Rapport

# Projet de calcul scientifique

*réalisé par*

Samy Afker & Maxime Moshfeghi

ENSEEIHT - 1SN - Calcul scientifique - 2022 - 2023

# Table des matières

# Introduction

Dans ce projet, le but sera d'implanter différents algorithmes de décomposition spectrale d'une matrice. Nous allons implanter deux versions de la méthode de la puissance itérée ainsi que 4 versions de la méthode "subspace iteration". Ensuite, nous allons comparer les performances des différents algorithmes pour différents paramètres. Enfin, nous allons les appliquer pour compresser un image.

# Première partie

# Séance 1

# Chapitre 1

# Limites de la puissance itérée

## 1 Temps d'exécution

**Question 1 :** En terme de temps d'exécution, la méthode de la puissance itérée implantée en suivant l'algorithme 1 donnée dans le sujet est moins efficace que la méthode `eig` existant initialement dans **Matlab**. Elle est en effet 10 à 100 fois moins efficace, selon les types des matrices et leurs tailles.

**Voir le tableau comparatif des temps d'exécution**

## 2 Optimisation de l'algorithme

**Question 2 :** En inspectant l'implantation de l'algorithme, on observe que celui-ci peut-être allégé d'une multiplication matrice×vecteur. Voici comment :

Listing 1.1 – Boucle `Répéter` de l'algorithme 1 optimisé

```
    while(norme > eps && nb_it < maxit)
      % y = z;
      % v = y / norm(y,2);
      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
```

Cette nouvelle implémentation nécessite bien sûr une initialisation de z avant la boucle.

On peut désormais comparer les vitesses d'exécution des trois programmes :

TABLE 1.1 – Temps d'exécution pour chacun des algorithmes

| Propriétés de la matrice | Type | 1 | | 2 | | 3 | | 4 | |
|---|---|---|---|---|---|---|---|---|---|
| | Taille | 200 | 400 | 200 | 400 | 200 | 400 | 200 | 400 |
| Algorithme | `eig` | 8,00E-02 | 3,00E-02 | 1,00E-02 | 2,00E-02 | 2,00E-02 | 5,00E-02 | 2,00E-02 | 3,00E-02 |
| | `powerv11` | 5,75E+00 | 4,19E+01 | 9,00E-02 | 1,20E+00 | 2,00E-01 | 1,11E+00 | 4,97E+00 | 4,29E+01 |
| | `powerv12` | 3,16E+00 | 2,24E+01 | 7,00E-02 | 8,30E-01 | 1,50E-01 | 7,20E-01 | 3,22E+00 | 2,30E+01 |

On remarque que la fonction `eig.m` est la meilleure en terme de temps d'exécution. En ce qui concerne les méthodes de la puissance itérée, on observe que la fonction `powerv12.m` est en effet 2 fois plus rapide que `powerv11.m` .

**Question 3 :**

Cependant, cette méthode présente un principal défaut : l'algorithme effectue la décomposition spectrale complète de la matrice. Ce processus peut alors s'avérer très long (complexité en $\mathcal{O}(n^2)$).

# Chapitre 2

# Amélioration de la méthode de la puissance itérée pour obtenir les sous-espaces propres dominants

**Question 4 :**

La matrice V converge vers une matrice contenant les vecteurs propres de A exprimés dans une autre base. On obtient la matrice de passage X permettant de retrouver les vecteurs propres de A dans la base canonique en effectuant la décomposition spectrale de H (à l'aide de la fonction `eig.m`). Ainsi, on retrouve alors les vecteurs propres en calculant $V_{out} = V \cdot X$

**Question 5 :**

Il est raisonnable d'effectuer une décomposition spectrale complète de H car il s'agit d'une matrice beaucoup plus petite que A (on a $H \in \mathbb{R}^{m \times m}$).

**Question 6 :**

On implémente alors l'algorithme sur **Matlab** de la manière suivante :

**Voir `subspace_iter_v0.m` en annexe**

**Question 7 :**

L'ensemble des étapes de l'algorithme 4 sont identifiées via les commentaires du programme **Matlab** suivant :

**Voir `subspace_iter_v1.m` en annexe**

# Chapitre 3

# subspace_iter_v2 et subspace_iter_v3

**Question 8 :**

L'opération $A \times A$ a une complexité de $\mathcal{O}(n^3)$. On peut montrer par récurrence immédiate que la complexité du calcul de $A^p$ est $\mathcal{O}((p-1)n^3)$. Ainsi, la complexité totale du calcul de $A^p \cdot V$ est de $\mathcal{O}((p-1)n^3 + mn^2)$

