



Projet Calcul Scientifique

Aya Rifai

Fadwa ELLAIK

CD11

Sommaire

1	Introduction	2
2	Méthodes d'Iteration de Sous-Espace	3
2.1	Question 1.1	3
2.2	Question 1.2	4
2.3	Question 1.3	5
2.4	Question 1.4	5
2.5	Question 1.5	5
2.6	Question 1.6	5
2.7	Question 1.7	6
2.8	Question 1.8	6
2.9	Question 1.9	7
2.10	Question 1.10	7
2.11	Question 1.11	8
2.12	Question 1.12	8
2.13	Question 1.13	8
2.14	Question 1.14	9
2.15	Question 1.15	10
3	Application à la compression des images	12
3.1	Question1	12
3.2	Question2	12

1 Introduction

Dans ce projet, nous nous intéressons à un problème fondamental en mathématiques appliquées et en informatique : le calcul des valeurs propres d'une matrice. Ce type de calcul joue un rôle central dans de nombreux domaines, allant de la mécanique à l'apprentissage automatique, en passant par le traitement du signal et l'analyse de données.

La première partie de notre travail a consisté à explorer et comparer plusieurs méthodes numériques classiques pour approximer les valeurs propres d'une matrice symétrique. Nous avons ainsi étudié des approches itératives comme la méthode de la puissance ainsi que les variantes de l'itération sur sous-espace. L'objectif était de mieux comprendre leurs mécanismes, leurs performances, mais aussi leurs limites selon la nature des matrices traitées (spectre, taille, conditionnement...).

Dans un second temps, nous avons mis en pratique ces méthodes dans une application concrète : la compression d'images. L'idée est d'exploiter la structure spectrale des matrices associées aux images pour en extraire l'information essentielle tout en réduisant la taille des données.

Nous présentons donc dans ce rapport, dans un premier temps, une étude comparative des différentes méthodes d'approximation de valeurs propres, appuyée par des expérimentations numériques. Dans un second temps, nous montrons comment ces méthodes peuvent être utilisées pour compresser efficacement des images tout en préservant leur qualité visuelle.

2 Méthodes d’Itération de Sous-Espace

2.1 Question 1.1

La méthode `power_v11` (puissance itérée) permet de calculer la valeur propre dominante d’une matrice.

Nous avons lancé le test `test_v11`. Il nous a permis de comparer le temps d’exécution des méthodes `eig` et `power_v11` pour les 4 types de matrices et pour des tailles différentes.

Ci-dessous, le tableau représentant les différents résultats (Table 2).

Type de matrice	Taille	Temps eig (s)	Temps power_v11 (s)
Type 1	100×100	1.000e-02	7.000e-02
	300×300	5.000e-02	7.380e+00
	500×500	7.000e-02	3.377e+01
	700×700	1.200e-01	convergence non atteinte
Type 2	100×100	0.000e+00	1.000e-02
	300×300	2.000e-02	7.000e-02
	500×500	6.000e-02	1.400e-01
	700×700	1.300e-01	5.700e-01
Type 3	100×100	0.000e+00	1.000e-02
	300×300	4.000e-02	2.500e-01
	500×500	6.000e-02	9.400e-01
	700×700	1.100e-01	1.880e+00
Type 4	100×100	1.000e-02	8.000e-02
	300×300	5.000e-02	6.400e+00
	500×500	1.500e-01	3.379e+01
	700×700	1.900e-01	convergence non atteinte

Table 1: Temps d’exécution pour les méthodes `eig` et `power_v11`

On constate alors que la méthode `eig`, native de MATLAB, est plus robuste et possède un temps d’exécution court pour tous les types de matrices et pour toutes les tailles.

En revanche, la méthode `power_v11` de puissance itérée a un temps d’exécution bien plus élevé et dépend fortement du spectre. Elle converge bien pour les matrices de type 2 et les matrices de petites tailles, mais elle échoue ou devient lente pour les matrices de types 1 et 4. À partir de la taille 700×700, elle ne converge plus dans certains cas.

Globalement, `eig` reste plus fiable et efficace.

