# SVD et ses applications

**Khalil Al Sayed et Zahira Tahiri**
Ingénierie ISN (Science des données),
Professeur référent M.Bernhard Beckermann,
3 juin 2021


MDR1 MEMOIRE DE RECHERCHE
(rapport écrit)

Année 2020-2021

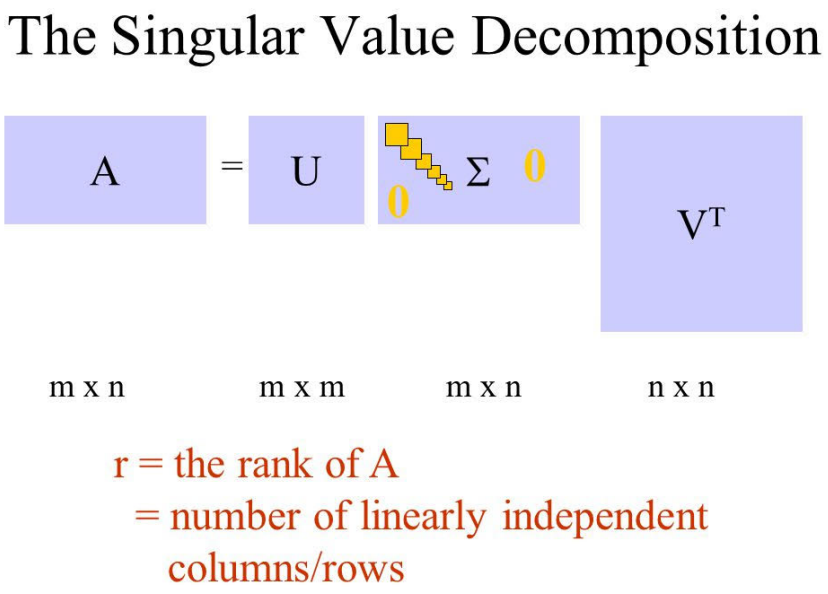
# Remerciements

Nous tenons à remercier les enseignants Bernhard Beckermann et Caterina Calgaro du laboratoire Paul Painlevé de l'Université de Lille de leur soutien qui nous ont apporté, ainsi leur contribution à l'élaboration de ce projet de recherche . Ils étaient toujours présents à satisfaire nos demandes et répondre à nos questions .

# Table des matières

# 1 Introduction

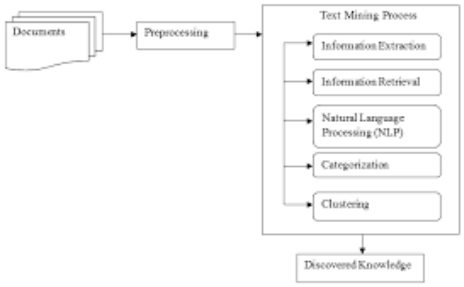
SVD : Singular Value Decomposition est une résolution d'une gamme de problèmes comme par exemple des moindres carrés bien ou mal conditionnés ou des problèmes sous-contraints et bien d'autres... La décomposition en valeur singulière est un outil indispensable, les justifications mathématiques sont riches mais nous le présenterons dans l'optique d'utilisation pour des classes de problème ciblée.
Soit $A$ , une matrice $m \times n$, la décomposition SVD est un algorithme de factorisation qui permet d'exprimer A comme le produit de trois matrices particulières, $U$, $V$ et $\Sigma$ telles que $A = U\Sigma V^T$
$U$ est une matrice $m \times m$, orthogonale
$\Sigma$ est une matrice $m \times n$, diagonale positive
$V^T$ est une matrice $n \times n$ orthogonale



# 2 Application Text Mining

## 2.1 Definition

Le text mining regroupe l'ensemble des techniques de data management et de data mining permettant le traitement des données particulières que sont les données textuelles. Par données textuelles, on entend par exemple les corpus de textes, les réponses aux questions ouvertes d'un questionnaire, les champs texte d'une application métier où des conseillers clientèle saisissent en temps réel les informations que leur donnent les clients, les mails, les posts sur les réseaux sociaux, les articles, les rapports...
Un des aspects centraux du text mining est de transformer ces données textuelles peu structurées – si ce n'est par la langue utilisée – en données exploitables par les algorithmes classiques de data mining. Il s'agit tout simplement de transformer un texte brut en tableau de données indispensable aux analystes chargés d'en dégager du sens. Il s'agit ensuite de déployer les méthodes statistiques les plus à même de répondre à une problématique donnée.




## 2.2 Les techniques du NLP

**Traitement du corpus** Cela commence par des opérations élémentaires de normalisation du texte qui réduisent le nombre de formes, mais peuvent aussi détruire de l'information. Le package classique dans l'environnement R est 'tm', les packages 'tidytext' ou 'quanteda' proposent eux des grammaires simplifiées. Les principales opérations sont les suivantes :
-Mettre le texte en minuscule, mais au risque de perdre les capitales initiales qui indicent les noms communs.
-Éliminer les nombres et dates.
-Repérer (et éliminer) les liens URL et les mentions de personnes.
-Éliminer aussi les mots sans signification que l'anglais dénomme par "stopwords" avec des dictionnaires

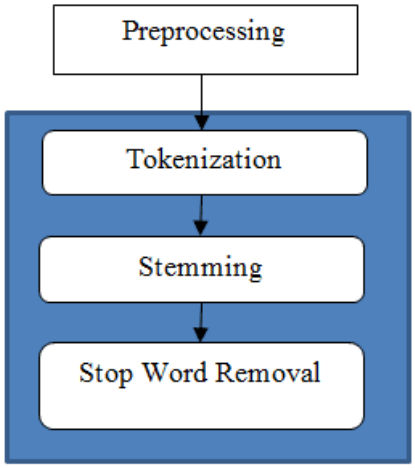
-Repérer et traiter les émoticônes. Le langage numérique a la particularité de réintroduire des éléments iconiques dans une écriture culturellement alphabétique.
-Dans le cas d'un langage vernaculaire la correction orthographique est indispensable. Le package 'hunspell' permet de repérer les fautes.

**Tokenisation** Cette première phase de lissage, de correction, de nettoyage, des données textuelles laisse la place à une seconde série d'opérations qui vise à définir l'objet de l'analyse : c'est la tokenisation des documents qui consiste à identifier les unités de textes élémentaires qui peuvent être des mots, mais aussi des lettres, des syllabes, des phrases, ou des séquences de ces éléments.
Chaque document devient alors une liste ordonnée (ou non) de termes élémentaires : les tokens. Nous passons d'un plan de données qui liste des documents (et les associe éventuellement à des auteurs), à un plan qui associe un document à une série d'attributs qui sont ses éléments unitaires (lettres, mots, syllabes, phrases).

**Stemming** «En linguistique, la racinisation ou désuffixation (anglais : stemming) est un procédé de transformation des flexions en leur radical ou racine (anglais : stem).» Il existe 2 méthodes : Soit on utilise un dictionnaire (énorme) soit un algorithme. Les algorithmes diffèrent entre les langues. "Porter", les plus populaire pour l'Anglais ne marche pas très bien en français, et il tente de travailler avec "Carry" ou encore un algorithme adaptable comme Paice/Husk.

**Lemmatisation** «La lemmatisation désigne l'analyse lexicale du contenu d'un texte regroupant les mots d'une même famille. Chacun des mots d'un contenu se trouve ainsi réduit en une entité appelée lemme (forme canonique). La lemmatisation regroupe les différentes formes que peut revêtir un mot, soit le nom, le pluriel, le verbe à l'infinitif, etc.»

**Enlever d'autres mots** Lorsqu'on analyse un domaine précis (par ex. des fiches sur le "jeu"), on peut enlever des termes comme "jeu" qui se retrouvent dans chaque document.




### 2.3 L'analyse sémantique latente LSA

Le point de départ d'une analyse sémantique latente consiste en un tableau lexical qui contient le nombre d'occurrences de chaque mot dans chacun des documents, un document pouvant être un texte, un paragraphe ou même une phrase. Pour dériver d'un tableau lexical les relations sémantiques entre les mots, la simple analyse des cooccurrences brutes se heurte à un problème majeur. Même dans un grand corpus de textes, la plus grande partie des mots sont relativement rares. Il s'ensuit que les cooccurrences le sont encore plus. Leur rareté les rend particulièrement sensibles à des variations aléatoires.
L'LSA résout ce problème en remplaçant le tableau de fréquences original par une approximation qui produit une sorte de lissage des associations. Pour cela, le tableau de fréquences fait l'objet d'une décomposition en valeurs singulières avant d'être recomposé à partir d'une fraction seulement de l'information qu'il contient. Les milliers de mots caractérisant les documents sont ainsi remplacés par des combinaisons linéaires ou 'dimensions sémantiques' sur lesquelles peuvent être situés les mots originaux. Contrairement à une analyse factorielle classique, les dimensions extraites sont très nombreuses (plusieurs centaines) et non interprétables. Elles peuvent toutefois être vues comme analogues aux traits sémantiques fréquemment postulés pour décrire le sens des mots .

### 2.4 Exemple (Romeo et Juliette )

Supposons que nous ayons l'ensemble de cinq documents suivant :
d1 : Romeo and Juliette.
d2 : Juliet : O happy dagger !
d3 : Romeo died by dagger.
d4 : "Live free or die", that's the New-Hampshire's motto.
d5 : Did you know, New-Hampshire is in New-England.

Tout d'abord, on va faire les techniques du NLP : on met le texte en minuscules, on élimine les ponctuations, on applique la Tokenisation, le Stemming et la Lemmatisation. Ensuite, on va faire une requête de recherche sur les deux mots : "die", "dagger".
clairement, d3 devrait être classé en haut de la liste du fait qu'il contient les deux mots : "die", "dagger".
Puis, d2 et d4 devrait suivre parce qu'ils contiennent un mot chacun de la requête.
Cependant, qu'en est-il de d1 et d5 ?
Naturellement (comme humain), sachant que d1 est plus proche de notre requête de recherche que d5, alors nous voudrons que d1 soit classé plus haut que d5. De ce fait la question qui se pose est : la machine peut-elle déduire cela ? la réponse sera alors oui étant donné que LSA est capable de faire la relation entre le terme "dagger" et les termes de d1 : "romeo" et "juliett" qui se trouvaient notamment dans d2 et d3.
Même chose pour le terme "die", ce dernier est relié à d1 et d5 étant donné que le terme "romeo" est associé au terme "die" dans d3 et que le terme "new-hampshire" est associé au terme "die" dans d4.
Par conséquent, LSA va donner un meilleur classement à d1 "romeo", "juliette" que d5 "new-hampshire" vu que d1 a une double connexion avec le terme "dagger" dans d2 et d3, et qu'il est aussi connecté au terme "die" par "romeo" dans d3. Contrairement à d5 qui a une seule connexion avec un seul terme "die" dans d4.

#### 2.4.1 SVD pour LSA

Soit $A$ la matrice terme-document $m \times n$ d'une collection de documents. Chaque colonne de A correspond à un document. Si le terme $i$ apparaît a fois dans le document $j$, donc $A[i, j] = a$ . La dimension de $A$ est $m \times n$ , m et n correspondent alors au nombre de mots et de documents, respectivement. Dans notre exemple on écrit :

| | $d_1$ | $d_2$ | $d_3$ | $d_4$ | $d_5$ |
|---|---|---|---|---|---|
| *romeo* | 1 | 0 | 1 | 0 | 0 |
| *juliet* | 1 | 1 | 0 | 0 | 0 |
| *happy* | 0 | 1 | 0 | 0 | 0 |
| *dagger* | 0 | 1 | 1 | 0 | 0 |
| *live* | 0 | 0 | 0 | 1 | 0 |
| *die* | 0 | 0 | 1 | 1 | 0 |
| *free* | 0 | 0 | 0 | 1 | 0 |
| *new-hampshire* | 0 | 0 | 0 | 1 | 1 |

Soit $B = A^T A$ est la matrice document-document. Si les documents $i$ et $j$ ont $b$ mots en commun donc $B[i, j] = b$. Soit $C = AA^T$ est la matrice termes-termes. Si le terme i et j apparaissent tous les deux dans le document $c$ donc $C[i, j] = c$. Clairement, $B$ et $C$ sont des matrices carrés et symétriques. $B$ est une matrice $m \times m$ et $C$ est une matrice $n \times n$.
On applique la SVD dans A ( avec les programmes C++) on utilisant les matrices $B$ et $C$. Soit $A = S\Sigma U^T$ tel que $S$ est une matrice de vecteur propre de $B$. $U$ est une matrice de vecteur propre de $C$ et $\Sigma$ est la matrice diagonale des valeurs singulières obtenues sous forme de racines carrées des valeurs propres de $B$.

La matrice $\Sigma$ dans notre exemple :

$$\Sigma = \begin{bmatrix} 2.285 & 0 & 0 & 0 & 0 \\ 0 & 2.010 & 0 & 0 & 0 \\ 0 & 0 & 1.361 & 0 & 0 \\ 0 & 0 & 0 & 1.118 & 0 \\ 0 & 0 & 0 & 0 & 0.797 \end{bmatrix}$$

les valeurs singulières le long de la diagonale de $\Sigma$ sont listées par ordre décroissant de leur grandeur.
Certaines des valeurs singulières sont «trop petites» et donc «négligeables». Dans LSA, nous ignorons ces petites valeurs singulières et les remplaçons par 0.
Supposons qu'on garde $k$ valeurs singulières dans $\Sigma$ . Alors on aura que des zéros dans $\Sigma$ sauf les premières $k$ entrées le long de sa diagonale. De ce fait on réduit la matrice $\Sigma$ en matrice $\Sigma_k$ qui est une matrice $k \times k$ on réduit aussi $S$ et $U_T$ en $S_k$ et $U_k^T$ pour avoir $k$ colonnes et $k$ lignes
La matrice $A$ est maintenant approximée par

$$A_k = S_k \Sigma_k U_k^T. \tag{1}$$

Puisque $S_k$, $\Sigma$ et $U_k^T$ sont des matrices $m \times k$, $k \times k$ et $k \times n$, respectivement . Donc leur produit sera la matrice $A_k$ de dimension $m \times n$.
Intuitivement, les $k$ éléments restants des vecteurs propres en $S$ et $U$ correspondent à $k$ «concepts cachés» auxquels participent les termes et les documents. Les termes et les documents ont maintenant une nouvelle représentation en termes de ces concepts cachés.
À savoir, les termes sont représentés par les vecteurs lignes de la matrice $S_k\Sigma_k$ de dimension $m \times k$ et les documents sont représentés par la matrice $\Sigma_k U_k^T$ de dimension $k \times n$ .
Ensuite, la requête est représentée par le barycentre des vecteurs pour ses termes.

