = VT * Y$}
    Compute $acc = \frac{\|A*V-V*H\|}{\|A\|}$
    $V \longleftarrow$ orthonormalisation of the columns of Y
until ($k >$*MaxtIter* **or** $acc \leq \varepsilon$)
Compute the spectral decomposition $X * \Lambda_{\text{out}} * X^T = H$, where the eigenvalues of $H(\text{diag}(\Lambda_{\text{out}}))$ are arranged **in** descending order of magnitude.
Compute the corresponding eigenspace $V_{\text{out}} = V * X$

Fig. 2. – Algorythme de la puissance itérée avec deflation et orthonormalisation

Il est intéressant de noter que, bien que l'on cherche à ne pas faire la décomposition spectrale complète de A, nous effectuons ici celle de H. Pourtant, ce n'est pas un problème : puisque H contient toutes les informations qui nous intéressent à la fin de la boucle, nous n'avons pas besoin d'effectuer la décomposition spectrale de A à chaque itération, mais une seule fois en sortie de boucle .

2.3. Amélioration en utilisant la projection de Rayleigh-Ritz

À présent, nous allons employer la projection de Rayleigh-Ritz pour améliorer les performances de notre code actuel. Voici l'algorithme de cette projection :

RAYLEIGH_RITZ.M ():
$H = V' * (A * V)$
[VH, DH] = eig(H)
[W, indice] = sort(diag(DH), 'descend')

Fig. 3. – Algorythme de projection de Rayleigh-Ritz

Notre programme va naturellement être modifié pour prendre en compte ce changement, notamment au niveau de la condition de sortie basée sur le pourcentage de la trace. Voici donc l'algorithme du fichier "subspace_iter_v1.m" :

SUBSPACE_ITER_V1.M ():

Input: Symmetric matrix A $\in R^{n\times n}$, tolerance ε **and** *MaxtIter* (max nb of iterations) **and** *PercentT* race the target percentage of the trace of A

Output: n_{ev} dominant eigenvectors V_{out} **and** the corresponding eigenvalues Λ_{out}.

Generate an initial set of m orthonormal vectors V $\in R^{n\times m}$;

$k = 0$

$PercentReached = 0$

repeat

  $k = k + 1$

  $Y = A * V$

  $H = VT * A * V$ {ou $H = VT * Y$}

  Compute $acc = \frac{\|A*V-V*H\|}{\|A\|}$

  $V \longleftarrow$ orthonormalisation of the columns of Y

  Rayleigh-Ritz projection applied on matrix A **and** orthonormal vectors V

  Convergence analysis step: save eigenpairs that have converged **and** update *PercentReached*

until ($PercentReached > PercentTrace$ **or** $n_{\text{ev}} = m \vee k > MaxIter$)

Fig. 4. – Algorythme de la puissance itérée avec deflation et orthonormalisation

Les différentes étapes de cet algorithme peuvent être identifiées comme telles :

```
function [ V, D, n_ev, it, itv, flag ] = subspace_iter_v1( A, m, percentage, eps, maxit )
    n = size(A,1);
    W = zeros(m,1);
    itv = zeros(m,1);

    % numéro de l'itération courante

    k = 0; % ligne 6 de la figure 5

    % somme courante des valeurs propres
    eigsum = 0.0;
    % nombre de vecteurs ayant convergés
    nb_c = 0;

    % indicateur de la convergence

    conv = 0; % ligne 7 de la figure 5

    % 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) %boucle repeat de la figure 5

        k = k+1; % ligne 9 de la figure 5

        %% Y <- A*V
        Y = A*Vr; % ligne 10 de la figure 5

        %% orthogonalisation
        Vr = mgs(Y); % ligne 11,12,13 de la figure 5

        %% Projection de Rayleigh-Ritz
        [Wr, Vr] = rayleigh_ritz_projection(A, Vr); % ligne 14 de la figure 5

        %% Quels vecteurs ont convergé à cette itération % ligne 15 de la figure 5
        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;
```

Fig. 5. – Algorithme subspace_iter_v1

Bien que notre programme soit fonctionnel, il est encore possible de l'optimiser, et pour ce faire nous allons implémenter à la fois l'approche en blocs et la méthode de la déflation afin d'accélérer la convergence de notre algorithme.