2.2 Question 1.2

La méthode powerv11 présente un inconvénient de calcul double du produit matrice vecteur. C'est ce que nous avons essayé d'optimiser dans la méthode powerv12. Ci-dessous la partie modifiée de l'algorithme précédent.

```

k = 0;
while (~convg && k < m)
    k = k + 1;
    % méthode de la puissance itérée
    v = randn(n,1);
    z = A*v;
    beta = v'*z;

    % convg = || 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)/norm(beta, 2);
        nb_it = nb_it + 1;
    end

```

Figure 1: Algorithme power_v12

Nous avons lancé le test **test_v11v12**. Il nous a permis de comparer le temps d'exécution des méthodes **power_v11** et **power_v12** pour les 4 types de matrices et pour des tailles différentes.

Ci-dessous, le tableau représentant les différents résultats (Table 2).

Type de matrice	Taille	Temps power_v11 (s)	Temps power_v12 (s)
Type 1	100×100	2.100e-01	1.500e-01
	300×300	2.421e+01	1.042e+01
	500×500	pourcentage non atteint	pourcentage non atteint
	700×700	pourcentage non atteint	pourcentage non atteint
Type 2	100×100	2.000e-02	1.000e-02
	300×300	5.500e-01	3.100e-01
	500×500	2.260e+00	1.320e+00
	700×700	6.510e+00	2.450e+00
Type 3	100×100	2.000e-02	0.000e+00
	300×300	7.700e-01	4.800e-01
	500×500	3.650e+00	1.670e+00
	700×700	1.153e+01	6.150e+00
Type 4	100×100	2.400e-01	1.600e-01
	300×300	2.111e+01	1.037e+01
	500×500	pourcentage non atteint	pourcentage non atteint
	700×700	pourcentage non atteint	pourcentage non atteint

Table 2: Temps d'exécution pour les méthodes **power_v11** et **power_v12**

On constate alors que la méthode de puissance itérée améliorée est deux fois plus rapide que la méthode de base. Malgré cela, on peut voir qu'il y a des tailles de matrices pour lesquelles le pourcentage est non atteint.

2.3 Question 1.3

La méthode de la puissance avec déflation nécessite d'appliquer plusieurs fois la méthode de puissance de manière séquentielle. À chaque étape, on calcule une valeur propre et le vecteur propre correspondant, puis on continue les calculs jusqu'à convergence. Cela implique qu'on répète plusieurs fois le processus d'itération avec une mise à jour du couple propre dominant à chaque étape, ce qui multiplie le coût et conduit alors à un temps d'exécution très élevé, notamment pour les matrices de grandes tailles.

2.4 Question 1.4

Si l'on applique directement l'algorithme 1 à m vecteurs, on converge vers une matrice V dont les colonnes représentent toutes le même vecteur propre dominant, celui associé à la plus grande valeur propre en module.

Puisqu'à chaque itération le vecteur v est multiplié par A , la composante associée à la plus grande valeur propre va croître exponentiellement plus vite que les autres. Ainsi, tous les vecteurs tendront à s'aligner sur le vecteur propre v_1 associé à λ_1 .

2.5 Question 1.5

La matrice H est de taille $m \times m$ où $m \ll n$. Ici, m correspond au nombre de valeurs propres dominantes que l'on souhaite approximer, et qui est très petit par rapport à la taille réelle n de A .

La décomposition spectrale d'une matrice de petite taille est peu coûteuse en temps de calcul et permet, via la méthode de Rayleigh-Ritz, d'obtenir une bonne approximation des valeurs propres dominantes de A . C'est pour cela que ce calcul ne pose pas problème dans l'algorithme 2.

2.6 Question 1.6

Ci-dessous les lignes de code ajoutées pour compléter l'algorithme **subspace_iter_v0**.

```
% on génère un ensemble initial de m vecteurs orthogonaux
V = randn(n,m);
V = ngs(V);
% rappel : conv = invariance du sous-espace V : ||AV - VH|| / ||A|| <= eps
while (~conv && k < maxit)
    k = k + 1;
    % calcul de Y = A.V
    Y = A'*V;
    % calcul de H, le quotient de Rayleigh H = V*Y/A.V
    H = V'*Y;
    % vérification de la convergence
    conv = (norm(Y - V'*H) <= eps*norm(A));
    % orthonormalisation
    V = ngs(Y);
end
% décomposition spectrale de H, le quotient de Rayleigh
[VH,DH] = eig(H);
```