On se place maintenant dans notre exemple, dans le cas de $k = 2$ , donc on considère que les deux premières valeurs singulières de $\Sigma_2$.

$$\Sigma_2 = \begin{bmatrix} 2.285 & 0 \\ 0 & 2.010 \end{bmatrix}$$

$$S_2 = \begin{array}{r} \textit{romeo} \\ \textit{juliet} \\ \textit{happy} \\ \textit{dagger} \\ \textit{live} \\ \textit{die} \\ \textit{free} \\ \textit{new-hampshire} \end{array} \begin{bmatrix} -0.396 & 0.280 \\ -0.314 & 0.450 \\ -0.178 & 0.269 \\ -0.438 & 0.369 \\ -0.264 & -0.346 \\ -0.524 & -0.246 \\ -0.264 & -0.346 \\ -0.326 & -0.460 \end{bmatrix}$$

$$U_2^T = \begin{bmatrix} -0.311 & -0.407 & -0.594 & -0.603 & -0.143 \\ 0.363 & 0.541 & 0.200 & -0.695 & -0.229 \end{bmatrix}$$

Les termes du «concept space» sont représentés par les vecteurs lignes de $S_2$ tandis que les documents par les vecteurs colonnes de $U_2^T$.
En effet, nous redimensionnons les deux coordonnées de ces vecteurs en multipliant par les valeurs singulières correspondantes de $\Sigma_2$ et représentons ainsi les termes par les vecteurs lignes de $S_2\Sigma_2$ et les documents par les vecteurs colonnes de $\Sigma_2 U_2^T$ .

$$romeo = \begin{bmatrix} -0.905 \\ 0.563 \end{bmatrix},\ juliet = \begin{bmatrix} -0.717 \\ 0.905 \end{bmatrix},\ happy = \begin{bmatrix} -0.407 \\ 0.541 \end{bmatrix},\ dagger = \begin{bmatrix} -1.001 \\ 0.742 \end{bmatrix},$$

$$live = \begin{bmatrix} -0.603 \\ -0.695 \end{bmatrix},\ die = \begin{bmatrix} -1.197 \\ -0.494 \end{bmatrix},\ free = \begin{bmatrix} -0.603 \\ -0.695 \end{bmatrix},\ new\text{-}hampshire = \begin{bmatrix} -0.745 \\ -0.925 \end{bmatrix},$$

et

$$d_1 = \begin{bmatrix} -0.711 \\ 0.730 \end{bmatrix},\ d_2 = \begin{bmatrix} -0.930 \\ 1.087 \end{bmatrix},\ d_3 = \begin{bmatrix} -1.357 \\ 0.402 \end{bmatrix},\ d_4 = \begin{bmatrix} -1.378 \\ -1.397 \end{bmatrix},\ d_5 = \begin{bmatrix} -0.327 \\ -0.460 \end{bmatrix}.$$

On calcule les vecteurs de la requête de la même manière que l'on calcul les vecteurs du barycentre. Dans notre exemple la requête est "die" et "dagger" donc le vecteur est :

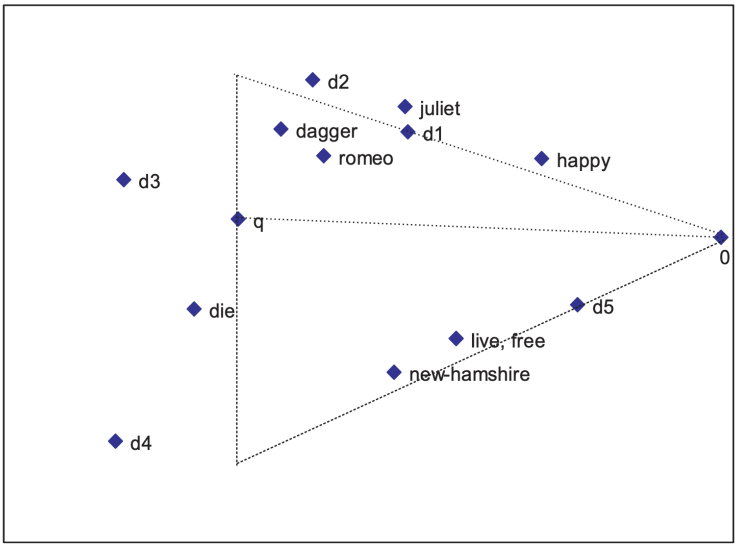
$$q = \frac{\begin{bmatrix} -1.197 \\ -0.494 \end{bmatrix} + \begin{bmatrix} -1.001 \\ 0.742 \end{bmatrix}}{2} = \begin{bmatrix} -1.099 \\ 0.124 \end{bmatrix}.$$

Afin de classer les documents par rapport à la requête q, nous utiliserons la distance cosinus : $\frac{d_i q}{|d_i||q|}$ pour $i \in [1, n]$ par ordre décroissant.

### 2.4.2 Interprétation

On observe d'après la figure ci-dessous :
Le document d1 est plus proche de la requête q que d5, par conséquent d1 est mieux classé que d5. Ce qui est conforme à notre préférence humaine.

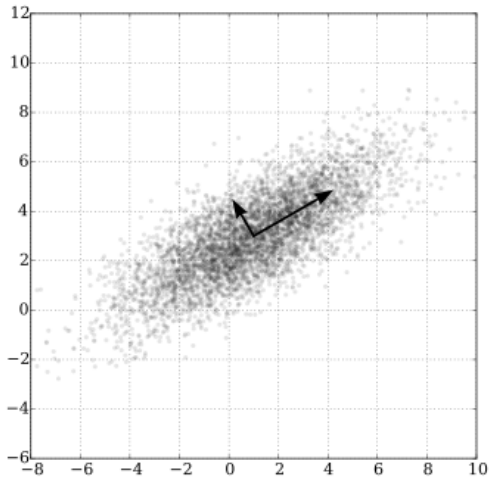
Le document d1 est un peu plus proche de q que d2, Ainsi, le système est assez intelligent pour découvrir que d1, contenant à la fois Roméo et Juliette, est plus pertinent pour la requête que d2 même si d2 contient l'un des mots de la requête alors que ce n'est pas le cas pour d1. Un utilisateur humain ferait probablement de même.




# 3 Application ACP

En pratique, le machine learning a de bonnes chances d'échouer quand il s'agit d'analyser des données de trés grandes dimensions comme de texte(pdf), du son(mp3) ou des images(jpg), on parle de la malédiction de la dimensionalité. pour lutter contre la malédiction de la dimensionnalité il est utile de chercher a résumer la complexité des données brutes en une représentation de moindre dimension, pour cela on utilise l'ACP.

L'analyse en composantes principales (ACP), est une méthode de la famille de l'analyse des données et plus généralement de la statistique multivariée, qui consiste à transformer des variables liées entre elles (dites (corrélées) en statistique) en nouvelles variables décorrélées les unes des autres. Ces nouvelles variables sont nommées ( composantes principales ), ou axes principaux. Elle permet au praticien de réduire le nombre de variables et de rendre l'information moins redondante.




Il s'agit donc d'une approche à la fois géométrique (les variables étant représentées dans un nouvel espace, selon des directions d'inertie maximale) et statistique (la recherche portant sur des axes indépendants expliquant au mieux la variabilité — la variance des données).

### 3.1 ACP et décomposition en valeurs singulières

### 3.2 Axe principale (cas de deux variables)

Soit $X = (X_1, X_2)$ le tableau des données brutes (base des données) à deux variables :

| Individu | variable1 | variable2 |
|---|---|---|
| 1 | $x_1$ | $y_1$ |
| 2 | $x_2$ | $y_2$ |
| . | . | . |
| . | . | . |
| . | . | . |
| n | $x_n$ | $y_n$ |

On se place dans le repère de centre $(mean(X_1), mean(X_2))$. Ceci revient à considérer les variables centrées.

L'axe principale a pour direction

$$PC_1 = argmax\{var(Xu^T) : u \in \mathbb{R}^2, ||u||^2 = 1\} \tag{2}$$

Parmi toutes les projections $Xu^T$ sur l'axe de direction $u$, celle qui a la plus grande variance est $XPC_1^T$.

### 3.3 Proposition

Le vecteur colonne $PC_1$ est un vecteur propre de la matrice de covariance $C = X^T X/n$ de $(X, Y)$. La plus grande valeur propre

$$\lambda_1 = max\{var(Xu^T) : u \in \mathbb{R}^2, ||u||^2 = 1\} \tag{3}$$

Les projections des individus

$$f_{i1} = \langle PC_1, (x_i, y_i)\rangle, i = 1, ..., n, \tag{4}$$

forment la colonne

$$f_1 = XPC_1{}^T, \tag{5}$$

appelée le premier facteur. Sa variance

$$var(f_1) = \lambda_1. \tag{6}$$

**Preuve** Admis (optimisation et lagrangien)
Le deuxième axe principal

$$PC_2 = argmax\{var(Xu^T) : u \in \mathbb{R}^2, u \perp PC_1, ||u||^2 = 1\} \tag{7}$$

La deuxième valeur propre de $C$

$$\lambda_2 = max\{var(Xu^T) : u \in \mathbb{R}^2, u \perp PC_1, ||u||^2 = 1\} \tag{8}$$

Les projections des individus

$$f_{i2} = \langle PC_2, (x_i, y_i)\rangle, i = 1, ..., n, \tag{9}$$

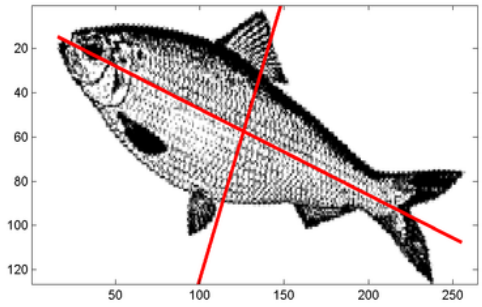
forment la colonne

$$f_2 = XPC_2{}^T, \tag{10}$$

Sa variance

$$var(f_2) = \lambda_2. \tag{11}$$

**Remarque** Exprimer dans la base canonique de $\mathbb{R}^2$ les variables sont corrélées. Exprimer dans la nouvelle base $(PC_1, PC_2)$ les variables deviennent non corrélées.




### 3.4 Axe principale (cas de trois variables)


### 3.5 Lien entre SVD et les valeurs propres de la matrice de covariance

$$\Sigma^T\Sigma = (U^TAV)^TU^TAV = V^TA^TUU^TAV = V^TA^TAV \iff A^TAV = V\Sigma^T\Sigma \implies \left[A^TAv_1, ..., A^TAv_p\right] = \left[{\sigma_1}^2v_1, ..., {\sigma_p}^2v_p\right]$$
$$\text{donc } A^TAv_i = {\sigma_i}^2v_i \ \forall i = 1, ...p$$

De meme

$$\Sigma\Sigma^T = U^TAV(U^TAV)^T = U^TAA^TU \iff AA^TU = \Sigma\Sigma^TU \implies \left[AA^Tu_1, ..., AA^Tu_p\right] = \left[{\sigma_1}^2u_1, ..., {\sigma_p}^2u_p\right]$$
$$\text{donc } AA^Tu_i = {\sigma_i}^2u_i \ \forall i = 1, ...p$$

Donc les ${\sigma_i}^2$ sont les valeurs propres de $A^TA$ et $AA^T$, et les colonnes de $V$ sont les vecteurs propres de $A^TA$ donc de $C = A^TA/n$, et on à la matrice $F = (f_1, f_2, ..., f_p) = AV$, donc à partir de $SVD$ on trouve tous les composantes principales et les facteurs principales pour la base des données $A$.

### 3.6 Application : analyse textuelle d'un corpus d'emails

Dans cette partie, on analyse un jeu de données textuelles dans le but de tenter de déterminer les caractéristiques de spams. Une telle analyse est classiquement basée sur la fréquence d'une sélection de mots dans un ensemble d'apprentissage constitué de courriels qui appartiennent à 2 catégories possibles : spam ou non-spam. Les données analysées dans cet exemple se trouvent dans le file spam.csv.

Il a été constitué un échantillon de 4601 messages électroniques dans chacun desquels a été évalué le nombre d'occurrences d'une sélection de mots et caractères. Les variables considérées sont des ratios qui correspondent au nombre d'occurrences d'un mot spécifique sur le nombre total de mots, ou nombre d'occurrences d'un caractère sur le nombre de caractères du message. Il a egalement considéré trois variables prenant en compte la casse (majuscule / minuscule) des caractères et une dernière variable qualitative binaire indiquant le type de chaque message : spam ou non spam.

**output: pdf_document**

soit le matrice spam :

```
##   word_freq_make word_freq_address word_freq_all word_freq_3d word_freq_our
## 1           0.00              0.64          0.64            0          0.32
## 2           0.21              0.28          0.50            0          0.14
## 3           0.06              0.00          0.71            0          1.23
## 4           0.00              0.00          0.00            0          0.63
##   word_freq_over word_freq_remove word_freq_internet word_freq_order
## 1           0.00             0.00               0.00            0.00
## 2           0.28             0.21               0.07            0.00
## 3           0.19             0.19               0.12            0.64
## 4           0.00             0.31               0.63            0.31
##   word_freq_mail word_freq_receive word_freq_will word_freq_people
## 1           0.00              0.00           0.64             0.00
## 2           0.94              0.21           0.79             0.65
## 3           0.25              0.38           0.45             0.12
## 4           0.63              0.31           0.31             0.31
##   word_freq_report word_freq_addresses word_freq_free word_freq_business
## 1             0.00                0.00           0.32               0.00
## 2             0.21                0.14           0.14               0.07
## 3             0.00                1.75           0.06               0.06
## 4             0.00                0.00           0.31               0.00
##   word_freq_email word_freq_you word_freq_credit word_freq_your word_freq_font
## 1            1.29           1.9              0.00           0.96              0
## 2            0.28           3.5              0.00           1.59              0
## 3            1.03           1.4              0.32           0.51              0
## 4            0.00           3.2              0.00           0.31              0
##   word_freq_000 word_freq_money word_freq_hp word_freq_hpl word_freq_george
## 1          0.00            0.00            0             0               0
## 2          0.43            0.43            0             0               0
## 3          1.16            0.06            0             0               0
## 4          0.00            0.00            0             0               0
##   word_freq_650 word_freq_lab word_freq_labs word_freq_telnet word_freq_857
## 1             0             0              0                0             0
## 2             0             0              0                0             0
## 3             0             0              0                0             0
## 4             0             0              0                0             0
##   word_freq_data word_freq_415 word_freq_85 word_freq_technology word_freq_1999
## 1              0             0            0                    0          0.00
## 2              0             0            0                    0          0.07
## 3              0             0            0                    0          0.00
## 4              0             0            0                    0          0.00
##   word_freq_parts word_freq_pm word_freq_direct word_freq_cs word_freq_meeting
## 1               0            0             0.00            0                 0
## 2               0            0             0.00            0                 0
## 3               0            0             0.06            0                 0
## 4               0            0             0.00            0                 0
##   word_freq_original word_freq_project word_freq_re word_freq_edu
## 1               0.00                 0         0.00          0.00
```

```
## [2,]             -0.13        -0.30         -0.20          -0.071
## [3,]             -0.13        -0.24         -0.13          -0.071
## [4,]             -0.13        -0.30         -0.20          -0.071
##      word_freq_conference char_freq_; char_freq_( char_freq_[ char_freq_!
## [1,]                -0.11       -0.16     -0.5143       -0.16       0.6239
## [2,]                -0.11       -0.16     -0.0260       -0.16       0.1262
## [3,]                -0.11       -0.12      0.0147       -0.16       0.0085
## [4,]                -0.11       -0.16     -0.0075       -0.16      -0.1619
##      char_freq_$ char_freq_# capital_run_length_average
## [1,]       -0.31     -0.1030                    -0.0452
## [2,]        0.42      0.0088                    -0.0024
## [3,]        0.44     -0.0797                     0.1459
## [4,]       -0.31     -0.1030                    -0.0521
##      capital_run_length_longest capital_run_length_total   y
## [1,]                      0.045                  -0.0087 1.2
## [2,]                      0.251                   1.2282 1.2
## [3,]                      2.221                   3.2584 1.2
## [4,]                     -0.062                  -0.1522 1.2
```

...jusqu'a la ligne 4601

Pour appliquer l'ACP il faut que cette matrice soit centrée reduit, alors soit $B$ le version centrée reduit de spam :

```
##      word_freq_make word_freq_address word_freq_all word_freq_3d word_freq_our
## [1,]          -0.34             0.331          0.71       -0.047         0.012
## [2,]           0.35             0.052          0.44       -0.047        -0.256
## [3,]          -0.15            -0.165          0.85       -0.047         1.365
## [4,]          -0.34            -0.165         -0.56       -0.047         0.473
##      word_freq_over word_freq_remove word_freq_internet word_freq_order
## [1,]          -0.35            -0.29             -0.263           -0.32
## [2,]           0.67             0.24             -0.088           -0.32
## [3,]           0.34             0.19              0.037            1.97
## [4,]          -0.35             0.50              1.308            0.79
##      word_freq_mail word_freq_receive word_freq_will word_freq_people
## [1,]         -0.371            -0.30           0.11           -0.312
## [2,]          1.087             0.75           0.29            1.847
## [3,]          0.016             1.59          -0.11            0.087
## [4,]          0.606             1.24          -0.27            0.718
##      word_freq_report word_freq_addresses word_freq_free word_freq_business
## [1,]            -0.17               -0.19          0.086              -0.32
## [2,]             0.45                0.35         -0.132              -0.16
## [3,]            -0.17                6.57         -0.229              -0.19
## [4,]            -0.17               -0.19          0.074              -0.32
##      word_freq_email word_freq_you word_freq_credit word_freq_your
## [1,]            2.08          0.15            -0.17           0.13
## [2,]            0.18          1.02            -0.17           0.65
## [3,]            1.59         -0.17             0.46          -0.25
## [4,]           -0.35          0.85            -0.17          -0.42
```

...jusqu'a la ligne 4601

Maintenant il faut calculer le SVD de la matrice $B$, c'est à dire les matrices $V$, $U$ et $\Sigma$, mais puisqu'on veut representer les nuages des points dans un plan alors on a besoin seulement de $\sigma_1$,$\sigma_2$, $V_1$, $V_2$, $U_1$ et $U_2$, et pour tracer le graphe des variables il faut calculer $B^T \begin{bmatrix} U_1 & U_2 \end{bmatrix}$ et pour tracer le graphe des individus il faut calculer $B \begin{bmatrix} V_1 & V_2 \end{bmatrix}$, d'après l'algorithme de

