# Rapport calcul scientifique

**MARTIN Nolann** **GUTIERREZ Tom**

Projet de première année de SN

# Table des matières

# Introduction

Dans ce projet, il est question d'implémenter et tester différents algorithmes de calcul de couples propres de matrices, plus ou moins grandes, avec des valeurs propres plus ou moins réparti équitablement

# Partie I

## Limite de la méthode des puissances

### Comparaisons de la méthode des puissance et de eig

Tableau comparant le temps pour des matrices de tailles différentes, sachant que nous utilisons ici un type de matrice où la valeur propre D(i) = i:

| | 10 | 50 | 100 | 200 | 500 | 1000 |
|---|---|---|---|---|---|---|
| fonction eig (s) | 0 | 0 | $3 * 10^{-2}$ | $3 * 10^{-2}$ | $7 * 10^{-2}$ | $8,6 * 10^{-1}$ |
| power v11 (s) | 0 | $1 * 10^{-2}$ | $5 * 10^{-2}$ | 1,5 | $3,6 * 10^{1}$ | Non Convergence |

Tableau comparant le temps pour des matrices de tailles identiques de matrices $(200 * 200)$ mais avec différents types de matrices. Nous pouvons trouver les différents types de matrices dans le script « matgen_csad ». Nous utiliserons la valeur de « imat » pour annoter quel type nous utilisons ici :

| | imat = 1 | imat = 2 | imat = 3 | imat = 4 |
|---|---|---|---|---|
| fonction eig (s) | $3 * 10^{-2}$ | $2 * 10^{-2}$ | $1 * 10^{-2}$ | $1 * 10^{-2}$ |
| power v11 (s) | 1,5 | $3 * 10^{-2}$ | $4 * 10^{-2}$ | 1,83 |

On peut voir, de ces deux tableaux, que la fonction eig de matlab reste meilleure que la fonction power V11. On voit tout de même que le temps reste similaire pour des matrices de petites tailles d'après le tableau 1, et que d'après le tableau 2 la méthode imat 2 donne des résultats similaire pour les deux fonctions.

Néanmoins, nous pouvons quand même voir que la fonction eig de matlab reste la plus performante en terme de temps contre le script power v11.

### Algorithme power_v12

```
Input: Matrix A ∈ Rn×n
Output: (λ1, v1) eigenpair associated to the largest (in module) eigenvalue.
v ∈ Rn given*
z = A · v
β = vT · z
repeat
v = z/ ‖z‖
βold = β
z = A · v
β = vT · z
until |β − βold| / |βold| < ε
λ1 = β and v1 = v
```

### Comparaisons de la méthode des puissance amélioré et de eig

Nous allons comparé de la même manière que précédemment et commencer par des matrices de tailles différente en utilisant un seul type de matrice :

| | 10 | 50 | 100 | 200 | 500 | 1000 |
|---|---|---|---|---|---|---|

| | | | | | | |
|---|---|---|---|---|---|---|
| fonction eig (s) | 0 | 0 | $3 * 10^{-2}$ | $3 * 10^{-2}$ | $7 * 10^{-2}$ | $8,6 * 10^{-1}$ |
| power v12 (s) | 0 | $1 * 10^{-2}$ | $4 * 10^{-2}$ | 1 | $1,9 * 10^{1}$ | Non Convergence |

Et nous allons dans un deuxième temps comparer sur les différentes types de matrices en prenant des matrices de $(200 * 200)$. Nous continuerons à utiliser les annotations imat :

| | imat = 1 | imat = 2 | imat = 3 | imat = 4 |
|---|---|---|---|---|
| fonction eig (s) | $3 * 10^{-2}$ | $2 * 10^{-2}$ | $1 * 10^{-2}$ | $1 * 10^{-2}$ |
| power v12 (s) | 1 | $2 * 10^{-2}$ | $3 * 10^{-2}$ | $8,3 * 10^{-1}$ |

Nous voyons donc bien que le script power v12 est plus rapide d'un facteur 2 et est donc beaucoup plus rapide, se rapprochant de la fonction eig.