Figure 2: Algorithme **subspace_iter_v0**

2.7 Question 1.7

L'algorithme subspace_iter_v1 :

Algorithm 1 Subspace iteration method v1 with Raleigh-Ritz projection

Input: Symmetric matrix $A \in \mathbb{R}^{n \times n}$, tolerance ε , $MaxIter$ (max nb of iterations) and $PercentTrace$ 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 $V \in \mathbb{R}^{n \times m}$; $k = 0$; $PercentReached = 0$

repeat

$k = k + 1$

 Compute Y such that $Y = A \cdot V$ // ligne 56

$V \leftarrow$ orthonormalisation of the columns of Y // ligne 58

Rayleigh-Ritz projection applied on matrix A and orthonormal vectors V // ligne 61

Convergence analysis step: save eigenpairs that have converged and update $PercentReached$ // lignes 64-132

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

2.8 Question 1.8

Le calcul de A^p suivi du produit $A^p \cdot V$ est très coûteux en termes de *flops*. Tout d'abord, calculer A^p explicitement nécessite $p - 1$ multiplications matricielles. Chaque multiplication entre deux matrices $n \times n$ coûte environ $2n^3$ flops. Le coût total pour obtenir A^p est donc approximativement $2(p - 1)n^3$ flops.

Ensuite, multiplier A^p par V (où $V \in \mathbb{R}^{n \times m}$) coûte $2n^2m$ flops. Au total, cela représente un coût complexe et élevé.

Plutôt que de calculer A^p explicitement, on peut réorganiser le calcul en effectuant p produits successifs entre A et V de cette manière :

```
for i = 1:p
    Y = A * Y;
end
```

Ce procédé ne nécessite que p produits matrice-vecteur (ou matrice-matrice selon le contexte), chacun coûtant environ $2n^2m$ flops. Le coût total devient alors $2pn^2m$, qui est bien plus raisonnable que celui de $A^p \cdot V$ avec calcul explicite de A^p .

2.9 Question 1.9

Ci-dessous la modification apportée au code. Nous avons utilisé le calcul explicite de A^p malgré sa complexité.

```

1+12 matgen_cspd.m | dat_ip.m | test_v2v1v2v3.m | comp_methods.m | subspace_iter_v2.m | subspace_iter_v3.m | subspace_iter_v0.m | + |
2+13 % range : conv = [epsilon>= trace] | (nb_c == n)
3+14 while (~conv && k < maxit)
4+15 k = k+1;
5+16 A = sp2mat(A);
6+17 V = A\W;
7+18 Y = (A'q)\V;
8+19 % Normalisation
9+20 Xk = q*Y/V;
10+21 % R = A*Xk;
11+22 % Projection de Rayleigh-Ritz
12+23 [Wr, Vr] = rayleigh_ritz.projection(A, Vr);
13+24 % Quels vecteurs ont convergé à cette itération
14+25 analyse_cvrg_finite = 0;
15+26 % Nombre de vecteurs ayant convergé à cette itération
16+27 nb_cvrg = 0;
17+28 % nb_c est le dernier vecteur à avoir convergé à l'itération précédente
18+29 nb_c = nb_cvrg;
19+30 p = nb_cvrg;
20+31 % nb_cvrg = nb_cvrg + 1;
21+32 while (~conv && k < maxit)
22+33 % Tous les vecteurs de notre sous-espace ont convergé
23+34 % on a fini (sans avoir obtenu le pourcentage)
24+35 if(k > n)
25+36 analyse_cvrg_finite = 1;
26+37 end

```

Figure 3: Enter Caption

2.10 Question 1.10

À partir de certaines valeurs de p , nous avons exécuté le test pour comparer les temps d'exécution pour une matrice de taille 100×100 pour chacun des quatre types de matrices.