Golub-Kahan et l'algorithme QR qu'on utilise dans le programme C++ on trouve $\sigma_1 = 175.577$ et $\sigma_2 = 130.391$ (regarde aa.txt) qui est identique avec les $\sigma_1$ et $\sigma_2$ trouver dans RStudio, la même chose pour $V_1$, $V_2$, $U_1$ et $U_2$ qui sont respectivement (regarde UU.txt et VV.txt) :

$V_1$ :

```
  [1] "0.0459207"    "0.00939708"   "0.0509435"    "0.00729108"   "0.0414265"
  [6] "0.0504251"    "0.052844"     "0.0382473"    "0.0510423"    "0.0243324"
 [11] "0.0556575"    "0.0218216"    "0.039732"     "0.0189633"    "0.0365575"
 [16] "0.0473511"    "0.0511672"    "0.0256393"    "0.0866817"    "0.0350694"
 [21] "0.0830122"    "0.0155651"    "0.0610149"    "0.0441894"    "-0.201779"
 [26] "-0.197807"    "-0.04196"     "-0.259234"    "-0.204575"    "-0.282425"
 [31] "-0.288317"    "-0.321128"    "-0.00922544"  "-0.320664"    "-0.24969"
 [36] "-0.293108"    "-0.0489814"   "0.00082914"   "-0.0383054"   "-0.293885"
 [41] "-0.011726"    "-0.0268016"   "-0.0645005"   "-0.0139264"   "-0.0129312"
 [46] "-0.00249552"  "-0.00394674"  "-0.00616606"  "-0.000128268" "-0.130605"
 [51] "-0.0193784"   "0.0488566"    "0.0587961"    "0.00108001"   "0.0197667"
 [56] "0.0361644"    "0.0495115"    "0.133008"
```

$V_2$ :

```
  [1] "-0.151365"   "0.0197945"   "-0.157819"   "-0.0167226"  "-0.130603"   "-0.165313"
  [7] "-0.162328"   "-0.134538"   "-0.218729"   "-0.143749"   "-0.208781"   "-0.0599642"
 [13] "-0.116316"   "-0.060956"   "-0.202802"   "-0.117428"   "-0.198599"   "-0.168022"
 [19] "-0.179087"   "-0.134287"   "-0.269566"   "-0.00573847" "-0.244609"   "-0.147542"
 [25] "0.0392505"   "0.0338714"   "0.115393"    "-0.0703323"  "-0.0539937"  "-0.0925554"
 [31] "-0.124527"   "-0.151646"   "0.0884754"   "-0.152541"   "-0.0801441"  "-0.109852"
 [37] "0.127311"    "0.0199976"   "0.0828364"   "-0.175706"   "0.0864966"   "0.0776327"
 [43] "0.0676724"   "0.059533"    "0.0955886"   "0.113587"    "0.0131877"   "0.0570422"
 [49] "0.0424082"   "-0.0651415"  "0.0494529"   "-0.128715"   "-0.233304"   "-0.0342036"
 [55] "-0.122933"   "-0.231998"   "-0.211047"   "-0.389276"
```

$U_1$ :

```
   [1] "0.0061692"    "0.00983414"   "0.0125054"    "0.00680102"   "0.00680102"
   [6] "0.0044603"    "0.00847796"   "0.00453631"   "0.0108"       "0.00702342"
  [11] "0.00301757"   "0.00543778"   "0.00591254"   "0.0086806"    "0.0105502"
  [16] "0.00374786"   "0.00586535"   "0.00704757"   "0.00682051"   "-0.000283315"
  [21] "0.000903692"  "0.00754578"   "0.00459702"   "0.00573693"   "0.00287854"
  [26] "0.00754578"   "0.00287854"   "0.008469"     "0.00343272"   "0.00954502"
  [31] "0.00709949"   "0.00627788"   "0.00738337"   "0.00699978"   "-0.00634515"
  [36] "0.00637866"   "0.00450579"   "0.00895652"   "0.00572385"   "0.00677834"
  [41] "0.0104833"    "0.00477414"   "0.00471643"   "0.00683587"   "0.0083054"
  [46] "0.0100726"    "0.00954014"   "0.0134705"    "0.0140168"    "0.00974328"
  [51] "0.00701255"   "0.0144329"    "0.00700051"   "0.00696052"   "0.01418"
  [56] "0.00573019"   "0.00716743"   "0.00792182"   "0.0062255"    "0.00699407"
```

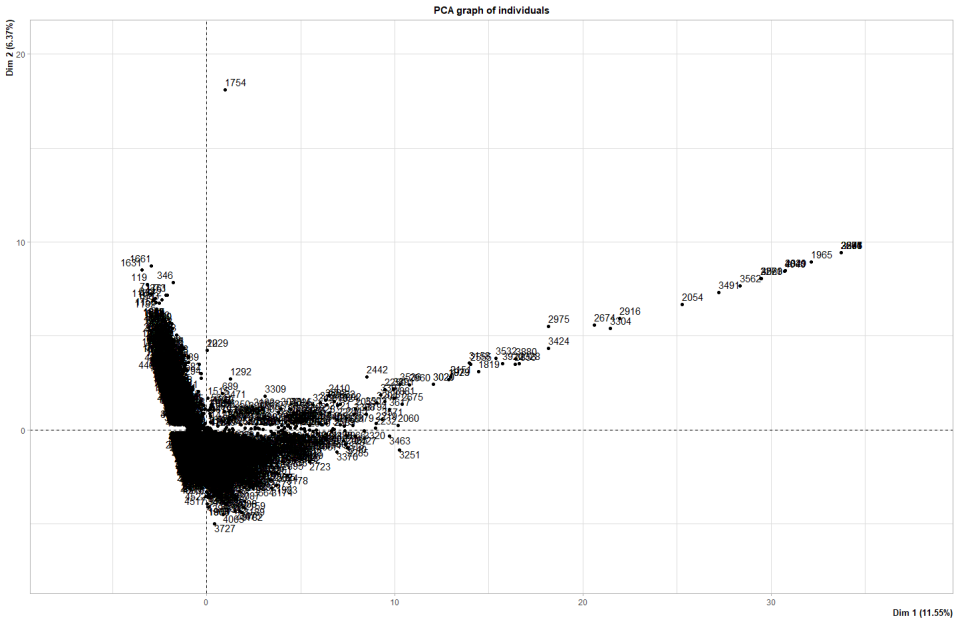
...jusqu'a la ligne 4601

$U_2$ :

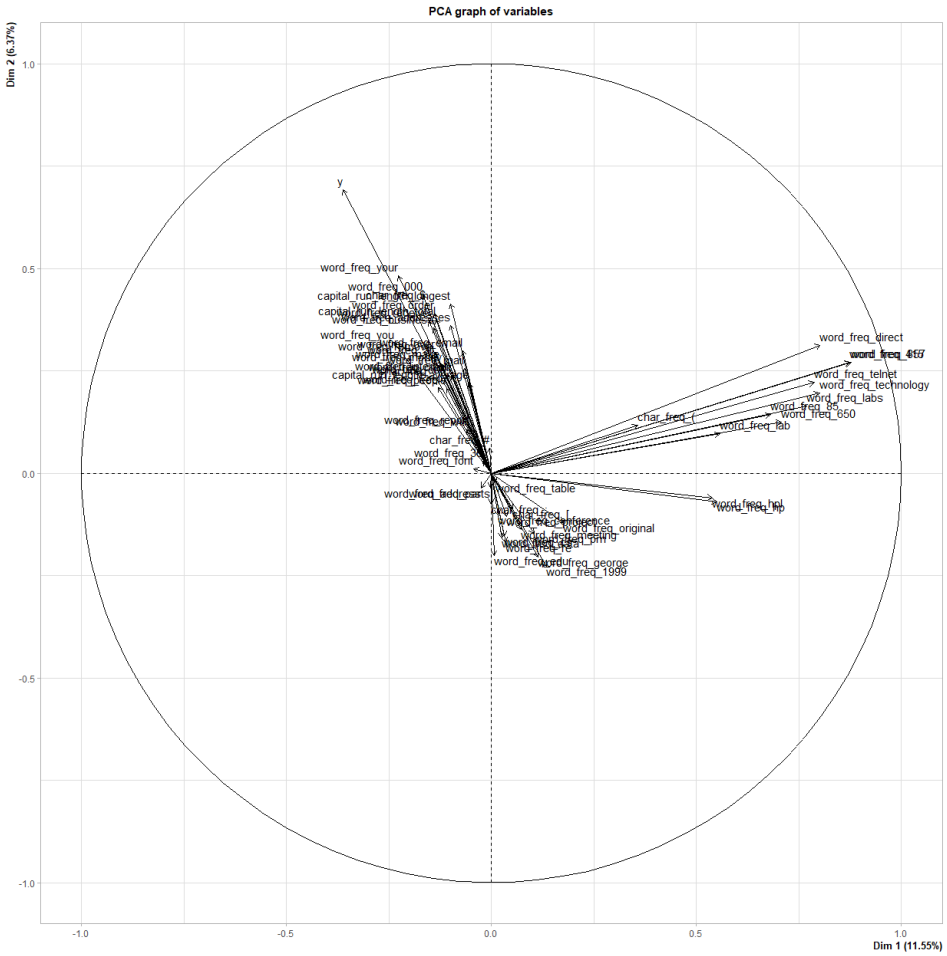
```
   [1] "-0.00304261"   "-0.0185636"    "-0.0401971"    "-0.00655279"   "-0.00655279"
   [6] "-5.59418e-005" "-0.0110768"    "-6.7706e-005"  "-0.0301926"    "-0.00620864"
  [11] "-0.0172359"    "1.35459e-005"  "-0.00295121"   "-0.0112443"    "-0.0168106"
  [16] "-0.00625596"   "-0.000694183"  "-0.00741828"   "-0.00649476"   "-0.0353433"
  [21] "0.00578563"    "-0.0109951"    "-0.000877815"  "-0.00313003"   "0.00646223"
  [26] "-0.0109951"    "0.00646223"    "-0.00894471"   "0.00323837"    "-0.0230807"
  [31] "-0.00413177"   "-0.00591848"   "-0.00576777"   "-0.0096354"    "-0.00832994"
  [36] "-0.00453258"   "-0.000939809"  "-0.0133728"    "-0.00225144"   "-0.00551525"
  [41] "-0.0183634"    "-0.000625092"  "-0.000688804"  "-0.00653592"   "-0.0126549"
  [46] "-0.0206752"    "-0.0151911"    "-0.0318869"    "-0.0355497"    "-0.0175442"
  [51] "-0.00381153"   "-0.0358985"    "-0.00611581"   "-0.00803215"   "-0.0350996"
  [56] "8.03604e-005"  "-0.0083194"    "-0.0071739"    "-0.00280795"   "-0.00624245"
```

...jusqu'a la ligne 4601

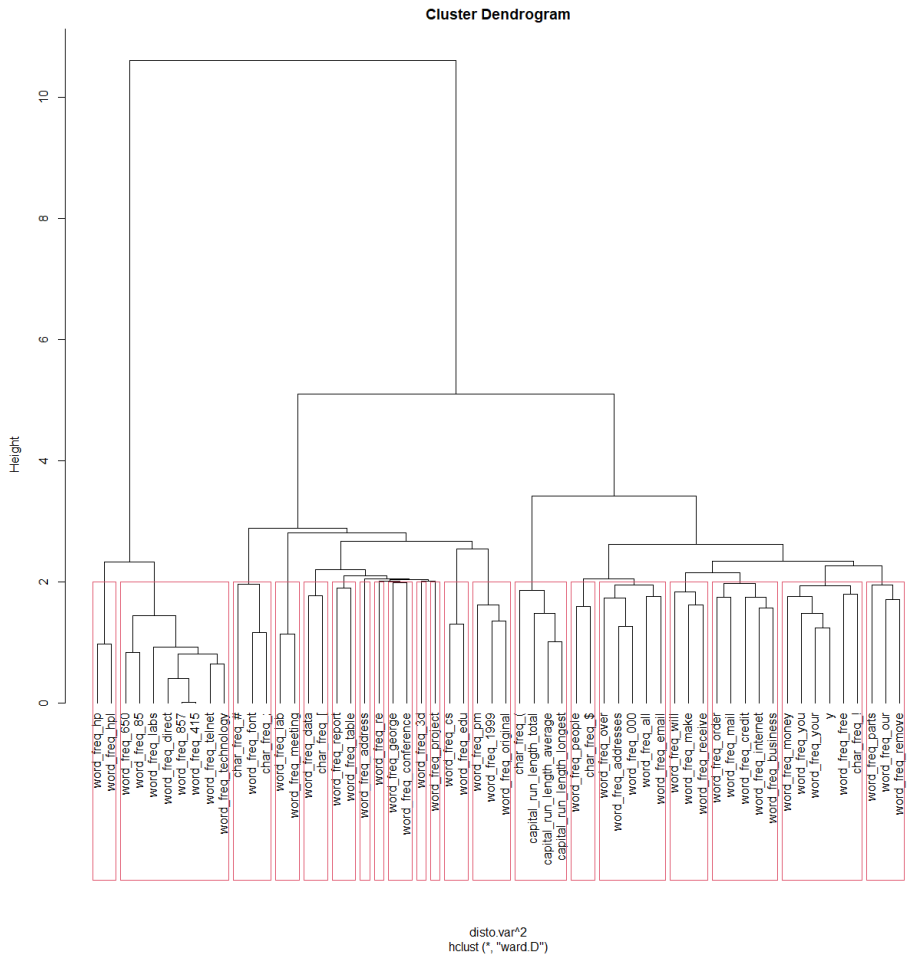
Maintenant après avoir calculé $\sigma_1, \sigma_2, V_1, V_2, U1 et U_2$ par l'algorithme Golub-kahan et QR, on peut tracer le graphe des individus :




et on peut tracer le graphe des variables :




d'aprés ce graphe du cercle des corrélations, on voit que beaucoup des variables (il comprend le vriable $Y$ de spam) sont corélées positivement à l'axe 1 et negativement à l'axe 2, donc on peut dire que tout les variables representé par une flèche proche de la flèche de $Y$ représentent les mots et les caractères dans un message de spam, pour mieux visualiser la classification des variables :




Maintenant on peut visualiser clairement d'après le rectangle 19 du Dendrogram que les variables (word-freq-money, word-freq-you, word-freq-your, word-freq-free, char-freq-!) sont dans la même classe, donc on peut deduire de ces 4601 messages électroniques que la plupart des spams messages contiennent beaucoup des mots (money, you, your, free) et beaucoup de caractères (!).

# 4 Approximation d'une matrice pour un rang fixe

Le procédé d'algèbre linéaire de décomposition en valeurs singulières (ou SVD, de l'anglais singular value decomposition) d'une matrice est un outil important de factorisation des matrices rectangulaires réelles ou complexes.

Le rôle important des valeurs singulières c'est que $\sigma_j$ la j-ieme valeur

singulière donne la distance de $A$ a l'ensemble des matrices rang au plus j-1 . Donc mesure la distance à l'ensemble des matrices à faible rang. Soit les matrices $A$ et $B$ de dimension $m \times n$

$$\sigma_j = min\{\|A - B\| : rang(B) \leq j - 1\}$$

On a $A$ la matrice par exemple une matrice de données on cherche de faire une compression de l'information . De ce fait approcher la matrice $A$ par une matrice $B$ plus simple et la simplicité on la mesure par le rang .