2.4. Approche en blocs

Nous pouvons constater que l'orthonormalisation, coûteuse algorithmiquement parlant, est complètement effectuée à chaque itération. Pour ce faire, nous allons donc effectuer les p produits à chaque itération ; voici le résultat de cet algorithme :

SUBSPACE_ITER_V2.M ():
Input: Symmetric matrix A $\in R^{n\times n}$, tolerance ε **and** *MaxtIter* (max nb of iterations) **and** *PercentT* race the target percentage of the trace of A
Output: n_{ev} dominant eigenvectors V_{out} **and** the corresponding eigenvalues Λ_{out}.

Generate an initial set of m orthonormal vectors V $\in R^{n\times m}$;

$k = 0$
$PercentReached = 0$
repeat
  $k = k + 1$
  $Y = A^p * V$
  $H = VT * A * V$ {ou $H = VT * Y$}
  Compute $acc = \frac{\|A*V-V*H\|}{\|A\|}$
  $V \longleftarrow$ orthonormalisation of the columns of Y
  Rayleigh-Ritz projection applied on matrix A **and** orthonormal vectors V
  Convergence analysis step: save eigenpairs that have converged **and** update *PercentReached*
until ($PercentReached > PercentTrace$ **or** $n_{\text{ev}} = m \vee k > MaxIter$)

Fig. 6. – Algorythme de la puissance itérée avec deflation et orthonormalisation par block

Ainsi, le coût de A^p étant de (p-1)n^3, et celui de $A^p * v$ de $p * n^3$, en calculant A^p avant la boucle, nous réduisons les coûts de calcul.

Lors de différents tests de la valeur p, nous pouvons constater quelque chose : plus la valeur est importante, moins le nombre d'itérations est important. Au-delà de p = 100, nous commençons à atteindre les limites de cette méthode, qui réduit considérablement le temps de calcul.

2.5. Méthode de la déflation

Finalement, l'ultime amélioration résulte dans le fait de limiter la projection de Rayleigh-Ritz aux colonnes non convergées de V au lieu de toute la matrice.

D'abord, remarquons que dans les deux premières versions, les erreurs ont un rapport de 10^5 entre elles : cela peut s'expliquer du fait que nous continuons à prendre en compte les paires ayant déjà convergées, ce qui les fait davantage converger et ainsi créent cet écart entre les vecteurs.

Pour la version 3, puisque nous ne les prenons plus en compte, cela va alors uniformiser le rapport d'erreurs entre les vecteurs.

Ainsi, voici le programme “subspace_iter_v3.m” :

SUBSPACE_ITER_V2.M ():

Input: Symmetric matrix A $\in R^{n\times n}$, tolerance ε **and** *MaxtIter* (max nb of iterations) **and** *PercentT* race the target percentage of the trace of A

Output: n_{ev} dominant eigenvectors V_{out} **and** the corresponding eigenvalues Λ_{out}.

Generate an initial set of m orthonormal vectors V $\in R^{n\times m}$;

$k = 0$

$PercentReached = 0$

repeat

  $k = k + 1$

  $Y = [V(:, 1 : \text{nbc}) \quad (A^p) * V(:, \text{nbc} + 1 : \textbf{end})]$

  $H = VT * A * V$ {ou $H = VT * Y$}

  Compute $acc = \frac{\|A*V-V*H\|}{\|A\|}$

  $V \longleftarrow$ orthonormalisation of the columns of Y

  Rayleigh-Ritz projection applied on matrix A **and** orthonormal vectors V

  Convergence analysis step: save eigenpairs that have converged **and** update *PercentReached*

until ($PercentReached > PercentTrace \vee n_{\text{ev}} = m \vee k > MaxIter$)