Type	p	Nombre itérations v2	Temps d'exécution v2
1	10	9	1.700e-01
	30	3	1.000e-01
	50	2	1.000e-01
	70	2	5.000e-02
2	10	5	7.000e-02
	30	31	1.200e-01
	50	convergence non atteinte	convergence non atteinte
	70	convergence non atteinte	convergence non atteinte
3	10	2	3.000e-02
	30	2495	5.090e+00
	50	184	3.200e-01
	70	convergence non atteinte	convergence non atteinte
4	10	9	1.200e-01
	30	3	5.000e-02
	50	2	7.000e-02
	70	2	7.000e-02

Table 3: Temps d'exécution (en secondes) pour une matrice 300×300 , selon le type de matrice et la valeur de p

Nous constatons alors que pour les matrices de type 1 et 4, l'augmentation de p réduit le nombre d'itérations avec un temps d'exécution stable voire diminué, ce qui montre un gain d'efficacité modéré.

Pour les matrices de types 2 et 3, la méthode devient très lente voire échoue pour des valeurs de p grandes. Cela montre que ces matrices sont fortement sensibles à l'amplification des erreurs numériques dues à l'exponentiation.

Un compromis est alors nécessaire pour éviter une croissance du conditionnement de A^p , qui dégrade fortement la stabilité numérique de l'approche.

2.11 Question 1.11

Avec la méthode **subspace_iter_v1**, bien que la convergence globale soit atteinte pour des vecteurs propres, la précision de chaque vecteur propre individuel varie. Cela s'explique par le fait que l'algorithme ne vérifie pas séparément le critère de convergence pour chaque vecteur de V , mais plutôt un critère global.

Ainsi, les vecteurs associés aux valeurs propres maximales convergent plus rapidement. À la fin de l'algorithme, certains vecteurs respecteront bien la condition de convergence

$$\frac{\|A \cdot v - \beta \cdot v\|}{|\beta|} < \varepsilon,$$

mais d'autres ne la respecteront pas, ce qui entraîne une différence de précision entre les couples propres calculés.

On trouve par exemple, dans le cas d'une matrice de type 1 et de taille 100×100 , la différence de qualité suivante entre deux couples propres par rapport au critère d'arrêt :

$$[4.224e-09, 6.019e-08]$$

2.12 Question 1.12

La méthode **subspace_iter_v3** introduit une stratégie de déflation qui suggère que les vecteurs propres déjà convergés sont figés et mis de côté, ce qui permet de concentrer les efforts sur les vecteurs restants. Cette approche permet de réduire le coût des opérations tout en accélérant la convergence des vecteurs non encore stabilisés.

Puisque la projection de Rayleigh-Ritz est toujours utilisée, la méthode produit toujours des couples propres de bonne qualité.

Pourtant, les vecteurs associés à des valeurs propres non dominantes — et donc qui convergent plus lentement — peuvent être moins bien orthogonalisés, ce qui peut induire des erreurs numériques et une perte progressive d'orthogonalité, menant à une moindre précision sur ces derniers vecteurs calculés.

2.13 Question 1.13

Ci-dessous la modification apportée à **subspace_iter_v2** pour obtenir **subspace_iter_v3**:

```

+10 | test_v11v12.m | test_v01.m | test_v01v2.m | subspace_iter_v3.m | mgs.m | mgs_block.m | verification_qualite.m | matgen_csdad.m | 4
49 % on génère un ensemble initial de m vecteurs orthogonaux
50 V = zeros(n,m);
51 Vr = rmpf(Vr);
52 
53 % rappel : conv = (epsilon >= trace) | (nb_c == n)
54 while (~conv && k < maxit)
55 
56     k = k+1;
57     %M Y <- A\V
58     Y = A\B;
59     Y(:,nb_c+1:n) = (A'\p)\*(V(:,nb_c+1:n));
60 
61     %M orthogonalisation
62     Vr = rmpf(Y);
63 
64     %M Projection de Rayleigh-Ritz
65     [Wr, Vr] = rayleigh_ritz_projection(A, V);
66 
67     %M Quels vecteurs ont convergé à cette itération
68     analyse_cvg_final = 0;
69     %M Nombre de vecteurs ayant convergé à cette itération
70     nb_cvg = 0;
71     %M nb_c est le dernier vecteur à avoir convergé à l'itération précédente
72     i = nb_c + 1;
73 
```

Figure 4: `subspace_iter_v3`

2.14 Question 1.14

Pour répondre à cette question, nous avons tracé la distribution des valeurs propres de matrices de chaque type en fonction de la taille.