### 4.1 Théorème de Eckart Young avec la norme induite

Soit $A \in \mathbb{R}^{m\times n}$ $m \geq n$ et $k < r = rang(A)$ et $A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T$ donc

$$min\|A - B\|_2 = \|A - A_k\|_2 = \sigma_{k+1} \tag{12}$$

*Démonstration.* Par définition de $A_k$ on dispose de sa SVD On a

$$A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T$$

Soit $U^T A_k V = diag(\sigma_1, ..., \sigma_k, 0, .., 0)$ et $rang(A_k) = k$
et

$$U^T(A - A_k)V = diag(0, ..., 0, \sigma_{k+1}, ..., \sigma_p)$$

On obtient

$$\|A - A_k\|_2 = \sigma_{k+1}$$

On suppose que $rang(B) \leq k$ Dans ce cas, $\dim(\mathbb{K}er(B)) \geq n - k$ pour $1 \leq k \leq r$ et $B \in \mathbb{R}^{m\times n}$
En passant éventuellement à l'adjoint, on peut supposer sans perte de la généralité que $m \geq n$
Ceci implique qu'il existe $n - k$ vecteurs $x_k, ..., x_n \in \mathbb{K}er(B)$ deux a deux orthogonaux (et donc libres).

$$vect(x_1, ..., x_{n-k}) \cap vect(v_1, ..., v_{k+1}) \neq \{0\} \qquad (*)$$

Démontrons (*) $dim(x_1, ..., x_{n-k}) = n - k$ et $dim(v_1, ..., x_v k_{+1}) = k + 1$ ils ne sont pas libre
Donc

$$\exists \alpha_j, \beta_j \qquad \sum_{j=1}^{n-k} \alpha_j x_j + \sum_{j=1}^{k+1} \beta_j v_j = 0$$

Ce qui s'écrit

$$\exists \alpha_j, \beta_j \qquad \sum_{j=1}^{n-k} \alpha_j x_j = -\sum_{j=1}^{k+1} \beta_j v_j$$

Si on suppose $a = 0$ appartient a (*) , on va avoir une combinaison linéaire non trivial des $x_j$ pour $1 \leq j \leq n-k$ ou des $v_j$ pour $1 \leq j \leq k+1$ ou les deux ce qui est impossible, donc $a \neq 0$
Soit $z$ est dans l'intersection de (*), qui est un espace vectoriel, donc sans perte de la generalité $\|z\|_2 = 1$ .
Alors par définition de SVD on a :

$$Az = \sum_{i=1}^{p} \sigma_i (v_i^T z) u_i = \sum_{i=1}^{k+1} \sigma_i (v_i^T z) u_i + \sum_{i=k+2}^{p} \sigma_i (v_i^T z) u_i$$

Comme $z \in Vect(v_1, .., v_{k+1})$ et $V$ une matrice orthogonale, donc $v_i \perp z$ pour $k+2 \leq i \leq p$
On a donc

$$Az = \sum_{i=1}^{p} \sigma_i (v_i^T z) u_i = \sum_{i=1}^{k+1} \sigma_i (v_i^T z) u_i$$

On a $Bz = 0$ par construction $z \in \mathbb{K}er(B)$

$$\|A - B\|_2^2 \geq \|(A-B)z\|_2^2 = \|Az - Bz\|_2^2 = \|Az\|_2^2$$

Comme $U$ une matrice orthogonale on a :

$$\|Az\|_2^2 = \sum_{i=1}^{k+1} \sigma_i^2 (v_i^T z)^2$$

Puisque $\sigma_i \geq \sigma_{k+1}$ , on remplacant chaque $\sigma_i$ par $\sigma_{k+1} < \sigma_i$ donc :

$$\|Az\|_2^2 = \sum_{i=1}^{k+1} \sigma_i^2 (v_i^T z)^2 \geq \sigma_{k+1}^2 \|z\|_2^2$$

Comme $\|z\|_2^2 = 1$
Donc

$$\|Az\|_2^2 \geq \sigma_{k+1}^2$$

Donc

$$\|A - B\|_2^2 \geq \sigma_{k+1}^2$$

□

## 4.2 Théorème de Eckart Young avec la norme de Frobenius

### 4.2.1 Von Neumann's trace inequality (pour les matrices de taille $m \times n$)

Soit $A$ et $B$ deux matrices $m \times n$ avec des valeurs singulières $\alpha_1 \geq \alpha_2 \geq ... \geq \alpha_{min(m,n)}$ et $\beta_1 \geq \beta_2 \geq ... \geq \beta_{min(m,n)}$ respectivement, alors $|Tr(A^T B)| \leq \sum_{i=1}^{min(m,n)} \alpha_i \beta_i$.

*Démonstration.* Il faut utiliser d'abord le théorème de Von Neumann's trace inequality pour les matrices carrées (sa preuve est admise Parce que c'est trop long) qui dit : Soit $A$ et $B$ deux matrices $n \times n$ avec des valeurs singulières $\alpha_1 \geq \alpha_2 \geq ... \geq \alpha_n$ et $\beta_1 \geq \beta_2 \geq ... \geq \beta_n$ respectivement, alors $|Tr(A^T B)| \leq \sum_{i=1}^{n} \alpha_i \beta_i$.

si nous supposons que $n < m$, alors on peut transformer la matrice $A$ en une matrice carrée de taille $m \times m$ :

$$A = \begin{bmatrix} a_1 & a_2 & \cdots & \cdots & \cdots & a_n \\ \vdots & a_{n+2} & a_{n+3} & & & \vdots \\ \vdots & \ddots & a_{2n+3} & a_{2n+4} & & \vdots \\ \vdots & & & \ddots & \ddots & \vdots \\ \vdots & & & & a_{(m-1)n-1} & a_{(m-1)n} \\ \vdots & \cdots & \cdots & \cdots & \cdots & a_{mn} \end{bmatrix} \Longrightarrow$$

$$A' = \left[\begin{array}{ccccccc|c} a_1 & a_2 & \cdots & \cdots & \cdots & a_n & & \\ \vdots & a_{n+2} & a_{n+3} & & & \vdots & & \\ \vdots & \ddots & a_{2n+3} & a_{2n+4} & & \vdots & & \\ \vdots & & & \ddots & \ddots & \vdots & & 0 \\ \vdots & & & & a_{(m-1)n-1} & a_{(m-1)n} & & \\ \vdots & \cdots & \cdots & \cdots & \cdots & a_{mn} & & \end{array}\right]$$

et on fait la même chose avec $B$ qui devient $B'$, alors les valeurs singulières de $A'$ et de $B'$ deviennent respectivement $\alpha_1 \geq \alpha_2 \geq ... \geq \alpha_n \geq \alpha_{n+1} = 0 \geq ... \geq \alpha_m = 0$ et $\beta_1 \geq \beta_2 \geq ... \geq \beta_n \geq \beta_{n+1} = 0 \geq ... \geq \beta_m = 0$. et puisque $Tr(A^T B) = Tr(A'^T B')$ alors on utilisons Von Neumann's pour les matrices carrées $A'$ et $B'$ on peut conclure que :

$$|Tr(A^T B)| = |Tr(A'^T B')| \leq \sum_{i=1}^{m} \alpha_i \beta_i = \sum_{i=1}^{min(m,n)} \alpha_i \beta_i \tag{13}$$

□

### 4.2.2 Eckart Young avec la norme de Frobenius

Soit $k < r = rang(A)$ et $A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T$ donc :

$$\min_{rang(B) \leq k} \|A - B\|_F = \|A - A_k\|_F = \sqrt{(}\sigma_{k+1}^2 + \sigma_{k+2}^2 + ... + \sigma_p^2) \tag{14}$$

*Démonstration.* $\leq$: $U^T(A - A_k)V = diag(0, ..., 0, \sigma_{k+1}, ..., \sigma_p) \implies (A - A_k) = U diag(0, ..., 0, \sigma_{k+1}, ..., \sigma_p)V^T \underset{corollaire 2.4.3}{\implies} \|A - A_k\|_F = \sqrt{\sigma_{k+1}^2 + ... + \sigma_p^2} \implies \min_{rang(B) \leq k} \|A - B\|_F \leq \sqrt{\sigma_{k+1}^2 + ... + \sigma_p^2}$.

$\geq$: pour cette partie, il faut utiliser :
1- $\|A\|_F^2 = trace(A^T A)$
2- Von Neumann's trace inequality pour les matrices de taille $(m \times n)$ : Soit $A$ et $B$ deux matrices $m \times n$ avec des valeurs singuliéres $\alpha_1 \geq \alpha_2 \geq ... \geq \alpha_{min(m,n)}$ et $\beta_1 \geq \beta_2 \geq ... \geq \beta_{min(m,n)}$ respectivement, alors $|Tr(A^T B)| \leq$

$\sum_{i=1}^{min(m,n)} \alpha_i\beta_i$.

3- la norme de Frobenius est invariante par les matrices unitaires.

Soit $B$ matrice tel que $rang(B) = k' \leq k$ et a pour matrice diagonale de valeurs singulières $\Gamma$, on a

$$\|A-B\|_F^2 \underset{1}{=} Tr((A-B)^T(A-B)) = Tr(A^TA - A^TB - B^TA + B^TB) = Tr(A^TA) + Tr(B^TB) - Tr(B^TA) - Tr(A^TB) = \|A\|_F^2 + \|B\|_F^2 - Tr(B^TA) - Tr(A^TB) \underset{3}{=} \|\Sigma\|_F^2 + \|\Gamma\|_F^2 - Tr(B^TA) - Tr(A^TB) \underset{1}{=} Tr(\Sigma^T\Sigma) + Tr(\Gamma^T\Gamma) - Tr(B^TA) - Tr(A^TB) \geq Tr(\Sigma^T\Sigma) + Tr(\Gamma^T\Gamma) - |Tr(B^TA)| - |Tr(A^TB)|$$

et d'aprés 2 on a $|Tr(B^TA)| \leq \sum_{i=1}^{min(m,n)} \alpha_i\beta_i = \sum_{i=1}^{k'} \alpha_i\beta_i = \Gamma^T\Sigma^{k'}$ et de même pour $Tr(A^TB)$ alors

$$Tr(\Sigma^T\Sigma) + Tr(\Gamma^T\Gamma) - |Tr(B^TA)| - |Tr(A^TB)| \geq Tr(\Sigma^T\Sigma) + Tr(\Gamma^T\Gamma) - 2Tr(\Gamma^T\Sigma^{k'}) = Tr((\Sigma^{>k'})^T\Sigma^{>k'}) + Tr((\Sigma^{k'})^T\Sigma^{k'}) - 2Tr(\Gamma^T\Sigma^{k'}) + Tr(\Gamma^T\Gamma) = Tr((\Sigma^{>k'})^T\Sigma^{>k'}) + \|\Sigma^{k'} - \Gamma\|_F^2 \geq Tr((\Sigma^{>k'})^T\Sigma^{>k'}) = \|\Sigma^{>k'}\|_F^2 = \sigma_{k'+1}^2 + \sigma_{k'+2}^2 + ... + \sigma_p^2 \underset{k' \leq k}{\geq} \sigma_{k+1}^2 + \sigma_{k+2}^2 + ... + \sigma_p^2$$

Donc $\|A-B\|_F \geq \sqrt{(}\sigma_{k+1}^2 + \sigma_{k+2}^2 + ... + \sigma_p^2) = \|A - A_k\|_F \Longrightarrow \underset{rang(B)\leq k}{min} \|A - B\|_F \geq \sqrt{(}\sigma_{k+1}^2 + \sigma_{k+2}^2 + ... + \sigma_p^2) = \|A - A_k\|_F$.

Donc on déduit que $\underset{rang(B)\leq k}{min} \|A-B\|_F = \sqrt{(}\sigma_{k+1}^2 + \sigma_{k+2}^2 + ... + \sigma_p^2) = \|A - A_k\|_F$ □

# 5 Le calcul sur ordinateur d'une SVD

Le calcule sur ordinateur d'une SVD se fait généralement en 2 étapes :

1. D'abord on calcule des matrices orthogonales (ou à colonnes orthonormées) $U$ et $V$ de sorte que $B = U^TAV$ est une matrice bidiagonale. Ceci peut se faire par des méthodes directes (la technique de Householder), ou alors des méthodes itératives (Bidiagonalisation de Golub-Kahan-Lanczos).

2. Ensuite on calcule les éléments propres de la matrice tridiagonale $B^T B$ par la méthode itérative $QR$ (ou une variante pour des matrices bidiagonales).

On devrait dans cette section comprendre l'ensemble de ces méthodes et leur implémentation, et comparer leur complexité.

## 5.1 Pourquoi la bidiagonalisation

Nous savons que la décomposition en valeurs singulières d'une matrice A de taille $m \times n$ ($m \geq n$) peut être écrite comme

$$A = U\Sigma V^T; \tag{15}$$

où $U = \Big[u_1, ..., u_m\Big]$ est une matrice $m \times m$ unitaire, $V = \Big[v_1, ..., v_n\Big]$ est une matrice $n \times n$ unitaire, et $\Sigma$ est une matrice $m \times n$ diagonale avec des entrées diagonales réelles $\sigma_i$, de i = 1,...,n.

On a deja demontrer que le problème du calcul des triplets $(\sigma_i, u_i, v_i)$ de la matrice $A$ peut être formulé comme un problème de valeurs propres impliquant une matrice hermitienne liée à $A$, comme le matrice $A^T A$, et les valeurs singulières de $A$ sont les racines carrées non négatives des valeurs propres de la matrice $A^T A$.

il n'est pas recommandé de calculer explicitement la matrice de produits croisés $A^T A$, en particulier dans le cas où $A$ est creuse, alors la bidiagonalisation a été proposée comme moyen de tridiagonaliser la matrice $A^T A$ sans la former explicitement. Considérez la décomposition

$$A = PBQ^T \tag{16}$$

$$B = \left[\begin{array}{cccccc} \alpha_1 & \beta_1 & \cdots & \cdots & \cdots & 0 \\ 0 & \alpha_2 & \beta_2 & & & \vdots \\ \vdots & \ddots & \alpha_3 & \beta_3 & & \vdots \\ \vdots & & & \ddots & \ddots & \vdots \\ \vdots & & & & \alpha_{n-1} & \beta_{n-1} \\ 0 & \cdots & \cdots & \cdots & 0 & \alpha_n \\ \hline & & & 0 & & \end{array}\right]$$

où $P = \left[p_1, ..., p_m\right]$ et $Q = \left[q_1, ..., q_n\right]$ sont des matrices unitaires, et $B$ est une matrice bidiagonale supérieure de taille $m \times n$. Alors la matrice tridiagonale $B^T B$ est similaire (de facon unitaire) à $A^T A$. De plus, il existe des méthodes spécifiques qui calculent les valeurs singulières de $B$ sans former $B^T B$. Par conséquent, après avoir calculé le SVD de $B$,

$$B = X \Sigma Y^T \tag{17}$$

il ne reste plus qu'à combiner (16) et (17) pour obtenir la solution du problème (15) avec $U = PX$ et $V = QY$.

La bidiagonalisation peut être accomplie au moyen de transformations Householder ou alternativement via les récurrences de Golub-Kahan-Lanczos.

## 5.2 La méthode de Bidiagonalisation de Householder

**Définition 5.1.**
*On appelle* ***matrice élémentaire de Householder*** *une matrice $H$ de la forme*

$$H = Id - 2uu^T,$$

*avec $u \in \mathbb{R}^n$ et $\|u\|_2 = 1$. Par abus de langage, on considérera que la matrice identité est également une matrice de Householder.*

**Theorem 5.2.**
*Toute matrice de Householder $H = Id - 2uu^T$ est symétrique et orthogonale.*

**Theorem 5.3.**
*Soient $a$ et $b$ deux vecteurs de $\mathbb{R}^n$ non colinéaires, avec $\|b|_2 = 1$. Alors, il existe*
*$u \in \mathbb{R}^n$ avec $\|u\|_2 = 1$ et un réel $\alpha \in \mathbb{R}$ tels que si*

$$H = Id - 2uu^T$$

*alors*

$$Ha = \alpha b$$

*Démonstration.*
Puisque $H$ est orthogonale, on a $||Ha|| = ||\alpha b||$ donc $||Ha|| = |\alpha|$ alors $||a|| = |\alpha| \underbrace{||b||}_{=1}$.
Ce qui donne deux choix pour $\alpha$ : $||a||$ ou $-||a||$.
$Ha = \alpha b$ s'écrit $a - 2u\underbrace{u^T a}_{\lambda} = \alpha b \iff a - \alpha b = 2\lambda u$.

Pour calculer $\lambda$ faisons le profuit scalaire de l'égalité précédente par $a$. On obtient :

$$\alpha^2 - \alpha a^T b = a^T a - \alpha a^T b = 2\lambda a^T u = 2\lambda^2 = \beta \quad (*)$$

Les vecteurs $a$ et $b$ n'étant pas colinéaires $|a^T b| < ||a|| \cdot ||b|| = |\alpha|$
et l'égalité précédente permet effectivement de définir $\lambda \neq 0 \qquad a^T a - \alpha a^T b = \alpha^2 - \alpha a^T b > 0$.
$u$ est alors défini par $u = \dfrac{1}{2\lambda}\left[a - \alpha b\right]$
La matrice $H$ ne dépend pas du signe de $\lambda$

□

---

**(Householder vector)** Étant donnée $x \in \mathbb{R}^m$, cette fonction calcule $v \in \mathbb{R}^m$ avec $v(1) = 1$ et $\beta \in \mathbb{R}$ tels que $P = I_m - \beta v v^T$ est orthogonale et $Px = \|x\|_2 e_1$.