**Inconvégnants de la métode des puissances**

Le principal désavantage de cette méthode est qu'elle nécessite beaucoup d'itérations pour converger et donc se révèle très coûteuse en termes de temps et de puissance de calcul.

On peut notamment relever que le fait de choisir le vecteur propre initial au hasard peut mener à un grand temps de calcul s'il est très éloigné de notre vecteur des valeurs propres recherchées, ainsi que le fait qu'il y est une déflation qui soit fait après le calcul d'un vecteur propre, ce qui rajoute encore des itérations au calcul.

## Extension de la méthode de la puissance

**Subspace_iter_v0**

Avec cette méthode, si nous prenons une base initial de m vecteurs, nous convergerons vers une matrice V des vecteurs propres de ces m vecteurs initiaux. Dont les m vecteurs propres en colonnes différent de la matrice A qui est symétrique.

Nous pouvons aussi voir que en prenant la variante de cette méthode incluant H, qu'il n'y a pas de problème sur le fait de faire toute la décomposition spectrale de H, car c'est une matrice de dimension m*m, qui est largement plus petit que A qui est une matrice de dimension n*n. Ce qui fait que ça n'est pas un problème, cela ne rajoute pas beaucoup de temps de calcul par rapport à ce que prennent les autres étapes.

**subspace_iter_v1**

**Algorithm 2** Subspace iteration method v1 with Raleigh-Ritz projection

**Input**: Symmetric matrix $A \in \mathbb{R}^{n \times n}$, tolerance $\varepsilon$, $MaxIter$ (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 $\Lambda_{out}$.

Generate an initial set of $m$ orthonormal vectors $V \in \mathbb{R}^{n \times m}$; ← line 47

$k = 0$; ← line 38

$PercentReached = 0$ ← line 45

**repeat**

$k = k + 1$←line 54

Compute $Y$such that $Y = A \cdot V$←line 56

$V \leftarrow$ orthonormalisation of the columns of $Y$←line 58

*Rayleigh-Ritz projection* applied on matrix $A$ and orthonormal vectors $V$←line 61

*Convergence analysis step* : save eigenpairs that have converged and update $PercentReached$ ←line 70

**Until** ($PercentReached > PercentTrace$ or $n_{\text{ev}} = m$ or k > $MaxIter$ ) ← line 52

## Vers une méthode efficace

**Subspace_iter_v2**

Nous allons par la suite améliorer l'algorithme en y ajoutant le fait que nous allons calculer $A^p * V$. Ce qui peut nous demander un certains nombre de calcul dont voici le coût :

pour $A^p$ :

- Nous avons A qui est une matrice symétrique de taille n*n et P un entier.
- $A^p$ va donc nous prendre p multiplications de A par elle même, nous faisons donc p multiplications matricielles, nous avons ici une complicité en p

pour $A^p * V$ :

- Nous savons déjà que la complexité de calcul est en p pour $A^p$, nous faisons donc simplement une multiplication matricielle de plus, nous rajoutons donc un de complexité, nous avons donc (p+1) nombre de calcul.

Nous voyons que quand nous augmentons p, nous avons d'abord une diminution du temps de calcul car nous économisons du temps par ces calcul de $A^p$, mais à partir d'un certain p, nous pouvons voir que le temps de calcul commence à devenir plus grand car les multiplications successives de A prennent trop de temps et allonge le temps de calcul plus qu'elles le diminue.