Ci-dessous les résultats:

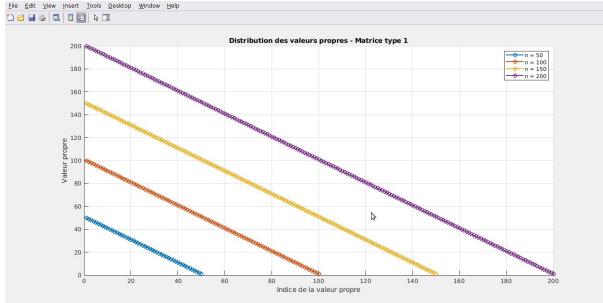


Figure 5: Distribution des valeurs propres pour le type 1

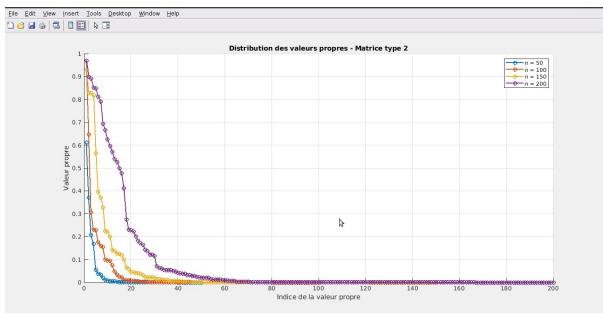


Figure 6: Distribution des valeurs propres pour le type 2

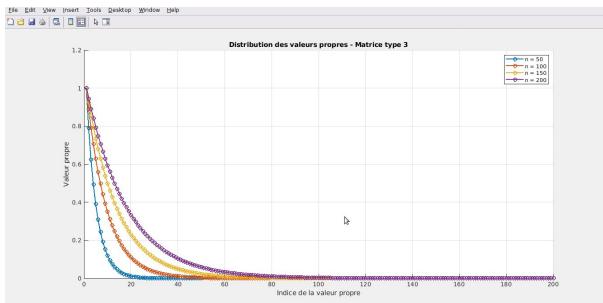


Figure 7: Distribution des valeurs propres pour le type 3

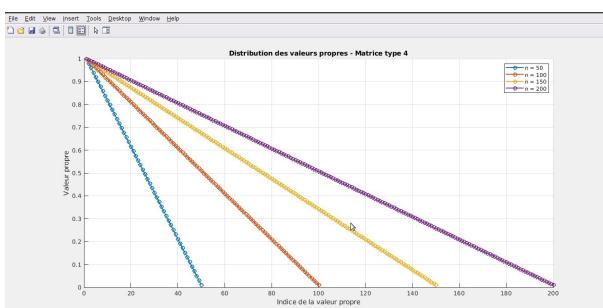


Figure 8: Distribution des valeurs propres pour le type 4

On remarque que pour les matrices de type 1, les valeurs propres décroissent d'une manière régulière et linéaire. Cette répartition est caractéristique d'une matrice diagonale définie positive à spectre décroissant régulier, ce qui simplifie les calculs et favorise la convergence rapide des méthodes itératives.

Pour le type 2, les premières valeurs propres sont dominantes, suivies d'une chute rapide. Cette décroissance exponentielle indique que l'énergie spectrale est conservée dans les premiers vecteurs propres. Cela rend ce type assez inefficace pour les méthodes itératives. Le type 3 montre une décroissance un peu plus lente que celle du type 2, mais conserve la forme exponentielle, ce qui reflète une situation intermédiaire : la plupart des valeurs propres sont faibles, mais quelques-unes restent significatives sur une plage plus large que pour le type 2.

Quant au type 4, les valeurs propres décroissent de façon linéaire mais dans un intervalle très réduit (entre 0 et 1).

2.15 Question 1.15

Pour répondre à cette question, nous avons tracé, pour les 4 types de matrices, et pour les méthodes vues pendant les étapes précédentes, l'évolution du temps d'exécution de la méthode en fonction de la taille de la matrice.