**fonction**$[v, \beta] =$ **house**$(x)$
$m =$ longueure$(x)$, $\sigma = x(2:m)^T x(2:m)$, $v = \begin{bmatrix} 1 \\ x(2;m) \end{bmatrix}$
**Si** $\sigma = 0$ et $x(1) > 0$
    $\beta = 0$
**Sinon si** $\sigma = 0$ et $x(1) < 0$
    $\beta = -2$
**Sinon**
    $u = \sqrt{x(1)^2 + \sigma}$
    **Si** $x(1) \leq 0$
        $v(1) = x(1) - \mu$
    **Sinon**
        $v(1) = -\dfrac{\sigma}{x(1) + \mu}$
    **Fin**
    $\beta = \dfrac{2v(1)^2}{\sigma + v(1)^2}$
    $v = \dfrac{v}{v(1)}$
**Fin**

---

On suppose $A \in \mathbb{R}^{m \times n}$ avec $m \geq n$. L'idée est de calculé les matrices orthodonales $U_B$ de dimension $m \times m$ et $V_B$ de dimension $n \times n$

$$U_B^T A V_B = \left[\begin{array}{cccccc} d_1 & f_1 & 0 & \cdots & \cdots & 0 \\ 0 & d_2 & f_2 & & & 0 \\ \vdots & \ddots & d_3 & f_3 & & \vdots \\ \vdots & & & \ddots & \ddots & \vdots \\ \vdots & & & & d_{n-1} & f_{n-1} \\ 0 & \cdots & \cdots & \cdots & 0 & d_n \\ \hline & & & 0 & & \end{array}\right]$$

$U_B = U_1 \cdots U_n$ et $V_B = V_1 \cdots V_{n-2}$ peuvent chacun être déterminées comme un produit de matrices de Householder, par exemple :

$$\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix} \xrightarrow{U_1} \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix} \xrightarrow{V_1}$$

$$\begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix} \xrightarrow{U_2} \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix} \xrightarrow[V_2]{}$$

$$\begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & 0 \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix} \xrightarrow{U_3} \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & 0 \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \\ 0 & 0 & 0 & \times \end{bmatrix} \xrightarrow{U_4} \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & 0 \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

En général, $U_k$ introduit des zéros dans le kième colonne, tandis que $V_k$ zéros les entrées appropriées dans la ligne k.
Dans l'ensemble, nous avons :

$$
\left\{
\begin{array}{l}
\textbf{Pour } j = 1 : n \\
\qquad [v, \beta] = \text{house}(A(j : m, j)) \\
\qquad A(j : m, j : n) = \left(I_{m-j+1} - \beta v v^T\right) A(j : m, j : n) \\
\qquad A(j + 1 : m, j) = v(2 : m - j + 1) \\
\qquad \textbf{Si } j \leq n - 2 \\
\qquad\qquad [v, \beta] = \text{house}(A\,(j, j + 1 : n)^T) \\
\qquad\qquad A(j : m, j + 1 : n) = A(j : m, j + 1 : n)(I_{n-j} - \beta v v^T) \\
\qquad\qquad A(j, j + 2 : n) = v(2 : n - j)^T \\
\qquad \textbf{Fin}
\end{array}
\right.
$$

Cet algorithme demande $4mn^2 - 4n^3/3$ flops .

### 5.3 La méthode de Bidiagonalisation de Golub-Kahan-Lanczos

La technique de bidiagonalisation de Golub-kahan-Lanczos peut être dérivée de plusieurs perspectives équivalentes. Considérons une version compacte de (16),

$$A = P_n B_n Q_n^T \tag{18}$$

où les lignes nulles de la matrice bidiagonale $B$ ont été supprimées et, par conséquent, $P_n$ est maintenant une matrice $m \times n$ avec des colonnes orthonormées, $Q_n$ est une matrice unitaire d'ordre $n$ qui est égale à $Q$ de (16), et $B_n$ est un matrice carré d'ordre $n$ qui peut s'écrire

$$
B_n = P_n^T A Q_n = \begin{bmatrix}
\alpha_1 & \beta_1 & \cdots & \cdots & \cdots & 0 \\
0 & \alpha_2 & \beta_2 & & & \vdots \\
\vdots & \ddots & \alpha_3 & \beta_3 & & \vdots \\
\vdots & & & \ddots & \ddots & \vdots \\
\vdots & & & & \alpha_{n-1} & \beta_{n-1} \\
0 & \cdots & \cdots & \cdots & 0 & \alpha_n
\end{bmatrix}
$$

Pré-multipliant $B_n$ par $P_n$, on a la relation $AQ_n = P_n B_n$. Aussi, si nous transposons les deux cotés de $B_n$ et pré-multiplions par $Q_n$, nous obtenons $A^T P_n = Q_n B_n^T$. L'égalité des $k \leq n$ premières colonnes des deux relations aboutit à

$$AQ_k = P_k B_k, \tag{19}$$

$$A^T P_k = Q_k B_k^T + \beta_k q_{k+1} e_k^T, \tag{20}$$

où $B_k$ désigne la sous-matrice principale de taille $k \times k$ de $B_n$. Des expressions analogues peuvent être écrites sous forme vectorielle en assimilant la jème colonne uniquement,

$$Aq_j = \beta_{j-1} p_{j-1} + \alpha_j p_j, \tag{21}$$

$$A^T p_j = \alpha_j q_j + \beta_j q_{j+1}, \tag{22}$$

Ces expressions donnent directement la double récursion

$$\alpha_j p_j = Aq_j - \beta_{j-1} p_{j-1}, \tag{23}$$

$$\beta_j q_{j+1} = A^T p_j - \alpha_j q_j, \tag{24}$$

avec $\alpha_j = ||Aq_j - \beta_{j-1} p_{j-1}||_2$ et $\beta_j = ||A^T p_j - \alpha_j q_j||_2$, car les colonnes de $P_n$ et $Q_n$ sont normalisées. l'iteration de l'algorithme devrait s'arrêter si $\beta_k = 0$, car d'après (19) et (20) et dans le cas où $\beta_k = 0$ on a

$$AQ_k = P_k B_k,$$

$$A^T P_k = Q_k B_k^T,$$

Et ainsi on obtient notre but

$$A^T A Q_k = Q_k B_k^T B_k,$$

mais dans le cas où notre $\beta_k$ ne s'annule pas avant $k = n$, on peut s'arreter à n'import indice $k \leq n$ et ça depend du nombre de triplés singuliers de $A$ $(\sigma_i, u_i, v_i)$ demandés .

**Algorithm 1** Golub-Kahan-Bidiagonalisation avec nombre d'iteration maximal $k$

choisissez un vecteur de norme 1 $q_1$
**Régler** $\beta_0 = p_0 = 0$
**Pour** $j = 1, 2, ..., k$
    $p_j = Aq_j - \beta_{j-1} p_{j-1}$
    $\alpha_j = ||p_j||_2$
    $p_j = p_j / \alpha_j$
    $q_{j+1} = A^T p_j - \alpha_j q_j$
    $\beta_j = ||q_{j+1}||_2$
    **Si** $\{\beta_j = 0\}$
        Break
    $q_{j+1} = q_{j+1} / \beta_j$
**Fin**

**pourquoi les vecteur $(p_j)_{1\leq i\leq k}$ et $(q_j)_{1\leq i\leq k}$ sont orthogonales entre eux ?**

Pour démontrer l'orthogonalité, il faut utilisé l'itération de Arnoldi, cette itération utilise le processus de Gram-Schmidt modifié sur une matrice symetrique $C$ pour produire une séquence de vecteurs orthonormés, $W_1$, $W_2$, $W_3$,..., appelés vecteurs d'Arnoldi, tels que pour tout n, les vecteurs $W_1$,..., $W_n$ couvrent le sous-espace de krylov $K_n$. Explicitement, l'algorithme est le suivant :

1. commencez par un vecteur arbitraire $W_1$ de norme 1.
2. Répétez pour $k = 2, 3, ...$

   $W_k \leftarrow$C $W_{k-1}$

   Répétez pour $j = 1, ..., k-1$

   $h_{j,k-1} \leftarrow W_j^T W_k$

   $W_k \leftarrow W_k - h_{j,k-1} W_j$

   $h_{k,k-1} \leftarrow ||W_k||$

   $W_k \leftarrow \dfrac{W_k}{h_{k,k-1}}$

si nous prenons $C = \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix}$ qui est symetrique, et on prend $W_1 = \begin{bmatrix} 0 \\ q_1 \end{bmatrix}$ telque $q_1$ est arbitraire (de l'algorithm 1), et si nous appliquons l'itération de Arnoldi sur $C$ et $W_1$, on trouve par une calcul matricielle simple que $W_2 = \begin{bmatrix} p_1 \\ 0 \end{bmatrix}$, $W_3 = \begin{bmatrix} 0 \\ q_2 \end{bmatrix}$, $W_4 = \begin{bmatrix} p_2 \\ 0 \end{bmatrix}$, $W_5 = \begin{bmatrix} 0 \\ q_3 \end{bmatrix}$, ..., qui sont orthogonales d'après l'iteration de Arnoldi, donc les les vecteur $(p_j)_{1\leq i\leq k}$ et $(q_j)_{1\leq i\leq k}$ sont orthogonales.

**Faire face à la perte d'orthogonalité entre les vecteurs $(p_j)_{1\leq i\leq k}$ et $(q_j)_{1\leq i\leq k}$.** Pour traiter la perte d'orthogonalité mutuelle entre les $(p_j)_j$ et $(q_j)_j$ après un nombre d'étapes $k$ trés grande, il faut utilisé la solution (algorithme 2) proposée dans l'article fondateur de golub et kahan**[1]**, cette solution s'appelle (full orthogonalization) ou orthogonalisation complète. Donc l'orthogonalisation complète revient à orthogonaliser le vecteur $p_j$ explicitement par rapport à tous les vecteurs de Lanczos gauche (les $(p_i)_{1\leq i\leq j-1}$) précédemment calculés, et orthogonaliser le vecteur $q_{j+1}$ explicitement par rapport à tous les vecteurs de Lanczos droits (les $(q_i)_{1\leq i\leq j}$) précédemment calculés. L'algorithme 2 montre cette variante avec une procédure d'orthogonalisation de Gram-Schmidt modifiée. Notez que dans l'algorithme 2 concernant le calcul de $p_j$ il n'est plus nécessaire de soustraire le terme $\beta_{j-1}p_{j-1}$ comme dans (23), puisque cela est déjà fait dans l'étape d'orthogonalisation, une remarque similaire vaut pour le calcul de $q_{j+1}$.

**Algorithm 2** Golub-Kahan-Bidiagonalisation avec orthogonalisation complète et nombre d'iteration maximal $k$

choisissez un vecteur de norme 1 $q_1$
**Pour** $j = 1, 2, ..., k$
$p_j = Aq_j$
**Orthogonaliser $p_j$ avec $P_{j-1}$**
**Pour** $i = 1, 2, ..., j-1$
$\gamma = p_i^T p_j$
$p_j = p_j - \gamma p_i$
**Fin**
$\alpha_j = ||p_j||_2$
$p_j = p_j/\alpha_j$
$q_{j+1} = A^T p_j$
**Orthogonaliser $q_{j+1}$ avec $Q_j$**
**Pour** $i = 1, 2, ..., j$
$\gamma = q_i^T q_{j+1}$
$q_{j+1} = q_{j+1} - \gamma q_i$
**Fin**
$\beta_j = ||q_{j+1}||_2$
**Si** $\{\beta_j = 0\}$
Break
$q_{j+1} = q_{j+1}/\beta_j$
**Fin**

### 5.3.1 Approximations de Ritz

L'idée de Ritz peut être appliquée pour extraire des valeurs singulières approximatives et des vecteurs singulières à gauche et à droite de $A$ à partir des matrices $P_k$, $Q_k$ et $B_k$ déjà calculer par l'algorithme 2, nous calculons simplement le SVD de la matrice bidiagonale $B_k$ (par des méthodes nous le mentionnerons et le détaillerons dans cette memoire plus tard)

$$X_k^T B_k Y_k = \Sigma = diag(\sigma_1, ..., \sigma_k) \tag{25}$$

alors forment les matrices

$$U_k = P_k X_k = \Big[u_1, ..., u_k\Big]$$

$$V_k = Q_k Y_k = \Big[v_1, ..., v_k\Big]$$

et il découle de (19), (20) et (25)

$$AV_k = U_k\Sigma,$$

$$A^T U_k = V_k\Sigma + q_{k+1}e_k^T X_k,$$

et donc pour i=1,...,k on a

$$Av_i = \sigma_i u_i, \tag{26}$$

$$A^T u_i = \sigma_i v_i + \beta_k q_{k+1} e_k^T x_i, \tag{27}$$

Donc à partir de (26) et (27) on trouve

$$A^T A v_i = \sigma_i^2 v_i + \beta_k q_{k+1} e_k^T x_i. \tag{28}$$

et donc le triplet $\{\sigma_i, u_i, v_i\}$ est appelé triplet singulier de Ritz pour le matrice $A^T A$ et son norme résiduelle associée est

$$||r_i||_2 = \beta_k |e_k^T x_i| \ .$$

Maintenant on veut trouver les valeurs propres de notre matrice tridiagonale $B^T B$.

## 5.4 La Méthode itérative QR

Cette méthode s'applique aux matrices quelconques. voici le principe de base de cette méthode.

Posons $A_1 = A$. on écrit la factorisation $QR$ de $A_1$ (par l'algorithme de householder ou de Givens)

$$A_1 = Q_1 R_1 \text{ où } Q_1 \text{ est une matrice unitaire et } R_1 \text{ est une matrice triangulaire supérieure.}$$

et on forme

$$A_2 = R_1 Q_1 = Q_1^T A_1 Q_1$$

A l'étape $k$,

$$A_k = Q_k R_k$$

on pose

$$A_{k+1} = R_k Q_k = Q_k^T A_k Q_k$$

par induction, on obtient

$$A_{k+1} = Q_k^T Q_{k-1}^T ... Q_1^T A Q_1 Q_2 ... Q_k$$

ce qui montre que $A_{k+1}$ est semblable à $A$ donc admet les memes valeurs propres.

Une itération de l'algorithme $QR$ appliquée à une matrice pleine coute cher, le nombre de multiplications et d'additions est de l'ordre de $\frac{2N^3}{3}$ pour la factorisation $A_k = Q_k R_k$, et de l'ordre de $\frac{N^3}{2}$ pour effectuer le produit $A_{k+1} = R_k Q_k$ où on connait ($Q_k$ et $R_k$).

Par contre, on verra que si $A_k$ est de la forme Hessenberg, $A_{k+1}$ l'est aussi. On pourra alors réduire le cout à $4N^2$ multiplications et additions (remarquons que dans notre cas on applique la méthode $QR$ sur $B^T B$ qui est deja tridiagonalisable donc de Hessenberg).

De plus, on accélérera la convergence de l'algorithme en effectuant des translations d'origine.

Ces deux améliorations font que la méthode $QR$ est la meilleure méthode pour obtenir toutes les valeurs propres d'une matrice.

### 5.4.1 Théorème

Soit $A \in \mathbb{R}^{d \times d}$ une matrice carrée inversible. On note $\lambda_1, ..., \lambda_d$ ses valeurs propres sur lesquelles on fait l'hypothèse

$$|\lambda_1| > |\lambda_2| > ... > |\lambda_d|.$$

On suppose, de plus, que $A$ s'écrit $A = PDP^{-1}$, avec une matrice $P$ admettant une décomposition $LU$ et $D$ la matrice diagonale telle que $D_{ii} = \lambda_i$ .

alors la méthode $QR$ définie plus haut converge, au sens où

$$(A_k)_{ii} \underset{k}{\to} \lambda_i \qquad \forall i = 1, 2, ..., d$$

$$(A_k)_{ij} \underset{k}{\to} 0 \qquad \forall j < i$$

### 5.4.2 Est-ce que la méthode QR converge ?

Oui, mais ça depend, si $A \in \mathbb{R}^{d \times d}$ une matrice carrée inversible. On note $\lambda_1, ..., \lambda_d$ ses valeurs propres sur lesquelles on fait l'hypothèse

$$|\lambda_1| > |\lambda_2| > ... > |\lambda_d|.$$

On suppose, de plus, que $A$ s'écrit $A = PDP^{-1}$, avec une matrice $P$ admettant une décomposition $LU$ et $D$ la matrice diagonale telle que $D_{ii} = \lambda_i$
.

alors la méthode $QR$ définie plus haut converge, au sens où

$$(A_k)_{ii} \underset{k}{\to} \lambda_i \qquad \forall i = 1, 2, ..., d$$

$$(A_k)_{ij} \underset{k}{\to} 0 \qquad \forall j < i$$

Pratiquement, on n'applique jamais directement la méthode $QR$ à la matrice $A$ mais à une matrice $H$ Hessenberg semblable à $A$, mais da ns notre cas $A = B^T B$ alors on peut utiliser directement la factorisation $QR$ d'une matrice de hessenberg.