**Subspace_iter_v3**

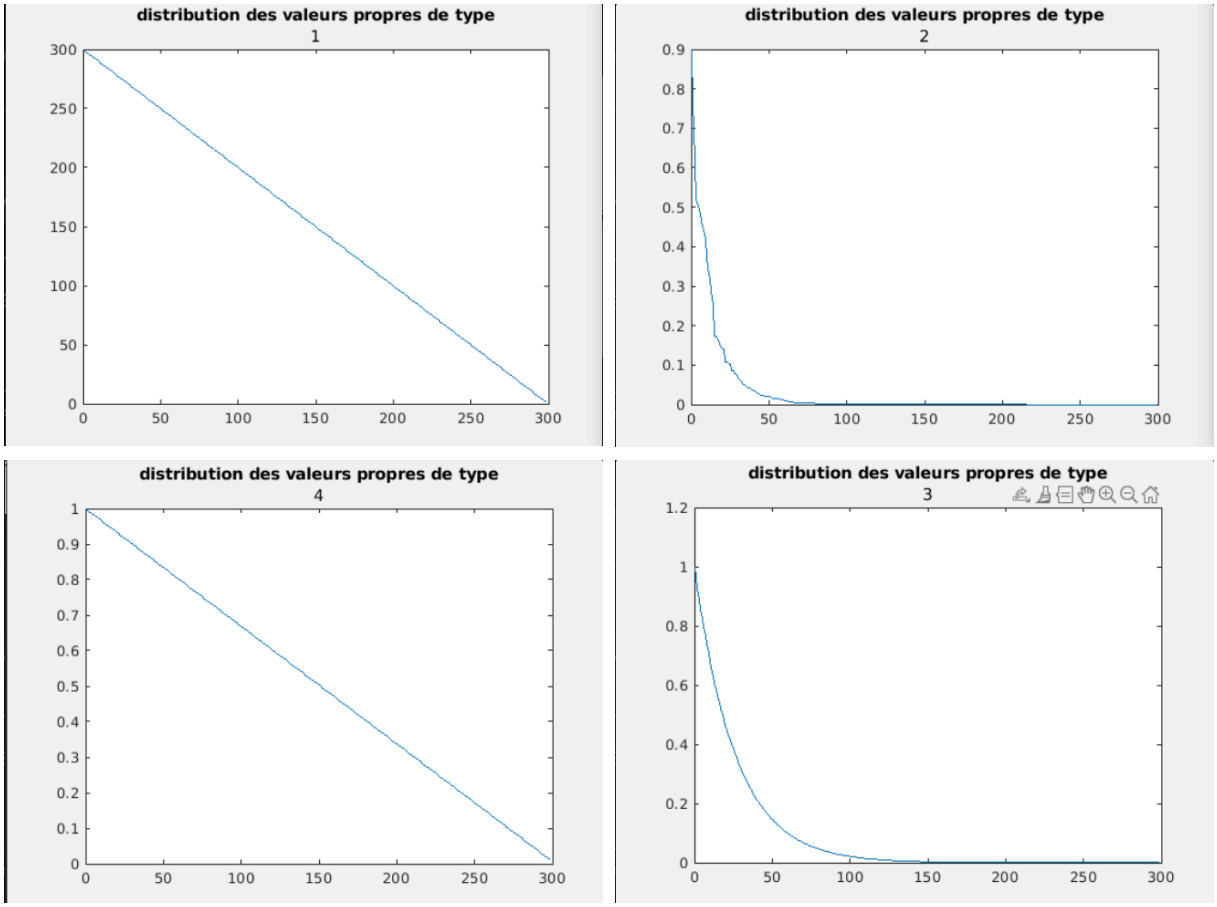
Nous pouvons voir dans le Subspace_iter_v1 et 2 que la qualité des paires dépend de plusieurs facteur influencés par :

- Si les matrices ont des vecteurs propres initiaux différent cela peut entraîner l'algorithme a converger vers une valeur qui n'est pas la bonne.
- Les sous espaces peuvent évoluer au fur et a mesure des itérations en fonction de la matrice de départ.

Pour essayer d'anticiper ce qu'il se passera avec ce nouvel algorithme, nous pouvons voir que le critère de convergence est atteint quand le critère de convergence est atteint, ce qui veut dire que le résidu est inférieur à une valeur epsilon multiplié par la norme de A.