Ci-dessous les images:

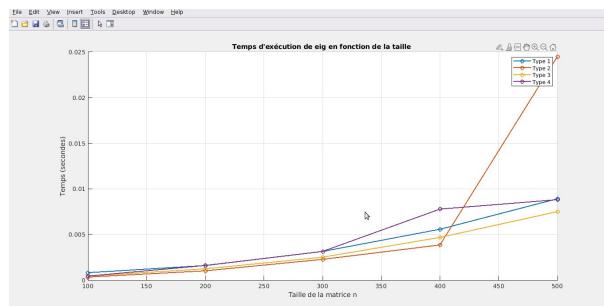


Figure 9: Temps d'exécution en fonction de la taille de matrice de la méthode eig

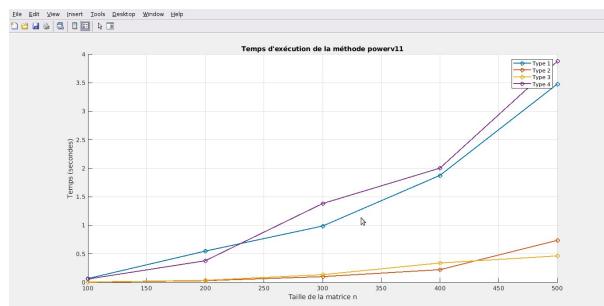


Figure 10: Temps d'exécution en fonction de la taille de matrice de la méthode powerv11

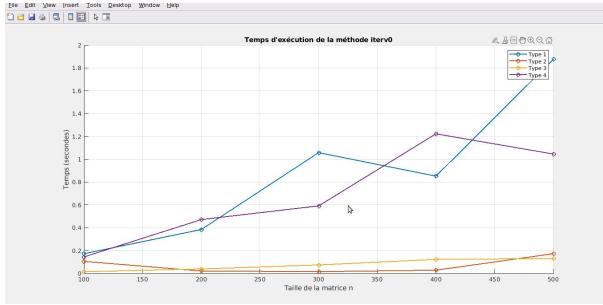


Figure 11: Temps d'exécution en fonction de la taille de matrice de la méthode subspace_iter_v0

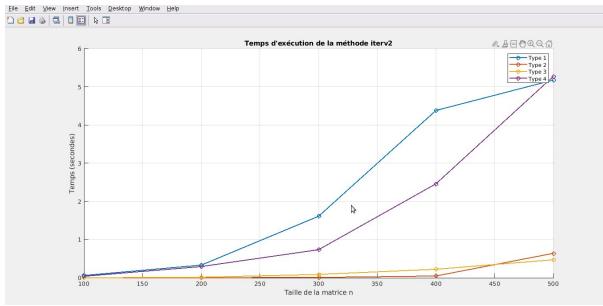


Figure 12: Temps d'exécution en fonction de la taille de matrice de la méthode subspace_iter_v1

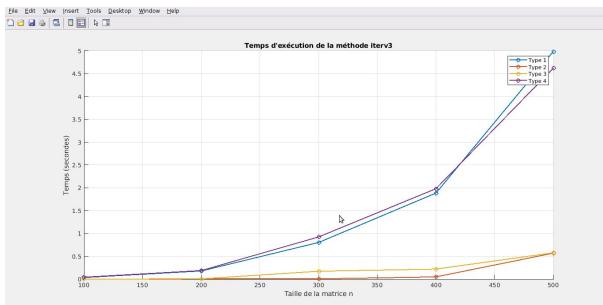


Figure 13: Temps d'exécution en fonction de la taille de matrice de la méthode subspace_iter_v3

On remarque que la **méthode eig** est la plus rapide globalement (temps < 0.015 s), quelle que soit la taille ou le type. Le temps croît presque linéairement avec la taille. On constate aussi que cette méthode est très bien optimisée, surtout pour les petites matrices.

La **méthode power_v11** possède un temps bien plus élevé que **eig**, surtout pour les matrices de type 1 et 4. On remarque que le type 2 est bien plus rapide à converger, car il possède une valeur propre dominante bien isolée.

La **méthode subspace_iter_v0** présente un temps d'exécution intermédiaire, plus stable entre les différents types. Le temps croît plus vite avec la taille que pour **eig**, mais reste raisonnable.

La **méthode subspace_iter_v1** est similaire à **subspace_iter_v0** mais légèrement plus coûteuse. Elle est toutefois plus stable sur certains types de matrices.

Enfin, la méthode `subspace iter v2` est plus lente que les autres, surtout pour les matrices de type 1 et 4. Cela est dû à l'orthonormalisation répétée et aux produits successifs de la forme $A^p \cdot V$. On en déduit qu'elle est très sensible au type de matrice, avec des performances mauvaises sur les spectres étalés.