### 5.4.3 Factorisation QR d'une matrice de Hessenberg

Nous expliquons dans cette section comment implémenter efficacement une étape de la méthode $QR$ quand on part d'une matrice $A^{(0)} = H^{(0)}$ sous la forme de Hessenberg. Pour tout $k \geq 1$, la première phase consiste à calculer la factorisation $QR$ de $H^{(k-1)}$ au moyen de $n-1$ rotations de Givens

$$(Q^{(k)})^T H^{(k-1)} = (G_{n-1}^{(k)})^T ... (G_1^{(k)})^T H^{(k-1)} = R^{(k)}, \tag{29}$$

où, pour $j = 1, ..., n-1$, $G_j^{(k)}$ est, pour $k \geq 1$, la j-ième matrice de rotation de Givens.

L'étape suivante consiste à compléter la transformation orthogonale

$$H^{(k)} = R^{(k)} Q^{(k)} = R^{(k)} (G_1^{(k)} ... G_{n-1}^{(k)}). \tag{30}$$

La matrice orthogonale $Q^{(k)} = (G_1^{(k)} ... G_{n-1}^{(k)})$ est de la forme de Hessenberg.

Dans l'algorithme ci-aprés, $[c, s] = givens(a, b)$ donne $c = \dfrac{a}{\sqrt{a^2 + b^2}}$ et $s = \dfrac{b}{\sqrt{a^2 + b^2}}$.

**Algorithm 3** Factorisation $QR$ pour matrice de Hessenberg avec rotations de Givens

**Entrée :** Une matrice de Hessenberg $A$ de taille $n \times n$

**Sortie :** Une matrice triangulaire supérieur $T$ tel que $A = UTU^T$ pour une matrice orthogonale $U$.

**Définir** $A_0 = A$

**Pour** $k = 1, 2, ..., m$

**Une étape QR de Hessenberg**

$H = A_{k-1}$

**Pour** $i = 1, 2, ..., n-1$

$[c_i, s_i] = givens(h_{i,i}, h_{i+1,i})$

$H_{i:i+1,i:n} = \begin{bmatrix} c_i & s_i \\ -s_i & c_i \end{bmatrix} H_{i:i+1,i:n}$

**Fin**

**Pour** $i = 1, 2, ..., n-1$

$H_{1:i+1,i:i+1} = H_{1:i+1,i:i+1} \begin{bmatrix} c_i & -s_i \\ s_i & c_i \end{bmatrix}$

**Fin**

$A_k = H$

**Fin**

**Retour** $T = A_m$

### 5.4.4 Factorisation QR avec la Méthode de Householder

- Étape 1 : Soit $a_1 \in \mathbb{R}^m$ la première colonne de la matrice $A_0 = A$ et $e_1$ le premier vecteur de la base canonique de $\mathbb{R}^m$.
  On détermine $H_1 = Id_m - \frac{1}{\beta} v_1 v_1^T$ matrice de Householder de $\mathcal{M}_m(\mathbb{R})$ telle que $H_1 a_1 = \alpha_1 e_1$. Si $a_1$ et $e_1$ sont colinéaires, il suffit de prendre $H_1 = Id_m$.
  La matrice $A_1 = H_1 A_0$ aura donc sa première colonne égale à $\alpha_1 e_1$. On obtient

$$A_1 = \left(\begin{array}{c|ccc} \alpha_1 & * & \cdots & * \\ \hline 0 & & & \\ \vdots & & \tilde{A}_1 & \\ 0 & & & \end{array}\right)$$

  Avec $\tilde{A}_1$ la sous-matrice des $(m-1)$ dernières lignes et $(n-1)$ dernières colonnes de $A_1$.

- Étape 2 : On applique le même principe à la sous-matrice $\tilde{A}_1$. On

détermine $\tilde{H_2} = Id_{m-1} - \frac{1}{\beta_2}\tilde{v_2}\tilde{v_2}^T$ matrice de Householder telle que $\tilde{H_2}\tilde{a_1} = \alpha_2\tilde{e_1}$, où $\tilde{a_1}$ est la première colonne de $\tilde{A_1}$ et $\tilde{e_1}$ le premier vecteur de la base canonique de $\mathbb{R}^{m-1}$, pui on définit :

$$H_2 = \left(\begin{array}{c|ccc} 1 & 0 & \cdots & 0 \\ \hline 0 & & & \\ \vdots & & \tilde{H_2} & \\ 0 & & & \end{array}\right)$$

En définissant $v_2 = \begin{pmatrix} 0 \\ \tilde{v_2} \end{pmatrix}$, alors on a : $H_2 = Id_m - \frac{1}{\beta_2}v_2{v_2}^T$.

On pose :

$$A_2 = H_2A_1 = \left(\begin{array}{c|c|ccc} \alpha_1 & * & * & \cdots & * \\ \hline 0 & \alpha_2 & * & \cdots & * \\ \hline 0 & 0 & & & \\ \vdots & \vdots & & \tilde{A_2} & \\ 0 & 0 & & & \end{array}\right)$$

- Étapes suivantes : On itère le procédé jusqu'au bout écrivant :

$$A_i = H_iA_{i-1} \text{ pour } 1 \leq i \leq n-1,$$

où $\tilde{H_i} \in \mathcal{M}_{m-i+1}(\mathbb{R})$ est une matrice de Householder et $H_i \in \mathcal{M}_m(\mathbb{R})$ a la structure suivante :

$$H_i = \left(\begin{array}{ccc|c} 1 & & & \\ & \ddots & & \\ & & 1 & \\ \hline & & & \tilde{H_i} \end{array}\right)$$

Comme $H_i^{-1} = H_i^T = H_i$, on aura à la fin :

$$A = A_0 = \underbrace{\prod_{i=1}^{n-1} H_i}_{Q}\ \underbrace{A_{n-1}}_{R} = QR,$$

Puisque le produit de matrices orthogonales est une matrice orthogonale et que $A_{n-1} = R = \begin{pmatrix} \overline{R} \\ 0 \end{pmatrix}$ avec $\overline{R} \in \mathcal{M}_n(\mathbb{R})$ triangulaire supérieure. Il reste à noter que les coefficients diagonaux de $\overline{R}$ ne sont ici pas forcément positifs !

### 5.4.5 Factorisation QR pour les matrices tridiagonales avec shift de Wilkinson

Dans le cas symétrique, l'algorithme QR de Hessenberg devient un algorithme QR tridiagonal. Cela peut être exécuté de manière explicite ou implicite. Sous la forme explicite, une étape QR est essentiellement.

---

1. Choisissez un shift $\mu$.
2. Calculer la factorisation QR $A - \mu I = QR$.
3. Mise à jour $A$ par $A = RQ + \mu I$.

---

Bien sûr, cela se fait au moyen de rotations d'avions et en respectant la symétrie structure tridiagonale de $A$.
Sous la forme implicite plus élégante de l'algorithme, nous calculons d'abord les premiers rotations $G_0 = G(1, 2, \vartheta)$ de la factorisation $QR$ qui annule l'élément $(2, 1)$ de $A - \mu I$,

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a_{11} - \mu \\ a_{21} \end{bmatrix} = \begin{bmatrix} * \\ 0 \end{bmatrix}, \quad c = \cos(\vartheta_0), \quad s = \sin(\vartheta_0).$$

Effectuer une transformation similaire avec $G_0$ que nous avons $(n = 5)$

$$G_0^* A G_0 = A' = \begin{bmatrix} \times & \times & + & & \\ \times & \times & \times & & \\ + & \times & \times & \times & \\ & & \times & \times & \times \\ & & & \times & \times \end{bmatrix}$$

Comme avec le double étape l'algorithme de Hessenberg $QR$ nous poursuivons le renflement vers le bas de la diagonale. Dans les $5 \times 5$ exemple cela devient

$$A \underset{=G(1,2,\vartheta_0)}{\xrightarrow{G_0}} \begin{bmatrix} \times & \times & + & & \\ \times & \times & \times & & \\ + & \times & \times & \times & \\ & & \times & \times & \times \\ & & & \times & \times \end{bmatrix} \underset{=G(2,3,\vartheta_1)}{\xrightarrow{G_1}} \begin{bmatrix} \times & \times & 0 & & \\ \times & \times & \times & + & \\ 0 & \times & \times & \times & \\ & + & \times & \times & \times \\ & & & \times & \times \end{bmatrix}$$

$$\underset{=G(3,4,\vartheta_2)}{\xrightarrow{G_2}} \begin{bmatrix} \times & \times & 0 & & \\ \times & \times & \times & & \\ & \times & \times & \times & + \\ & 0 & \times & \times & \times \\ & & + & \times & \times \end{bmatrix} \underset{=G(4,5,\vartheta_3)}{\xrightarrow{G_3}} \begin{bmatrix} \times & \times & & & \\ \times & \times & \times & & \\ & \times & \times & \times & 0 \\ & & \times & \times & \times \\ & & 0 & \times & \times \end{bmatrix} = \bar{A}$$

L'étape complète est donnée par

$$\bar{A} = Q^* A Q, \quad Q = G_0 G_1 \cdots G_{n-2}.$$

Car $G_k \mathbf{e}_1 = \mathbf{e}_1$ pour $k > 0$ On a

$$Q\mathbf{e}_1 = G_0 G_1 \cdots G_{n-2}\mathbf{e}_1 = G_0 \mathbf{e}_1$$

L'étape $QR$ explicite et implicite forme la même première rotation d'avion $G_0$. Nous voyons que l'étape explicite et implicite $QR$ calcule essentiellement le même $A$.

---

**Algorithme Tridiagonal Symétrique $QR$ avec shift implicite de Wilkinson.**

---

1. Soit $T \in \mathbb{R}^{n\times n}$ une matrice tridiagonale with diagonal entries $a_1, \ldots, a_n$ and off-diagonal entries $b_2, \ldots, b_n$ Cet algorithme calcule les valeurs propres $\lambda_1, \ldots, \lambda_n$ de $T$ et les vecteurs propres correspondants $\mathbf{q}_1, \ldots, \mathbf{q}_n$. Les valeurs propres sont stockées dans $a_1, \ldots, a_n$. les vecteurs propres sont stockés dans la matrice $Q$, tel que $TQ = Q\,\mathrm{diag}\,(a_1, \ldots, a_n)$.
2. $m = n/^*$ La dimension du problème actuel . $m$ est reduit dans le contrôle de convergence. $*/$
3. $Q = I_n$
4. **tant que** $m > 1$ **faire**
5. $\quad d := (a_{m-1} - a_m)/2; /^*$ calcule le shift de Wilkinson $*/$
6. **Si** $d = 0$ **alors**
7. $\quad s := a_m - |b_m|$
8. **Sinon**
9. $\quad s := a_m - b_m^2 / \left(d + \mathrm{sign}(d)\sqrt{d^2 + b_m^2}\right)$
10. **fin si**
11. $x := a(1) - s; /^*$ Les étapes de $QR$ implicite commencent ici $*/$
12. $y := b(2);$
13. **pour** $k = 1$ à $m - 1$ **faire**
14. $\quad$ **Si** $\quad m > 2$ **alors**
15. $\quad\quad [c, s] := \mathrm{givens}(x, y);$

16. **Sinon**

17. Determiner $[c, s]$ tel que $\begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} a_1 & b_2 \\ b_2 & a_2 \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix}$ est diagonale

18. **fin si**

19. $w := cx - sy$;

20. $d := a_k - a_{k+1}; \quad z := (2cb_{k+1} + ds)\, s$

21. $a_k := a_k - z; \quad a_{k+1} := a_{k+1} + z;$

22. $b_{k+1} := dcs + \left(c^2 - s^2\right) b_{k+1}$

23. $x := b_{k+1}$

24. **Si** $k > 1$ **alors**

25. $b_k := w$

26. **fin si**

27. **Si** $k < m - 1$ **alors**

28. $y := -sb_{k+2}; \quad b_{k+2} := cb_{k+2}$

29. **fin si**

30. $Q_{1:n;k:k+1} := Q_{1:n;k:k+1} \begin{bmatrix} c & s \\ -s & c \end{bmatrix}$

31. **fin pour** /∗ Les étapes de $QR$ implicite se terminent ici $^*$/

32. **Si** $|b_m| < \varepsilon\,(|a_{m-1}| + |a_m|)$ **alors** /$^*$ contrôle de convergence $*$/

33. $m := m - 1$

34. **fin si**

35. **fin tant que**

---

L'algorithme précédent montre l'algorithme tridiagonal symétrique implicite $QR$. Les shifts de travail sont choisis en accord avec Wilkinson. Un problème qui n'est pas traité dans cet algorithme est la déflation. La déflation est une grande importance pratique. Considérons la situation suivante des $6 \times 6$

$$T = \begin{bmatrix} a_1 & b_2 & & & & \\ b_2 & a_2 & b_3 & & & \\ & b_3 & a_3 & 0 & & \\ & & 0 & a_4 & b_5 & \\ & & & b_5 & a_5 & b_6 \\ & & & & b_6 & a_6 \end{bmatrix}.$$

Le changement pour l'étape suivante est déterminé à partir des éléments $a_5, a_6$ et $b_6$. La première rotation du plan est déterminée à partir du Shift de travail et des éléments $a_1$ et $b_1$.
L'algorithme implicite de décalage poursuit alors le renflement vers le bas de la diagonale. Dans cette situation particulière, la procédure se termine déjà en ligne/colonne 4 parce que $b_4 = 0$.
Ainsi, le changement qui est une approximation d'une valeur propre du deuxième bloc (lignes 4 à 6) est appliquée à la mauvaise premier bloc (lignes 1 à 3). De toute évidence, ce changement n'améliore pas la convergence.
Si l'algorithme $QR$ est appliqué sous sa forme explicite, alors le premier bloc n'est toujours pas traité correctement, c'est-à-dire avec un (probablement) mauvais décalage, mais au moins le deuxième bloc est diagonalisé rapidement.

La déflation se fait comme indiqué dans l'lgorithme précédnt :

$$\text{Si } |b_k| < \varepsilon \left(|a_{k-1}| + |a_k|\right) \text{ alors deflate.}$$

La déflation est particulièrement simple dans le cas symétrique puisqu'elle signifie simplement qu'un problème des valeurs propres tridiagonal découple dans deux (ou plusieurs) petits problèmes des valeurs propres tridiagonal.
Notez, cependant, que les vecteurs propres sont encore $n$ éléments longs.

### 5.4.6 Comparaison avec la méthode de la puissance inverse avec shift

Donné $\mu \in \mathbb{K}$, on applique la méthode de la puissance inverse à la matrice $(A - \mu I)$.
La plus grande valeur propre en module de $(A - \mu I)^{-1}$ est alors $\dfrac{1}{\min\limits_i |\lambda_i - \mu|}$,

avec $\lambda_i$ les valeurs propres de $A$.
Après avoir choisi $q_0 \in \mathbb{K}^n$ tel que $||q_0||_\infty = 1$, on itère de la façon suivante :

$$\begin{cases} (A - \mu I)x_k = q_{k-1}, \\ \gamma_k = x_k(i) \text{ avec } |x_k(i)| = ||x_k||_\infty, \\ q_k = \dfrac{x_k}{\gamma_k}. \end{cases}$$

**Corollaire 5.4.**
*On considère une matrice $A \in \mathcal{M}_n(\mathbb{K})$ diagonalisable. Si $\lambda$ est la valeur propre de $A$ la plus proche de $\mu$ avec $|\lambda - \mu| < |\lambda_i - \mu|$ pour $\lambda_i \in \mathrm{Sp}\{A\} - \{\lambda\}$, et si $q_0$ n'est pas orthogonal au sous-espace propre à gauche associé à $\lambda$, alors :*

- *La direction de $q_k$ tend vers la direction du sous-espace propre associé à $\lambda$,*
- $\lim\limits_{k \longrightarrow \infty} \gamma_k = \dfrac{1}{\lambda - \mu}$

*Le facteur de convergence vaut* $\dfrac{|\lambda - \mu|}{\min\limits_{\lambda_i \in \mathrm{Sp}\{A\} - \{\lambda\}} |\lambda_i - \mu|}$.