## Analyses numériques

Nous pouvons voir que les matrices sont formés différemment en fonction du imat choisi, nous pouvons d'ailleurs voir la répartition ci dessous des valeurs propres dans chacun des imat :




Nous voyons donc bien que la répartition des valeurs propre est vraiment différente en fonction de la façon de concevoir la matrice, ce qui peut impacter nos algorithme, si nous avons un algorithme qui as besoin de valeurs propres espacés de manière homogène, certaines types de matrices vont donner plus de temps et de moins bonnes valeurs avec cet algorithme là.

Nous pouvons à présent comparer les différentes valeurs de temps que nous donne les algorithme dans ce tableau :

| taille | fonction | type 1 | type 2 | type 3 | type 4 |
|---|---|---|---|---|---|
| 100 | eig | 0.01 | 0.01 | 0 | 0.01 |
| | subspace_iter_v0 | 2.9 | 0.06 | 0.39 | 2.8 |
| | subspace_iter_v1 | 0.05 | 0.01 | 0.01 | 0.06 |
| | subspace_iter_v2 | 0.01 | 0 | 0 | 0.05 |
| | subspace_iter_v3 | 0.001 | 0 | 0 | 0.01 |
| | power_v11 | 0.07 | 0.04 | 0 | 0.08 |
| | power_v12 | 0.01 | 0.01 | 0 | 0.05 |

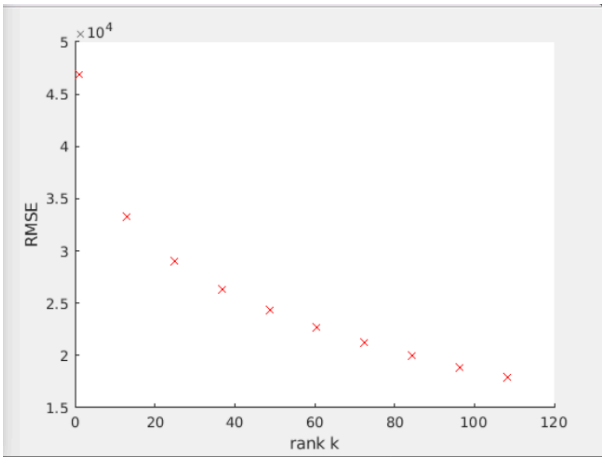
# Partie II

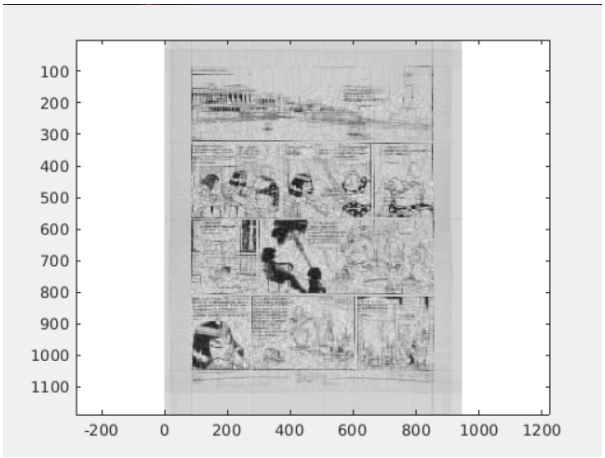
Dans cette partie nous nous intéressons à la reconstitution d'image a partir des algorithmes crées dans la partie I. Nous allons pour ce faire nous utilisons le triplet $(\Sigma_k, U_k, V_k)$ dont nous pouvons donner les dimensions :

- $\Sigma_k$ est une matrice constituée de q colonnes et p lignes, mais lorsqu'on fait une approximation au rang k, nous pouvons considérer qu'elle est de rang $k * k$ car les k premières valeurs singulières sont conservés
- $U_k$ est une matrice qui représente les vecteurs singuliers gauche, elle est donc de dimension $q * k$
- $V_k$ est une matrice qui représente les vecteurs singuliers droits, elle est donc de dimension $k * p$