Fig. 7. – Algorythme de la puissance itérée avec deflation et orthonormalisation par block

2.6. Analyse et Comparaison

Voici les différentes distributions des valeurs propres en fonction de imat :

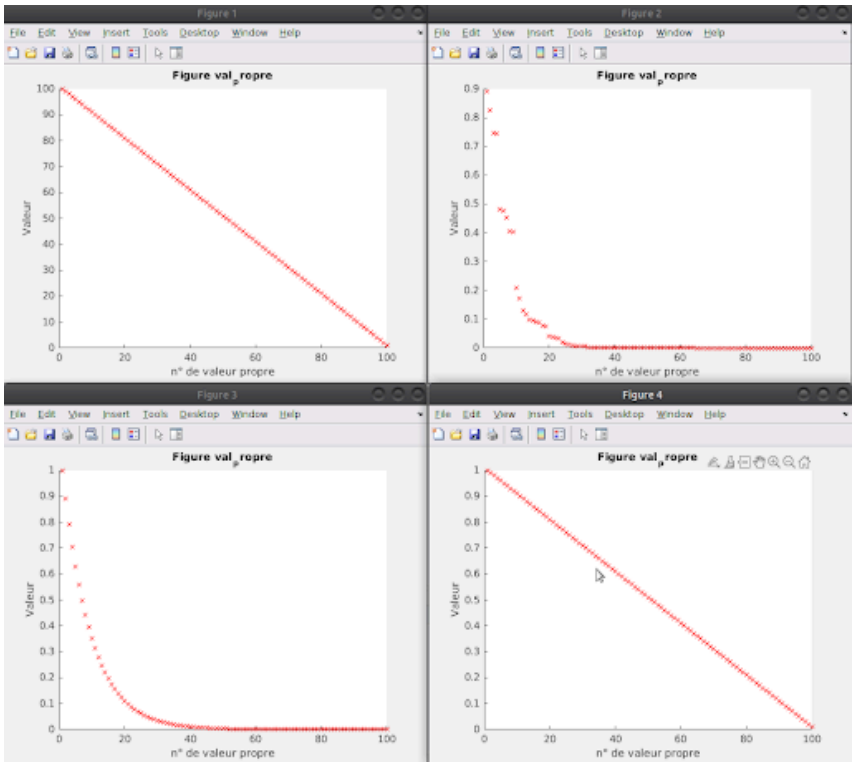

Fig. 8. – Tableau de données TP02

Voici le bilan des tests des différentes versions en fonction du type et de la taille de la matrice (temps et nombre d'itérations) ; Il faut bien penser que la qualité des couples et valeurs propres s'améliore nettement au fur et à mesure des versions (la v0 a un taux de qualité faible, la v1 a un bon taux , et la v2 et v3 ont un excellent taux) :

	v0	v1	v2	v3
Matrice 100x100				
Type 1	30 ms 1 iter.	80 ms 87 iter.	10 ms 1 iter.	0 ms (trop court?) 1 iter.
Type 2	30 ms 1 iter.	10 ms 4 iter.	Convergence non atteinte	Convergence non atteinte
Type 3	40 ms 1 iter.	20 ms 9 iter.	Convergence non atteinte	Convergence non atteinte
Type 4	40 ms 1 iter.	60 ms 87 iter.	10 ms 1 iter.	0 ms (trop court?) 1 iter.
Matrice 200x200				
Type 1	90 ms 1 iter.	270 ms 263 iter.	10 ms 4 iter.	10 ms 4 iter.
Type 2	60 ms 1 iter.	20 ms 8 iter.	30 ms 11 iter.	30 ms 11 iter.
Type 3	60 ms 1 iter.	20 ms 15 iter.	60 ms 28 iter.	50 ms 28 iter.
Type 4	80 ms 1 iter.	300 ms 265 iter.	20 ms 4 iter.	10 ms 4 iter.
Matrice 500x500				
Type 1	510 ms 1 iter.	1.8 s 240 iter.	120 ms 3 iter.	70 ms 3 iter.
Type 2	450 ms 1 iter.	80 ms 8 iter.	Convergence non atteinte	Convergence non atteinte
Type 3	390 ms 1 iter.	130 ms 17 iter.	Convergence non atteinte	Convergence non atteinte
Type 4	550 ms 1 iter.	1.77 s 242 iter.	140 ms 3 iter.	130 ms 3 iter.