### 5.4.7 Résultats du programme

```
*** Remote Interpreter Reinitialized  ***
>>>
La première valeur propre egale à 7.419360 pour mu=3.960000
La deuxième valeur propre egale à 7.413504 pour mu=3.970000
La troisième valeur propre egale à 7.410609 pour mu=3.975000
Les valeurs propre par sl.eig=[78.37702377+0.j  0.47298959+0.j  6.14998664+0.j]
C:\Users\Zahira\Desktop\zahira\ter_cpp\methode de puissance inverse.py:52: ComplexWarning: Casting complex values
  propre calculé par sl.eig %f"%(val1,e[2])
La valeur propre obtenue 7.419360 est proche à la valeur propre calculé par sl.eig 6.149987
>>>
```

# 6 Conclusion

En mathématiques, le procédé d'algèbre linéaire de décomposition en valeurs singulières d'une matrice est un outil important de factorisation des matrices rectangulaires réelles ou complexes.
Ses applications s'étendent du traitement du signal aux statistiques en passant par la météologie (à savoir : anticyclone, cyclone) et ce en tenant compte de ces changements climatiques avec des outils conçus à cet effet .
Le théorème spectral encore qu'une matrice normale peut être diagonalisée par une base orthonormée de vecteurs propres.
On peut voir la décomposition en valeurs singulières comme une généralisation du théorème spectral à des matrices arbitraires libre , artificielle,conventionnelle ,qui ne tient pas compte des exigences de la science ) qui ne sont pas nécessairement carrées.
En résumé,
Le SVD est une manière de factoriser une matrice non-carré qui généralise l'opération de décomposition en valeurs propres (pour une matrice symétrique)
La troncation des valeurs singulières produit des approximation de bas-rang d'une matrice d'entrée.
Le rang d'un matrice est les nombres des valeurs singulières non-nul.
Les matrices U et V sont orthogonales et unitaire : il préserve les distances entres les vecteurs de A.
La décomposition SVD pour une matrice non-carrée est unique.

## Références


[1] G. H. GOLUB AND W.KAHAN, *Calculating the singular values and pseudo-inverse of a matrix,* SIAM J. Numer.Anal., Ser.B,2(1965),pp.205-224.

[2] P. Lascaux,R. Théodor *analyse numérique matricielle appliquée à l'art de l'ingénieur* Tome 1. [Méthodes directes]

[3] P. Lascaux,R. Théodor *analyse numérique matricielle appliquée à l'art de l'ingénieur* Tome 2. [Méthodes itératives]

[4] Gene H.Golub, Charles F.Van Loan *Matrix Computations* 4th edition

[5] Text Mining https ://ia-data-analytics.fr/logiciel-data-mining/text-mining/text-mining-traiter-vos-donnees-textuelles/

[6] preuve de Eckart young avec norme de frobinus : https ://math.stackexchange.com/questions/3525372/prove-eckart-young-minsky-theorem-for-frobinius-norm

# A Annexe : CODE C++

## A.1 Code bidiagonalisation de Householder

```
//main.c++
#include <iostream>
#include <vector>
#include <fstream>
#include <math.h>
#include  "matricebande.h"
#include  "vecteur_template.cpp"
using namespace std;
struct return_house{
double Beta ;
vector<double>V ;
};
//declaration de la fonction matrice_bidiag et vector_house
return_house vector_house (vector<double>& x);
matrice Identity(int num);
int main()
{

return_house Y;


    int m=5, n=4;

matrice A(m,n);


    cout << "A dims " << A.dim1() << " " << A.dim2() << endl;

    A(0,0)= 4; A(0,1)=5 ;A(0,2)=3 ;A(0,3)=1 ;
    A(1,0)= 2; A(1,1)=2 ;A(1,2)=1 ;A(1,3)=3 ;
    A(2,0)=3 ; A(2,1)=4 ;A(2,2)=4 ;A(2,3)=1 ;
    A(3,0)=1 ;A(3,1)=2 ;A(3,2)=3 ;A(3,3)=4 ;
    A(4,0)=6 ;A(4,1)=5 ;A(4,2)=5 ;A(4,3)=1 ;
    cout << "Matrice A \n" << A;
```

```
matrice U;
matrice V;
matrice B;
matrice UU=Identity(m);
matrice VV=Identity(n);
matrice A_Org=A;


for (int j=0;j<n;j++)      // for j:n
{

  matrice A_P;
  vector<double> h;
  h= A.extract_col(j,j) ;
  Y=vector_house (h); // [v,beta]=house(Aj:m,j))

   matrice V_V_T,V_V_T_b;
   matrice O=A.coligne(Y.V);

   matrice OP=A.coligne(vector_zeros(Y.V,m));
   matrice O_T=O.transpose();
   matrice OP_T=OP.transpose();

   V_V_T=O_T*O;
   V_V_T_b=V_V_T*Y.Beta;
   A_P=Identity(m-j)*A.extract(j,j)-V_V_T_b*A.extract(j,j);//A(j:m,j:n)=(I-beta*v*v_t)
   U=Identity(m)-OP_T*OP*Y.Beta;
   A=A.update_matrice(A_P,j,j);
   vector<double> w ;
   w=colonne_start_end(Y.V,1,m-j);//v(2:m-j+1)
   A.update_colonne(w,j+1,j);//A(j+1:m,j)


   cout<<"start if"<<endl;
   if (j<=n-2)
    {
    matrice A_P_P;
    vector<double> g;
    g= A.extract_line(j,j+1) ;//A(j,j+1:n)T
    Y=vector_house (g);//house(A(j,j+1:n)T)
```

```
    matrice P=A.coligne(Y.V);
    matrice PP=A.coligne(vector_zeros(Y.V,n));
    matrice P_T=P.transpose();
    matrice PP_T=PP.transpose();
    matrice term =A.extract(j,j+1)*(P_T*P*Y.Beta);
    V=Identity(n)-PP_T*PP*Y.Beta;
    A_P_P=Identity(m-j)*A.extract(j,j+1)-term; //A(j:m,j+1:n)=(I-beta*v*v_t)A(j:m,j+1:
    A=A.update_matrice(A_P_P,j,j+1);
    vector<double> t;
    t=colonne_start_end(Y.V,1,n-j-1);//v(2:n-j)T
    A.update_ligne(t,j,j+1);//A(j,j+2:n)
    cout<<"A: \n"<<A<<endl;

    }

       VV=VV*V;
       UU=UU*U;
    }
  cout<<"La matrice U de SVD : \n"<<UU;
  cout<<"La matrice V de SVD : \n"<<VV;
   matrice C; //check of orthogonalité de U
   matrice UU_T=UU.transpose();

   C=UU_T*UU;

  cout << "U_T*U :  \n" << C<<endl;
  matrice K=UU_T*A;
  B=K*VV;
  cout <<"La matrice bidiagonale B: \n"<<K*VV<<endl;
  matrice B_T;
  B_T=B.transpose();
  matrice E;
  E=B_T*B;

  cout <<"B_T*B \n"<<E<<endl;


return 0;
}
```

```
//Definition de la fonction  matrice_bidiag et vector_house
return_house vector_house (vector<double>& x)
{
return_house A;
int m=x.size();
double sigma =0;


double beta;
vector<double> v(x.size());
vector<double> f(x.size());
double mu;
double c=1;


for(int i=1; i<x.size();i++)
{
sigma+=x[i]*x[i];
v[i]=x[i];
}
v[0]=1;

   if(sigma==0 & x[0]>=0)
    {
   beta=0;
    }
   else if(sigma==0 & x[0]<0)
    {
    beta=-2;
}
    else
    {
    mu=sqrt(pow(x[0],2) + sigma);
      if(x[0]<=0)
       {
       c=x[0]-mu;
       }
      else
      {
```

```
       c=(double)(-1*sigma)/(x[0]+mu);
       }

      beta=(double)(2*pow(c,2) )/(sigma + pow(c,2) );



}
for(int k=0;k<m;k++)

f[k]=(double)(v[k]/c) ;

A.V=f;
A.Beta=beta;

return A;
}
matrice Identity(int num)
{
    matrice A(num,num);
    int row, col;

    for (row = 0; row < num; row++)
    {
        for (col = 0; col < num; col++)
        {
            if (row == col)
                A(row,col)=1;
            else
                 A(row,col)=0;
        }

    }
    return A;
}

//fichier matrice.h

#if !defined (__IOSTREAM_H)
#include <iostream>
```

```
#endif
#if !defined (__FSTREAM_H)
#include <fstream>
#endif
#if !defined (__CASSERT_H)
#include <cassert>
#endif
#if !defined(__CMATH_H)
#include <cmath>
#endif
#if !defined (__VECTOR_H)
#include <vector>
#endif
#include <stdlib.h>

#include "vecteur_template.h"

using namespace std;

class matrice
{
private:
    int size1;  // dimension 1 de la matrice (nb de lignes)
    int size2;  // dimension 2 de la matrice (nb de colonnes)
    vector<double>* matrix;    // chaque ligne est un vector<double> de doubles

public:
    matrice() {this->size1 = this->size2 = 0;};  // constructeur par defaut
    matrice(int,int);                // constructeur en donnant les 2 dimensions
    matrice(const matrice&);         // constructeur par recopie
    ~matrice();                      // destructeur

    int dim1() const {return this->size1;}    // retourne la 1ere dimension de la matr
    int dim2() const {return this->size2;}    // retourne la 2eme dimension de la matr
    vector<double>& operator [] (int) const;  // retourne une ligne complete
    double& operator () (int,int) const;      // retourne un coefficient

    matrice& operator = (const matrice&); // surcharge de = (affectation)

    matrice transpose();    // transpose une matrice
```

```
    matrice coligne(const vector<double>& );//transpose vecteur
    vector<double> extract_col( int, const int);//extraire une colonne de la matrice A
    vector<double> extract_line(const int , const int );//extraire ligne de la matrice
    vector<double> colonne(const int);    // une colonne de la matrice
    matrice& update_colonne(const vector<double>& ,int ,int );//mise à jour d'un colon
    matrice& update_ligne(const vector<double>& ,int ,int );//mise à jour d'une ligne
    matrice& update_matrice(const matrice& ,int ,int );// mise à jour bloc dans la mat
    matrice extract(int k, int l);// extraire matrice depuis la matrice original

    matrice operator+(const matrice&); // somme de 2 matrices
    matrice operator-(const matrice&); // difference de 2 matrices
    matrice operator*(matrice&); // produit de 2 matrices, pas const matrice& à cause
    matrice operator * (const matrice& right);
    matrice operator*(const double&); // matrice x cte

    vector<double> operator*(const vector<double>&); // produit A*v
    friend vector<double> operator*(const vector<double>&, matrice&); // produit v^t *

    friend ostream& operator << (ostream&, const matrice &);        // surcharge << po
    friend istream& operator >> (istream&, const matrice &);        // surcharge >> po

    friend void save_matr(const char*, const matrice &); // ecriture dans un fichier
    friend matrice read_matr(const char *); // lecture dans un fichier

};

//-------------------------------------------------------
//-------------------------------------------------------
// Constructeur v1
matrice::matrice(int n, int m)
{
    assert((n>0)&&(m>0));
    this->size1 = n;
    this->size2 = m;
    this->matrix = new vector<double>[n];
    vector<double> v(m,0);
    for (int i=0; i<n ; i++)
        this->matrix[i]=v;
}
```

```
//-------------------------------------------------------
// Constructeur par copie
matrice::matrice(const matrice & mat)
{
  assert((mat.size1)&&(mat.size2));
  this->size1=mat.size1;
  this->size2=mat.size2;
  this->matrix=new vector<double>[size1];
  for (int i=0; i<mat.size1 ; i++)
      this->matrix[i]=mat.matrix[i];
}
//-------------------------------------------------------
// Destructeur
matrice::~matrice()
{
  if (this->size1||this->size2) delete[] this->matrix;
  this->size1=0;
  this->size2=0;
}
//-------------------------------------------------------
// Retourne la ligne i
vector<double>& matrice::operator [] (int i) const
{
  assert((i>=0)&&(i<this->size1));
  return this->matrix[i];
}
//-------------------------------------------------------
// Retourne le coefficient (i,j)
double& matrice::operator () (int i, int j) const
{
  assert((i>=0)&&(i<this->size1)&&(j>=0)&&(j<this->size2));
  return this->matrix[i][j];
}

//-------------------------------------------------------

// extraire la matrice depuis la ligne k->m et colonne l->n
matrice matrice::extract(int k, int l)
{
    //matrice T(this->size2,this->size1);
```

```
       int nb_ligne = this->size1-k;

    int nb_cl= this->size2-l;
    matrice T(nb_ligne,nb_cl);
    for (int i=0;i<nb_ligne; i++)
        for(int j=0;j<nb_cl; j++)
        T(i,j)= this->matrix[i+k][j+l];
    return T;
}
// extraire la colonne j avec les lignes commence de k->m
vector<double> matrice::extract_col( int k, const int l)
{

       int nb_ligne = this->size1-k;

        cout<<"k:"<<k<<endl;
    cout<<"this->size1:"<<this->size1<<endl;
    vector<double> vcol(nb_ligne);
    for (int i=0;i<nb_ligne; i++){
        vcol[i] = this->matrix[i+k][l];

    }
    return vcol;
}
// extraire la ligne k avec les colonnes commence de l->n
vector<double> matrice::extract_line(const int k, const int l)
{

    int nb_colonne = this->size2-l;
    vector<double> vcol(nb_colonne);
    for (int i=0;i<nb_colonne; i++)
        vcol[i] = this->matrix[k][i+l];
    return vcol;
}

// Surcharge = (affectation)
matrice& matrice::operator = (const matrice& mat)
{
  assert((mat.size1>0)&&(mat.size2>0));
```

```
  if ((!this->size1)||(!this->size2))
    {this->size1=mat.size1;
     this->size2=mat.size2;
     this->matrix=new vector<double>[size1];}
  assert((this->size1==mat.size1)&&(this->size2==mat.size2));
  for (int i=0; i<this->size1 ; i++)
    this->matrix[i]=mat.matrix[i];
  return (*this);
  }
//-------------------------------------------------------
// Transposee d'une matrice
matrice matrice::transpose()
{
    matrice T(this->size2,this->size1);
    for (int i=0; i<this->size2; i++)
        for (int j=0; j<this->size1; j++)
            T(i,j) = matrix[j][i];
    return T;
}
//-------------------------------------------------------
//Retourner la colonne ligne (zyada)<-------------------------------------
matrice matrice::coligne(const vector<double>& vcol)
{

    matrice T(vcol.size(),vcol.size());
        for (int j=0; j<vcol.size(); j++)
            T(0,j) = vcol[j] ;
    return T;
}
//-------------------------------------------------------
// Retourner la colonne j d'une matrice
vector<double> matrice::colonne (const int j)
{
    assert((j>=0)&&(j<this->size2));
    vector<double> vcol(this->size1);
    for (int i=0; i<this->size1; i++)
        vcol[i] = this->matrix[i][j];
    return vcol;
}
//-------------------------------------------------------
```

```
// mise à jour de la colonne " col" dans la matrice qui commnece par ligne "start_line
matrice& matrice::update_colonne(const vector<double>& vcol,int start_line,int col)
{

  int nb_ligne=this->size1-start_line;
  for (int i=0;i<vcol.size(); i++)
        //vcol[i] = this->matrix[i+k][l];
        this->matrix[i+start_line][col]=vcol[i];

  return (*this);
  }
// mise à jour de la ligne " line" dans la matrice qui commnece par la colonne  "start
 matrice& matrice::update_ligne(const vector<double>& vcol,int start_colonne,int line)
{

  int nb_colonne=this->size2-start_colonne;

  for (int i=0;i<vcol.size(); i++)
        //vcol[i] = this->matrix[i+k][l];
        this->matrix[line][i+start_colonne]=vcol[i];

  return (*this);
  }
// mise à jour d'un bloc dans la matrice commençant par le ligne startt_ligne->m et la
matrice& matrice::update_matrice(const matrice& mat,int start_ligne,int start_colonne)
{

  for (int i=0; i<mat.size1 ; i++)
    for(int j=0; j<mat.size2 ; j++)
    this->matrix[i+start_ligne][j+start_colonne]=mat.matrix[i][j];




  return (*this);
  }


// Surcharge operateur + non symetrique : somme de 2 matrices
matrice matrice::operator + (const matrice& A)
```

```
{
        assert((this->size1>0)&&(this->size1==A.size1));
        assert((this->size2>0)&&(this->size2==A.size2));
        matrice C(this->size1,this->size2);
        for (int i=0; i<this->size1; i++)
                C[i] = this->matrix[i]+A[i];

        return C;
}
//-------------------------------------------------------
// Surcharge operateur - non symetrique : difference de 2 matrices
matrice matrice::operator - (const matrice& A)
{
    assert((this->size1>0)&&(this->size1==A.size1));
    assert((this->size2>0)&&(this->size2==A.size2));
    matrice C(this->size1,this->size2);
    for (int i=0; i<this->size1; i++)
        for(int j=0;j<this->size2;j++)
        C(i,j) = -1*(A[i][j]-this->matrix[i][j]);
    return C;
}
//-------------------------------------------------------
// Surcharge operateur * non symetrique : produit de 2 matrices
matrice matrice::operator * (matrice& A) // pas const A à cause de la fonction transpo
{
    assert((this->size1>0)&&(this->size2>0)&&(A.size1>0)&&(A.size2>0));

    assert(this->size2==A.size1);
    matrice TA=A.transpose();
    matrice C(this->size1,A.size2);
    for (int i=0; i<this->size1; i++)
         for (int j=0; j<A.size2; j++)
             C(i,j) = matrix[i]*TA[j];
    return C;
}

// Surcharge operateur * non symetrique : produit de 2 matrices
matrice matrice::operator * (const matrice& right)
{
    matrice c(this->size1,right.size2);
```

```
    double sum_elems;
    for( int i = 0; i < this->size1 ; ++i)
    {
        for(int j = 0; j < right.size2 ; ++j)
        {
            sum_elems = 0;
            for( int k = 0; k < right.size1 ; ++k)
            {
                sum_elems += this->matrix[i][k] * right.matrix[k][j];
            }

            c.matrix[i][j] = sum_elems;
        }
    }
    return c;
}


//---------------------------------------------------------
// Surcharge operateur * non symetrique : matrice x cte
matrice matrice::operator * (const double& cte)
{
        assert((this->size1>0)&&(this->size2>0));
        matrice C(this->size1,this->size2);
        for (int i=0; i<this->size1; i++)
             C[i] = this->matrix[i]*cte;
        return C;
}
//---------------------------------------------------------
// Surcharge operateur * non symetrique : produit A*v
vector<double> matrice::operator * (const vector<double>& v)
{
    assert((this->size1>0)&&(this->size2>0)&&(this->size2==v.size()));
    vector<double> w(this->size1);
    for (int i=0; i<this->size1; i++)
        w[i] = this->matrix[i]*v;
    return w;
}
//---------------------------------------------------------
// Surcharge operateur * symetrique : produit v^t * A
```

```
vector<double> operator * (const vector<double>& v, matrice& A) // pas const A à cause
{
        assert((A.size1>0)&&(A.size2>0)&&(A.size1==v.size()));
        vector<double> w(A.size2);
        matrice TA=A.transpose();
        for (int i=0; i<A.size2; i++)
             w[i] = v*TA[i];
        return w;
}



//-------------------------------------------------------
// Surcharge operator << pour une matrice
ostream& operator << (ostream& s, const matrice & mat)
{
    assert((mat.size1>0)&&(mat.size2>0));
    for (int i=0; i<mat.size1; i++)
            s << mat[i];
    return s;
}
//-------------------------------------------------------
// Surcharge operator >> pour une matrice
istream& operator >> (istream& s, const matrice & mat)
{
//    assert((mat.size1>0)&&(mat.size2>0));
    cout << "Entrer au clavier une matrice avec " << mat.size1 <<" lignes et " << mat.s
    for (int i=0; i<mat.size1; i++)
        for (int j=0; j<mat.size2; j++)
            s >> mat(i,j);
    return s;
}
//-----------------------------------------
// Sauvegarde d'une matrice dans un fichier
void save_matr(const char *Nomfich, const matrice & mat)
{
    ofstream fichier;
    fichier.open(Nomfich); // ouverture du fichier
```

```
    assert(mat.size1>0 && mat.size2>0);
    fichier << "Dimension de la matrice = " << mat.dim1() << " " << mat.dim2() << "\n"
    for (int i=0; i<mat.size1 ; i++)
        fichier << mat[i];
    fichier.close();           // fermature du fichier
}
//--------------------------------------------
// lecture d'une matrice dans un fichier
// on va obtenir une matrice de la bonne taille
matrice read_matr(const char *Nomfich)
{
    ifstream fichier;
    fichier.open(Nomfich); // ouverture du fichier
    if (!fichier){
        cerr << "Probleme d'ouverture de fichier " << endl;
        exit(1);
    }

    int m,n;
    if (!fichier.eof())
      fichier >> m >> n;
    matrice M(m,n);

    for (int i=0;i<M.size1;i++){
        for (int j=0;j<M.size2;j++){
            fichier >> M(i,j);
            if (fichier.eof())
                cout << " Erreur dans le fichier, matrice incomplete \n ";
        }
    }
    fichier.close();   // fermeture du fichier
    return M;
}
//Fichier vecteur_template.cpp
#if !defined (__IOSTREAM_H)
#include <iostream>
#endif
#if !defined (__FSTREAM_H)
#include <fstream>
#endif
```

```
#if !defined (__ASSERT_H)
#include <assert.h>
#endif
#if !defined(__CMATH_H)
#include <cmath>
#endif
#if !defined (__VECTOR_H)
#include <vector>
#endif

using namespace std;

// Surcharge operator >> (vecteur vide au départ)
template <class T>
istream& operator >> (istream& s, vector<T> & vect){
    assert(vect.size()>0);
    cout << "Entrer au clavier un vecteur de taille " << vect.size() << endl;
    for (int i=0; i<vect.size(); i++)
        s >> vect[i];
    return s;
}

// Surcharge operator <<
template <class T>
ostream& operator << (ostream& s, const vector<T> & vect){
    assert(vect.size()>0);
    //cout << "Affiche le vecteur de taille " << vect.size() << endl;
    for (int i=0; i<vect.size(); i++)
        s << vect[i] << " ";
    s << endl;
    return s;
}

// Lire un vecteur depuis un fichier
//template <class T>
//void lire_vect(string nom_fichier, vector<T> & vect){
//    ifstream file;
//    T val;
//    file.open(nom_fichier,ios::in); // ouverture du fichier
//    assert(!file.fail()); // verifier ouverture correcte du fichier
```

```
//    while (!file.eof()){ // on lit tant qu'on est pas à la fin du fichier
//        file >> val;
//        vect.push_back(val);
//    }
//    file.close();
//    //return vect;
//    }
//
// Ecrire un vecteur dans un fichier
template <class T>
void ecrire_vect(const char* nom_fichier, vector<T> & vect){
    ofstream file;
    file.open(nom_fichier,ios::out); // ouverture du fichier
    assert(!file.fail()); // verifier ouverture correcte du fichier
    for (int i=0; i<vect.size(); i++)
        file << " " << vect[i];
    file.close();
}

// Ecrire deux vecteur dans un fichier
template <class T>
void ecrire_deux_vect(const char* nom_fichier, vector<T> & vect1,vector<T> & vect2){
    ofstream file;
    file.open(nom_fichier,ios::out); // ouverture du fichier
    assert(!file.fail()); // verifier ouverture correcte du fichier
    for (int i=0; i<vect1.size(); i++)
        file << vect1[i]<<" "<<vect2[i]<<endl;
    file.close();
}

// Surcharge operateur + : somme de 2 vecteurs
template <class T>
vector<T> operator + (const vector<T>& v1, const vector<T>& v2){
    assert((v1.size()>0)&&(v1.size()==v2.size()));
    vector<T> w(v1.size(),0);
    for (int i=0; i<v1.size(); i++)
         w[i]= v1[i]+v2[i];
    return w;
}
```

```
// Surcharge operateur - : difference de 2 vecteurs
template <class T>
vector<T> operator - (const vector<T>& v1, const vector<T>& v2){
    assert((v1.size()>0)&&(v1.size()==v2.size()));
    vector<T> w(v1.size(),0);
    for (int i=0; i<v1.size(); i++)
         w[i]= v1[i]-v2[i];
    return w;
}

// Surcharge operateur * : produit scalaire
template <class T>
double operator * (const vector<T>& v1, const vector<T>& v2){
    assert((v1.size()>0)&&(v1.size()==v2.size()));
    double ps=0.0;
    for (int i=0; i<v1.size(); i++)
         ps += v1[i]*v2[i];
    return ps;
}

// Surcharge operateur * : produit vecteur x cte
template <class T>
vector<T> operator * (const vector<T>& v, const T& cte){
    assert(v.size()>0);
    vector<T> w(v.size(),0);
    for (int i=0; i<v.size(); i++)
         w[i]= v[i]*cte;
    return w;
}

// Norme_inf
template <class T>
T norme_inf(const vector<T> & vect){
    T nrm=-1;
    assert(vect.size()>0); // verifier si vecteur non vide
    for (int i=0; i<vect.size(); i++)
        if (abs(vect[i])>nrm) nrm = abs(vect[i]);
    return nrm;
}
```

```
// Norme_1
template <class T>
T norme_1(const vector<T> & vect){
    T nrm=0.0;
    assert(vect.size()>0);
    for (int i=0; i<vect.size(); i++)
        nrm += abs(vect[i]);
    return nrm;
}

// Norme_2
template <class T>
double norme_2(const vector<T> & vect){
    assert(vect.size()>0);
    return sqrt(vect*vect);
}

// Norme_l2
template <class T>
double norme_l2(const vector<T> & vect){
    assert(vect.size()>0);
    return sqrt(vect*vect/vect.size());
}
//extraire depuis les elements du vecteur depuis start-> fin
template <class T>
vector<T>  colonne_start_end(const vector<T> & vect,int start,int fin){
    int length=fin-start;
    vector<T> w(length,0);
    for (int i=0; i<length; i++)
         w[i]= vect[i+start];
    return w;
}
//remplir la vecteur avec des zeros au debut
template <class T>
vector<T>  vector_zeros( vector<T> & vect,int taille){
    int length=taille-vect.size();

    vector<T> w(taille,0);

    if(length!=0){
```

```
    for (int i=0; i<vect.size(); i++)
         w[i+vect.size()]= vect[i];
    return w;
       }
       else{
        return vect;
       }
}
```