Nous pouvons voir, en changeant les paramètres que pour une valeur de p trop grande avec les algorithmes subspace_iter_v2 ou subspace_iter_v3, l'image deviendra complètement noir en sortie car les valeurs propres calculées par matlab sont trop petite. Nous pouvons essayer différents paramètres en utilisant subspace_iteration_v2:

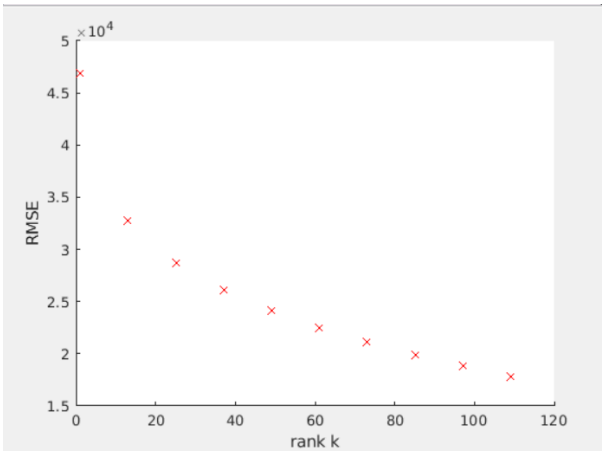
| | tolérance | max_it | taille de l'espace recherché | pourcentage fixé | p (pour les versions 2 et 3) |
|---|---|---|---|---|---|
| subspace_iter_v3 | $1 * 10^{-8}$ | 10000 | 400 | 0.995 | 3 |

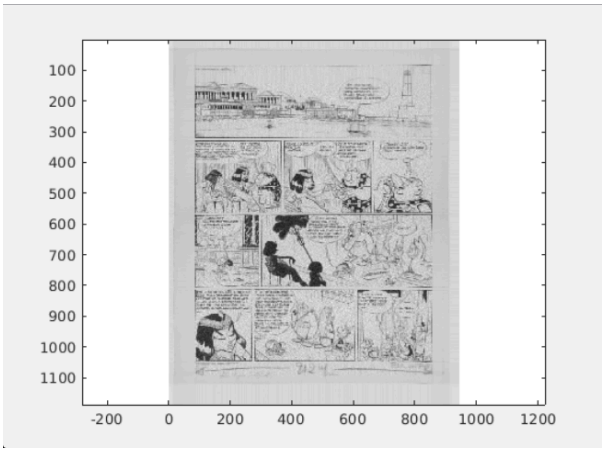


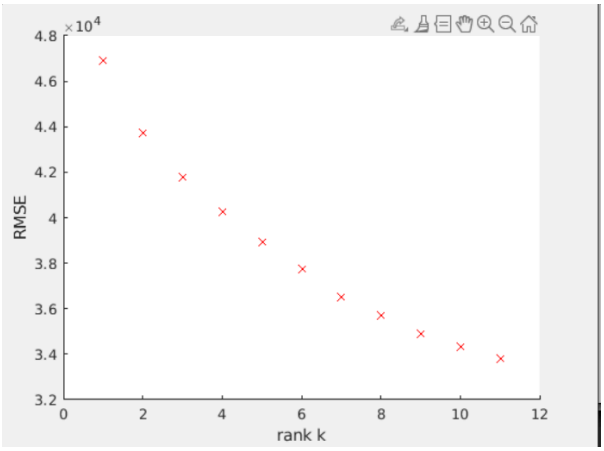

avec Subspace_iter_v3 :