**Question 9 :**

La seule modification à apporter par rapport à `subspace_iter_v2.m` est la suivante :

Listing 3.1 – Amélioration de `subspace_iter_v1.m`

```
%% Y <- A^p*V
Y = Vr;
for k=1:p
    Y = A*Y;
end
```

**Question 10 :**

Afin d'étudier le comportement de `subspace_iter_v2.m` en fonction de p, on lance le script `test_v0_v1_v2.m` pour des matrices de taille 400, de types différents et ceci pour différentes valeurs de p :

Table 3.1 – Temps d'exécution pour chacun des algorithmes

| Propriétés de la matrice | Type | 1 | | | 2 | | | 3 | | | 4 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Taille | 400 | | | | | | | | | | | |
| | Puissance | 5 | 20 | 50 | 5 | 10 | 15 | 5 | 15 | 30 | 5 | 20 | 50 |
| Algorithme | `subspace_iter_v0` | 9,23E+00 | | | 3,90E-01 | | | 1,19E+00 | | | 7,22E+00 | | |
| | `subspace_iter_v1` | 7,40E-01 | | | 3,00E-02 | | | 6,00E-02 | | | 7,00E-01 | | |
| | `subspace_iter_v2` | 2,00E-01 | 1,20E-01 | 1,00E-01 | 1,00E-02 | 3,00E-02 | 5,00E-02 | 2,00E-02 | 2,00E-02 | 6,00E-02 | 1,90E-01 | 1,40E-01 | 9,00E-02 |

On peut remarquer que la méthode `subspace_iteration_v2` est plus efficace que les méthodes `subspace_iteration_v0` et `subspace_iteration_v1` dans le cas de l'application aux matrices de type 1 et 4, et d'efficacité croissante avec p. Cela est du au bon conditionnement des ces deux types de matrice.
En revanche, cette efficacité a la tendance inverse pour les matrices de types 2 et 3

qui sont les matrices avec le moins bon conditionnement. Dans ce cas, les opérations *matrice* $\times$ *matrice* sont les plus coûteuses en temps.

**Question 11 :**

En comparant les précisions des vecteurs propres estimés pour `subspace_iter_v1`, on remarque que la qualité des vecteurs propres pour v1 est variable car les premiers vecteurs propres calculés sont affinés à chaque itération. Ainsi, la norme des résidus est croissante.

**Question 12 :**

Dans `subspace_iter_v3` on n'orthonormalise que la partie qui n'a pas convergé du vecteur V. Par conséquent, les résidus sont autour de la valeur de tolérence `eps`.

**Question 13 :**

**Voir `subspace_iter_v3.m` en annexe**

# Chapitre 4

# Expériences numériques

**Question 14 :**

Les matrices ont des conditionnements différents qui peuvent être classés dans cet ordre : $\kappa_2 > \kappa_3 > \kappa_4 > \kappa_1$ ; le type de matrice 1 est donc le mieux conditionné. Afin de visualiser la distribution des valeurs propres selon le type de matrice, on trace les figures ci-dessous :