3 Application à la compression des images

Dans cette deuxième partie du projet de calcul scientifique, nous explorons l'application des méthodes d'itération sur les sous-espaces à la compression d'images en niveaux de gris. L'objectif principal est de comparer différentes méthodes de décomposition matricielle, notamment la SVD classique, l'utilisation de `eig`, et les variantes d'itérations sur les sous-espaces, afin d'évaluer leur efficacité en termes de qualité de reconstruction et de temps d'exécution.

Question 1 :

Soit une image représentée par une matrice I de taille $q \times p$. Si l'on effectue une approximation de rang k avec $k < p < q$, les dimensions des matrices issues de la décomposition sont les suivantes :

- Σ_k : matrice diagonale de taille $k \times k$ contenant les k premières valeurs singulières (c'est-à-dire k éléments non nuls).
- U_k : matrice de taille $q \times k$, composée des k premiers vecteurs propres de II^\top (soit qk éléments).
- V_k : matrice de taille $p \times k$, composée des k premiers vecteurs propres de $I^\top I$ (soit pk éléments).

Dans l'algorithme, le choix de la matrice à utiliser dépend des dimensions de l'image $I \in \mathbb{R}^{p \times q}$:

- Si l'image est en orientation portrait, c'est-à-dire plus haute que large ($q < p$), on travaille avec la matrice $I^\top I \in \mathbb{R}^{q \times q}$.
- Si l'image est en orientation paysage (plus large que haute) ou carrée, on utilise la matrice $II^\top \in \mathbb{R}^{p \times p}$.

Ce choix permet de limiter la taille de la matrice utilisée pour le calcul spectral (valeurs propres et vecteurs propres), car l'une des dimensions (p ou q) est plus petite. Ainsi, cette approche optimise le temps de calcul tout en préservant les mêmes valeurs singulières.

Question 2 :

La figure ci-dessous montre comment l'erreur de reconstruction (RMSE) diminue avec l'augmentation du nombre de vecteurs singuliers utilisés (de 1 à 200), indiquant une amélioration de la qualité de l'image reconstruite.

Les résultats sont illustrés à l'aide de cinq images différentes : `BD_Asterix_1`, `BD_Asterix_2`, `BD_Asterix_Colored`, `BD_Spirou_1` et `BD_Spirou_2`.

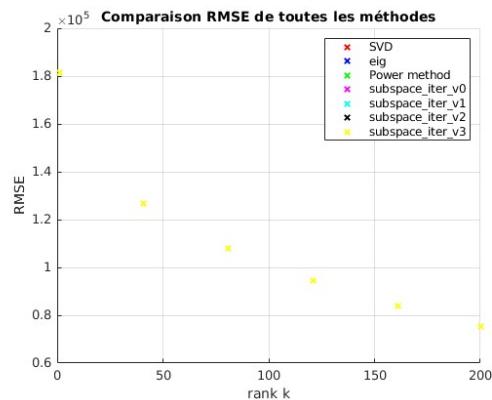
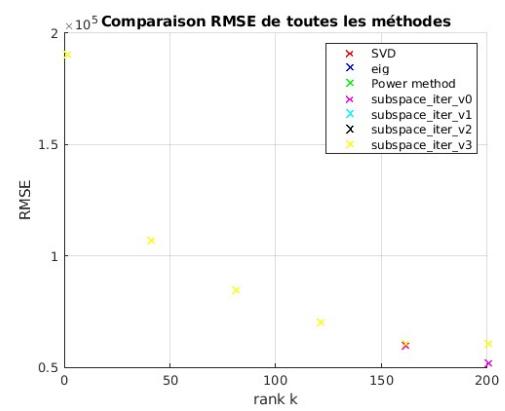
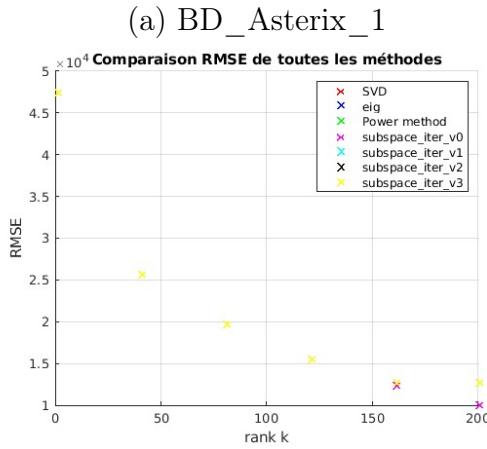
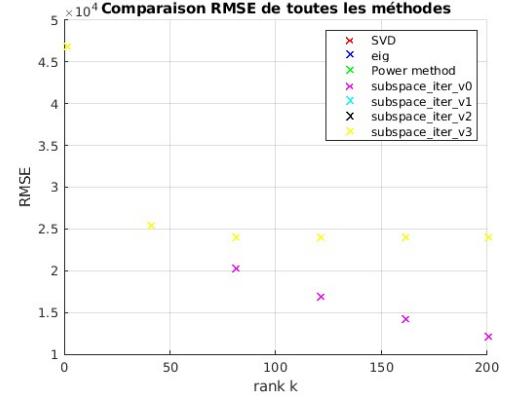
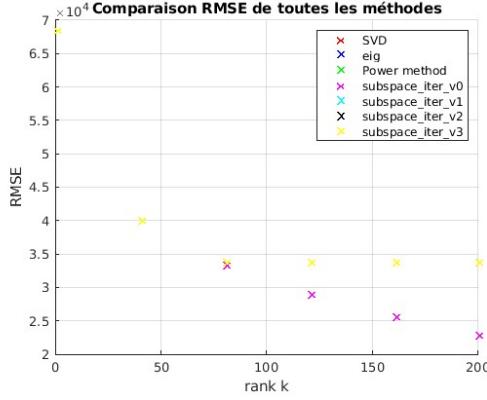