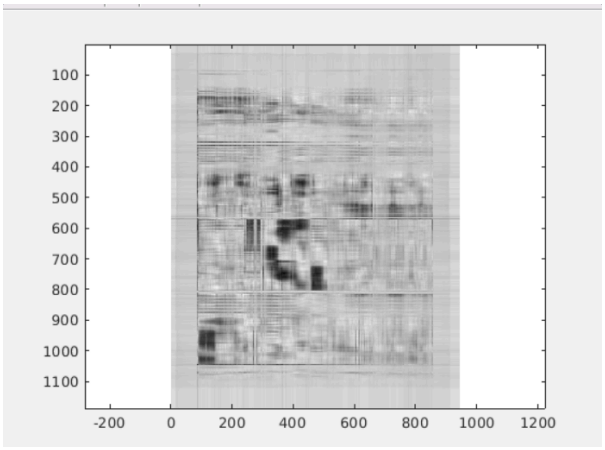
| | tolérance | max_it | taille de l'espace recherché | pourcentage fixé | p (pour les versions 2 et 3) |
|---|---|---|---|---|---|
| subspace_iter_v3 | $1 * 10^{-8}$ | 10000 | 400 | 0.995 | 3 |



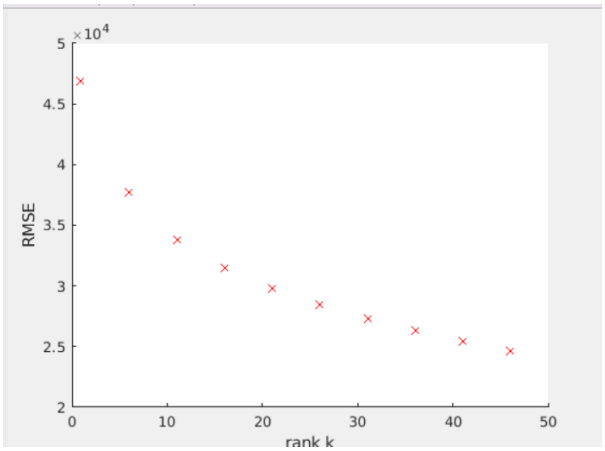



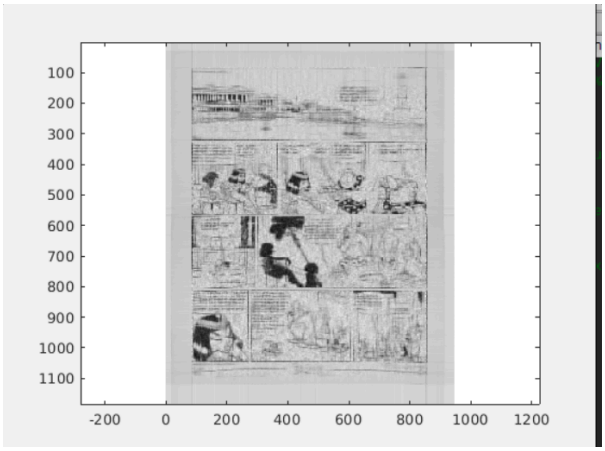
| | tolérance | max_it | taille de l'espace recherché | pourcentage fixé | p (pour les versions 2 et 3) |
|---|---|---|---|---|---|
| subspace_iter_v3 | $1 * 10^{-8}$ | 10000 | 400 | 0.980 | 3 |



| | tolérance | max_it | taille de l'espace recherché | pourcentage fixé | p (pour les versions 2 et 3) |
|---|---|---|---|---|---|
| subspace_iter_v3 | $1 * 10^{-8}$ | 10000 | 400 | 0.990 | 3 |



Nous faisons varier surtout le pourcentage car c'est le seul qui a un impact réel sur le résultat.

# Conclusion

Pour conclure, le calcul de valeurs propres peut être très différent en fonction de la méthode utilisée, avoir un temps et une qualité de calcul très disparate, hors comme nous l'avons vu en pratique, cela change beaucoup par exemple la reconstituions d'images. Voilà pourquoi il est important de toujours réfléchir a de nouvelles façons d'améliorer nos algorithmes, afin de fournir des qualités toujours meilleurs et un temps plus rapide.