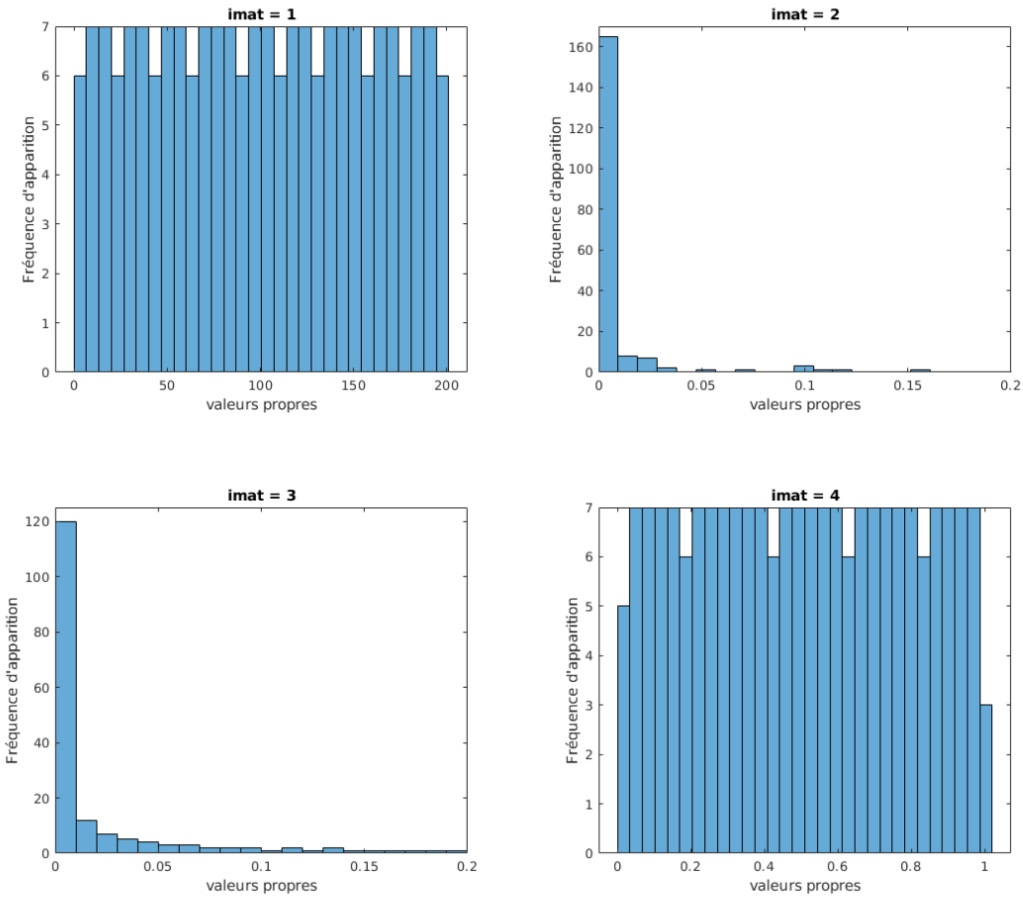



FIGURE 4.1 – Distribution des valeurs propres des matrices selon leur type

On remarque que les valeurs propres des matrices de type 2 et 3 sont rassemblées autour de 0, rendant difficile la détermination de leurs valeurs et vecteurs propres.

# Deuxième partie

# Séance 2

# Chapitre 1

# Compression d'image

**Question 1 :**

On a $U_k \in \mathbb{R}^{q\times k}$, $V_k \in \mathbb{R}^{k\times k}$ et $\Sigma_k \in \mathbb{R}^{p\times k}$

**Question 2 :**

La valeur de $\|I - I_k\|$ est très proche pour chacun des algorithmes :

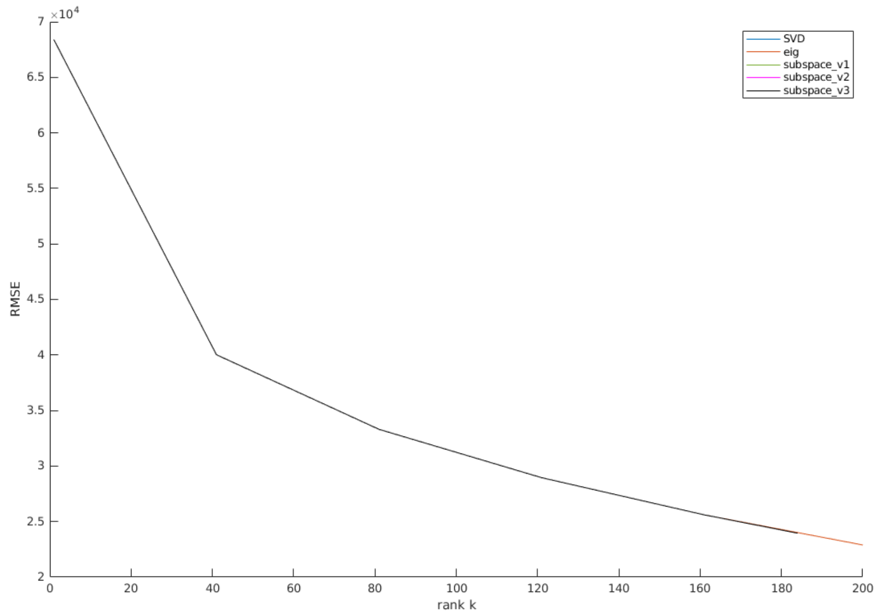



FIGURE 1.1 – $\|I - I_k\|$ en fonction de k

On peut maintenant comparer les temps d'exécution de reconstruction d'image pour chacun des algorithmes :

Table 1.1 – Temps d'exécution de chacun des algorithmes

| | | | | |
|---|---|---|---|---|
| Propriétés de la matrice | Valeur de k | 121 | | |
| | Taille | 1187 $\times$1920 | | |
| | Puissance | 1 | 2 | 4 |
| Algorithme | `eig` | 1,338E-01 | | |
| | `powerv11` | Temps d'exécution trop long | | |
| | `powerv12` | Temps d'exécution trop long | | |
| | `subspace_iter_v0` | Temps d'exécution trop long | | |
| | `subspace_iter_v1` | 1,8695E+00 | | |
| | `subspace_iter_v2` | 1,9499E+00 | 1,1344E+00 | 2,3639E+00 |
| | `subspace_iter_v3` | 1,9228E+00 | 1,2365E+00 | 2,8037E+00 |

**Voir la reconstitution de l'image en annexe**

On remarque que `eig.m` reste la méthode performante. Cependant, on réussit à créer des algorithmes seulement dix fois moins rapide que cette dernière.