Fig. 9. – Tableau de données TP02

3. Seconde Partie : Subspace Iteration Method

3.1. Présentation des objectifs

Dans cette partie, nous allons à présent employer les méthodes vues précédemment pour de la compression d'image. Cela revient à employer deux théorèmes : la Décomposition en Valeur Singulière (appelée SVD), et l'approximation du meilleur plus-petit rang. Nous avons à notre disposition différentes images d'une page de BD sous différentes formes : portrait, paysage, deux portraits, en couleur. L'objectif va être d'utiliser nos fonctions précédentes pour vérifier leurs performances sur ce cas concret.

3.2. Implémentation de la reconstruction d'images en fonction des différents programmes

En premier temps, il est important de noter que :

- Σ_k est de taille $k * k$
- U_k est de taille $q * k$
- V_k est de taille $p * k$

De plus, lorsque q $<$ p, l'image passe en format paysage. À présent, voici l'algorithme en question:

ReconstructionImage.m ():

```
Input: Image to reconstruct

Read and Preprocess the Image
I = imread(‹ BD_Asterix_Colored.jpg ›)
I = rgb2gray(I)
I = double(I)
[q, p] = size(I)

Perform SVD
[U, S, V] = svd(I)

Image Reconstruction Using Rank-k Approximations
inter = 1:40:200
inter(end) = 200
differenceSVD = zeros(size(inter,2), 1)
for k = inter
    ℑ_k = U(:, 1 : k) * S(1 : k, 1 : k) * V(:, 1 : k)'
    differenceSVD(k) = sqrt(sum(sum((I − ℑ_k).^2)))
end

Eigenvalue Decomposition Approach
A = I' * I
[V, S] = eig(A)
[S,ind] = sort(diag(S),‹ descend ›)
S = diag(S)
V = V(:,ind)
sigma = sqrt(S)
U = (I * V) * inv(sigma)

RMSE Comparison
for k = inter
    ℑ_k = U(:, 1 : k) * S(1 : k, 1 : k) * V(:, 1 : k)'
    differencePower(k) = sqrt(sum(sum((I − ℑ_k).^2)))
end
plot(inter, differenceSVD, ‹ rx ›)
xlabel(‹ rank k ›)
ylabel(‹ RMSE ›)
```

Fig. 10. – Algorythme de la puissance itérée avec deflation

3.3. Résultats obtenus

	power_v12	v0	v1	v2	v3	eig
Spirou 2 (p = 0.99)	ECHEC?	22.8s	5.1s	4.2s	4.2s	1.5s
Asterix 2 (p = 0.999)	ECHEC?	FLOU	2.2s	1.5s	1.5s	0.8s
Asterix 1 (p = 0.999)	ECHEC?	FLOU	42.2s	36.5s	25.8s	1.6s
Asterix Colorié	ECHEC?	FLOU	2.3s	1.7s	1.9s	1.2s

Fig. 11. – Tableau de données TP02

Les images en couleurs perdent de leurs couleurs, cependant elles se reconstruisent bien.

4. Conclusion

Finalement, ce projet nous a permis de développer une première méthode de calcul puis de l'optimiser jusqu'au maximum de nos compétences. De plus, nous avons pu les tester en situation réelle lors de compression d'images (sous différents formats).

Malgré toutes nos tentatives, il nous a été impossible d'égaler eig : ce qui est normal, puisque cette dernière est optimisée jusqu'à une profondeur de code dont nous n'avions pas accès.