FIGURE 1 – Erreur de reconstruction des différentes images pour les différentes méthodes en fonction du rang k

Comparaison des temps d'exécution

Les temps d'exécution pour chaque méthode sont listés dans le tableau ci-dessous :

Méthode \ Image	Asterix_1	Asterix_2	Asterix_Colored	Spirou_1	Spirou_2	Thorgal_1
SVD standard	0.5227	0.2134	0.4945	10.7486	7.7875	49.1447
eig	0.7327	0.4201	0.1855	13.9077	8.4327	75.3316
Power Method	101.6421	47.5244	12.1364	895.7618	705.2866	2985.5374
subspace_iter_v0	55.8791	34.8939	27.1263	497.0961	238.0532	1332.8471
subspace_iter_v1	1.5551	0.8893	0.8126	6.5848	7.7089	33.2641
subspace_iter_v2	1.3608	0.8323	0.9025	6.7062	7.7404	16.3923
subspace_iter_v3	1.6197	0.9120	0.8485	19.5966	7.6909	18.7645

TABLE 1 – Comparaison des temps d'exécution (en secondes) pour différentes méthodes et images

Observations :

On observe que les méthodes directes comme la SVD standard et la fonction `eig` de MATLAB sont globalement les plus rapides, même pour des images de grande taille. Les méthodes d'itération sur les sous-espaces affichent des performances variables : la méthode de puissance itérée, bien qu'intéressante sur le plan théorique, est extrêmement lente, en particulier pour les grandes images comme `Spirou_1` et `Thorgal_1`. À l'inverse, les variantes `subspace_iter_v1`, `v2`, et `v3` montrent une efficacité remarquable, approchant les performances des méthodes directes tout en étant adaptées à un traitement progressif et à des contextes nécessitant un ajustement du rang sans recalculation complet. Enfin, les temps d'exécution augmentent de façon prévisible avec la taille de l'image, surtout pour les méthodes itératives.

4 Conclusion

Ce projet a permis de comparer plusieurs techniques de compression d’images basées sur des décompositions matricielles. Les méthodes directes, comme la SVD standard et `eig`, offrent les meilleures performances en rapidité. Toutefois, certaines méthodes itératives, notamment les variantes de sous-espaces (`subspace_iter_v1`, `v2`, `v3`), se révèlent également efficaces et flexibles, notamment pour un ajustement progressif du rang. Le choix de la méthode doit ainsi dépendre du compromis entre précision, rapidité et contexte d’application.