# Troisième partie

# Conclusion

# 1 Bilan

Ce projet nous a permis dans un premier temps de découvrir les différentes implantations d'algorithmes de décomposition spectrale de matrice. On a alors pu comparer leurs temps d'exécution, et constater que certains programmes étaient davantage adaptés pour certains types de matrices. On a enfin appliqué l'ensemble de ces algorithmes pour la compression d'image, on a également comparé les précisions et les temps d'exécution de ces algorithmes.

# 2 Remerciements

Merci à l'ensemble des enseignants pour leurs aides précieuses apportées durant les séances de projet.

# Annexe

# Annexe A

# Programmes Matlab

Listing A.1 – Programme d'obtention du sous-espace propre dominant (v0)

```
% version basique de la méthode de l'espace invariant (v0)

% Données
% A           : matrice dont on cherche des couples propres
% m           : nombre de couples propres que l'on veut calculer
% eps         : seuil pour déterminer si les vecteurs de l'espace invariant
%               ont convergé
% maxit       : nombre maximum d'itérations de la méthode

% Résultats
% V : matrice des vecteurs propres
% D : matrice diagonale contenant les valeurs propres (ordre décroissant)
% it : nombre d'itérations de la méthode
% flag : indicateur sur la terminaison de l'algorithme
%  flag = 0  : on a convergé (on a calculé m valeurs propores)
%  flag = -3 : on n'a pas convergé en maxit itérations

function [ V, D, it, flag ] = subspace_iter_v0( A, m, eps, maxit )

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

    n = size(A,1);

    % indicateur de la convergence
    conv = 0;
    % numéro de l'itération courante
    k = 0;

    % on génère un ensemble initial de m vecteurs orthogonaux
    Q = rand(n , m);
    V = mgs(Q);

    % 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^T.A.V
        H = V'*Y;

        % vérification de la convergence
        compute_acc = norm(Y - V*H, 'fro')/normA;
        conv = compute_acc < eps;
```

```

        % orthonormalisation
        V = mgs(Y);

    end

    % décomposition spectrale de H, le quotient de Rayleigh
    [X, W] = eig(H);

    % on range les valeurs propres dans l'ordre décroissant
    [W, I] = sort(W, 'descend', 'ComparisonMethod','abs');

    % on permute les vecteurs propres en conséquence
    X =  I*X;

    % les m vecteurs propres dominants de A sont calculés à partir de ceux de H
    V = V*X;

    D = diag(W);

    it = k;

    if (conv)
      flag = 0;
    else
      flag = -3;
    end

end
```

Listing A.2 – Programme d'obtention du sous-espace propre dominant (v1)

```
% version améliorée de la méthode de l'espace invariant (v1)
% avec utilisation de la projection de Raleigh-Ritz

% Données
% A          : matrice dont on cherche des couples propres
% m          : taille maximale de l'espace invariant que l'on va utiliser
% percentage : pourcentage recherché de la trace
% eps        : seuil pour déterminer si un vecteur de l'espace invariant a convergé
% maxit      : nombre maximum d'itérations de la méthode

% Résultats
% V : matrice des vecteurs propres
% D : matrice diagonale contenant les valeurs propres (ordre décroissant)
% n_ev : nombre de couples propres calculées
% it : nombre d'itérations de la méthode
% itv  : nombre d'itérations pour chaque couple propre
% flag : indicateur sur la terminaison de l'algorithme
    %  flag = 0  : on converge en ayant atteint le pourcentage de la trace recherché
    %  flag = 1  : on converge en ayant atteint la taille maximale de l'espace
    %  flag = -3 : on n'a pas convergé en maxit itérations

function [ V, D, 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;
        %% 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)
            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
            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);
        D = diag(W);
        it = k;
    else
        % on n'a pas convergé
        D = 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
```

Listing A.3 – Programme d'obtention du sous-espace propre dominant (v3)

```
% version améliorée de la méthode de l'espace invariant (v3)
% avec utilisation de la projection de Raleigh-Ritz
% avec une accélération bloc
% avec utilisation de mgs_block

% Données
% A          : matrice dont on cherche des couples propres
% m          : taille maximale de l'espace invariant que l'on va utiliser
% percentage : pourcentage recherché de la trace
% p          : nombre de produits Y = A^p . V
% eps        : seuil pour déterminer si un vecteur de l'espace invariant a convergé
% maxit      : nombre maximum d'itérations de la méthode

% Résultats
% V : matrice des vecteurs propres
% D : matrice diagonale contenant les valeurs propres (ordre décroissant)
% n_ev : nombre de couples propres calculées
% it : nombre d'itérations de la méthode
% itv  : nombre d'itérations pour chaque couple propre
% flag : indicateur sur la terminaison de l'algorithme
```

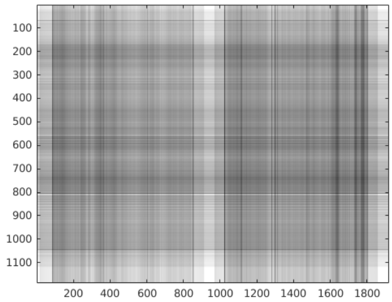
```matlab
    %  flag = 0  : on converge en ayant atteint le pourcentage de la trace recherché
    %  flag = 1  : on converge en ayant atteint la taille maximale de l'espace
    %  flag = -3 : on n'a pas convergé en maxit itérations

function [ V, D, n_ev, it, itv, flag ] = subspace_iter_v3( A, m, percentage, p, 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);
    Wr = zeros(1,m);

    % 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^p*V
        Temp = Vr(:, nb_c + 1:end);
        for k=1:p
            Temp = A*Temp;
        end

        Y = [Vr(:,1:nb_c) Temp];
        %% orthogonalisation
        Vr = mgs_block(Y, nb_c);

        %% Projection de Rayleigh-Ritz
        [Wr(:, nb_c + 1:end), Vr(:, nb_c + 1:end)] = rayleigh_ritz_projection(A, Vr(:, nb_c + 1:end));

        %% 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;
                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
            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);
        D = diag(W);
        it = k;
    else
        % on n'a pas convergé
        D = 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
```

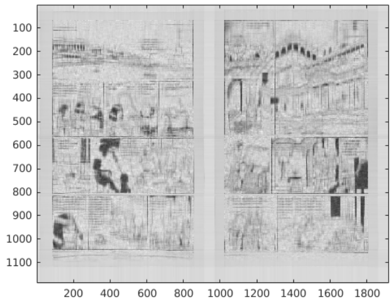
```
    end
end
```

# Annexe B

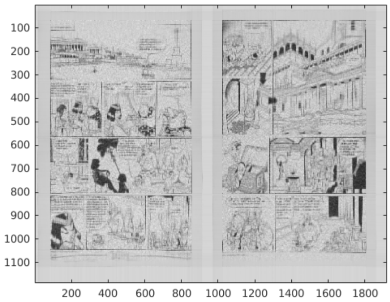
# Reconstitution d'image



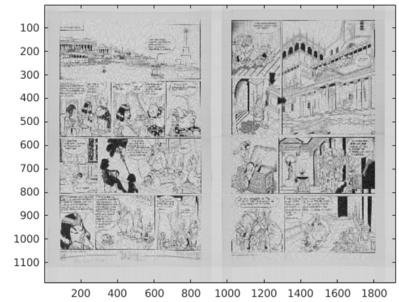

[1]



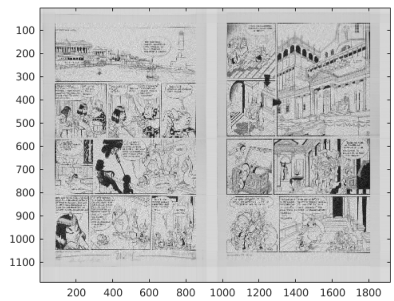

[41]



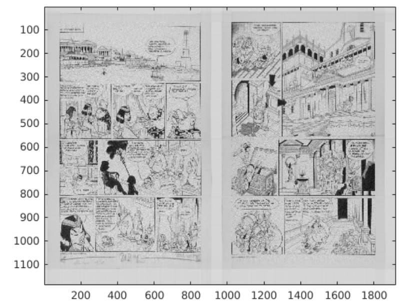

[81]




[121]




[161]




[201]

FIGURE B.1 – Image compressée pour différentes valeurs de k

# Bibliographie

[1] Equipe pédagogique - Calcul Scientifique (2022), *Subspace Iteration Methods*, ENSEEIHT.