## A.2 Code bidiagonalisation de Golub kahan et QR tridiagonalisation

Remarque : les classes matrice.h , matrice.cpp , o.h et o.cpp ce sont des programmes supplémentaires et va prend des grands nombres de page, mais ils sont téléchargé avec le pdf comme une fichier zip, vous pouvez les ignorer et regarder directement mon programme main.cpp.

### A.2.1 main.cpp

```
#include <iostream>
#include <vector>
#include "matrice.h"
#include <string>
#include "cmath"
#include <vector>
#include <fstream>
#include <math.h>

#include "o.h"

using namespace std;
int main()
{
// ouvrire ici le crochet seulement si vous voulez essayer l'exemple analyse textuelle

/*matrice A(4601,58);
ifstream monspam;
monspam.open("C:/Users/khalil al sayed/Desktop/spam9.txt", ios::in);
for (int i=0;i<4601;i++){
    for (int j=0;j<58;j++)
```

```
    {
        monspam>>A[i][j];
    }
}
int m=4601;
int n=58;*/
//-------------------------------------------------------------------------

// ouvrire ici le crochet seulement si vous voulez essayer la decomposition svd pour d

/*int m,n;
cout<<"donner le nombre des lignes m :"<<endl;
cin>>m;
cout<<"donner le nombre des colonnes n :"<<endl;
cin>>n;
matrice A(m,n); // remplissage de la matrice A
for (int i=0;i<m;i++)
{
    for (int j=0;j<n;j++)
    {
        cout<<"donner le coeficient A["<<i+1<<"]["<<j+1<<"] :"<<endl;
        cin>>A[i][j];
    }
}
cout<<"la matrice A est:"<<A<<endl;*/
//-------------------------------------------------------------------------

// Bidiagonalisation de Golub-kahan
matrice QQ(n,100);
matrice PP(m,100);
QQ(0,1)=1;

vector<double> beta(100,0);
vector<double> alpha(100,0);

int o=0;
for (int k=1;k<n+1;k++)
  {     o+=1;

        PP.egale(k,differencevecteur(A*QQ.colonne(k),produitcons(PP.colonne(k-1),beta[
```

```
        alpha[k]= sqrt(produitvecteur(PP.colonne(k),PP.colonne(k)));
        PP.egale(k,produitcons(PP.colonne(k),1/alpha[k]));
        QQ.egale(k+1,differencevecteur(A.transpose()*PP.colonne(k),produitcons(QQ.colo
        beta[k]=sqrt(produitvecteur(QQ.colonne(k+1),QQ.colonne(k+1)));

        if (beta[k]<pow(10,-4))
        {
            break;
        }

        QQ.egale(k+1,produitcons(QQ.colonne(k+1),1/beta[k]));


}
matrice B(o,o);
matrice Q(n,o);
matrice P(m,o);
for (int i=0; i<o ;i++)
{
    P.egale(i,PP.colonne(i+1));
    Q.egale(i,QQ.colonne(i+1));
}

for (int i=0; i<o; i++)
{
    B(i,i)=alpha[i+1];
    if (i<o-1)
    {
      B(i,i+1)=beta[i+1];
    }

}
matrice T;
matrice BT;
BT=B.transpose();
matrice TT;
T=B.transpose()*B;
TT=B*BT;
//-------------------------------------------------------------------------
```

```
// Algorithme QR (pour chercher la matrice V)
vector <double> a(n,0);
vector <double> b(n-1,0);
a[n-1]=T(n-1,n-1);
for (int i=0;i<n-1;i++)
{
    a[i]=T(i,i);
    b[i]=T(i,i+1);
}
int M=n-1;
double d;
double s;
double c;
double x;
double y;
double w;
double z;
double f;
double g;
double L;
matrice QQQ(n,n);
for (int i=0;i<n;i++)
{
    QQQ(i,i)=1;
}
while (M>0) {
        d= (a[M-1]-a[M])/2; // calcule de shift de Wilkinson
        if (d==0)
        {
            s= a[M]-abs(b[M-1]);
        }
        else {
            s=a[M]-(pow(b[M-1],2)/(d+sign(d)*sqrt(pow(d,2)+pow(b[M-1],2))));}
        x= a[0]-s; // QR step commence ici
        y= b[0];
        for (int k=0; k<M;k++)
        {
            if (M>1)
            {
```

```
            givens(x,y,c,s);
        }
        else{
            c=1;
            s=((a[0]-a[1])-sqrt(pow((a[0]-a[1]),2)+4*pow(b[0],2)))/(2*b[0]);
        }
        w=c*x-s*y;
        d=a[k]-a[k+1];
        z=(2*c*b[k]+d*s)*s;
        a[k]=a[k]-z;
        a[k+1]=a[k+1]+z;
        b[k]=d*c*s+(pow(c,2)-pow(s,2))*b[k];
        x=b[k];
        if (k>0)
        {
            b[k-1]=w;
        }
        if (k<M-1)
        {
            y=-s*b[k+1];
            b[k+1]=c*b[k+1];
        }
        for (int i=0;i<n;i++)
        {   f=QQQ(i,k);
            g=QQQ(i,k+1);
            QQQ(i,k)=f*c-g*s;
            QQQ(i,k+1)=f*s+g*c;
        }


    }
    if  (abs(b[M-1])<pow(10,-5)*(abs(a[M-1])+abs(a[M]))) // tester la convergence
    {
        M=M-1;
    }

}
//-----------------------------------------------------------------------------
// Algorithme QR (pour chercher la matrice U)
vector <double> aa(n,0);
```

```
vector <double> bb(n-1,0);
aa[n-1]=TT(n-1,n-1);
for (int i=0;i<n-1;i++)
{
    aa[i]=TT(i,i);
    bb[i]=TT(i,i+1);
}
int MM=n-1;
double dd;
double ss;
double cc;
double xx;
double yy;
double ww;
double zz;
double ff;
double gg;
double LL;
matrice QQQU(n,n);
for (int i=0;i<n;i++)
{
    QQQU(i,i)=1;
}
while (MM>0) {
        d= (aa[MM-1]-aa[MM])/2; // calcule de shift de Wilkinson
        if (dd==0)
        {
            ss= aa[MM]-abs(bb[MM-1]);
        }
        else {
            ss=aa[MM]-(pow(bb[MM-1],2)/(dd+sign(dd)*sqrt(pow(dd,2)+pow(bb[MM-1],2))));
        xx= aa[0]-ss; // QR step commence ici
        yy= bb[0];
        for (int k=0; k<MM;k++)
        {
            if (MM>1)
            {
                givens(xx,yy,cc,ss);
            }
            else{
```

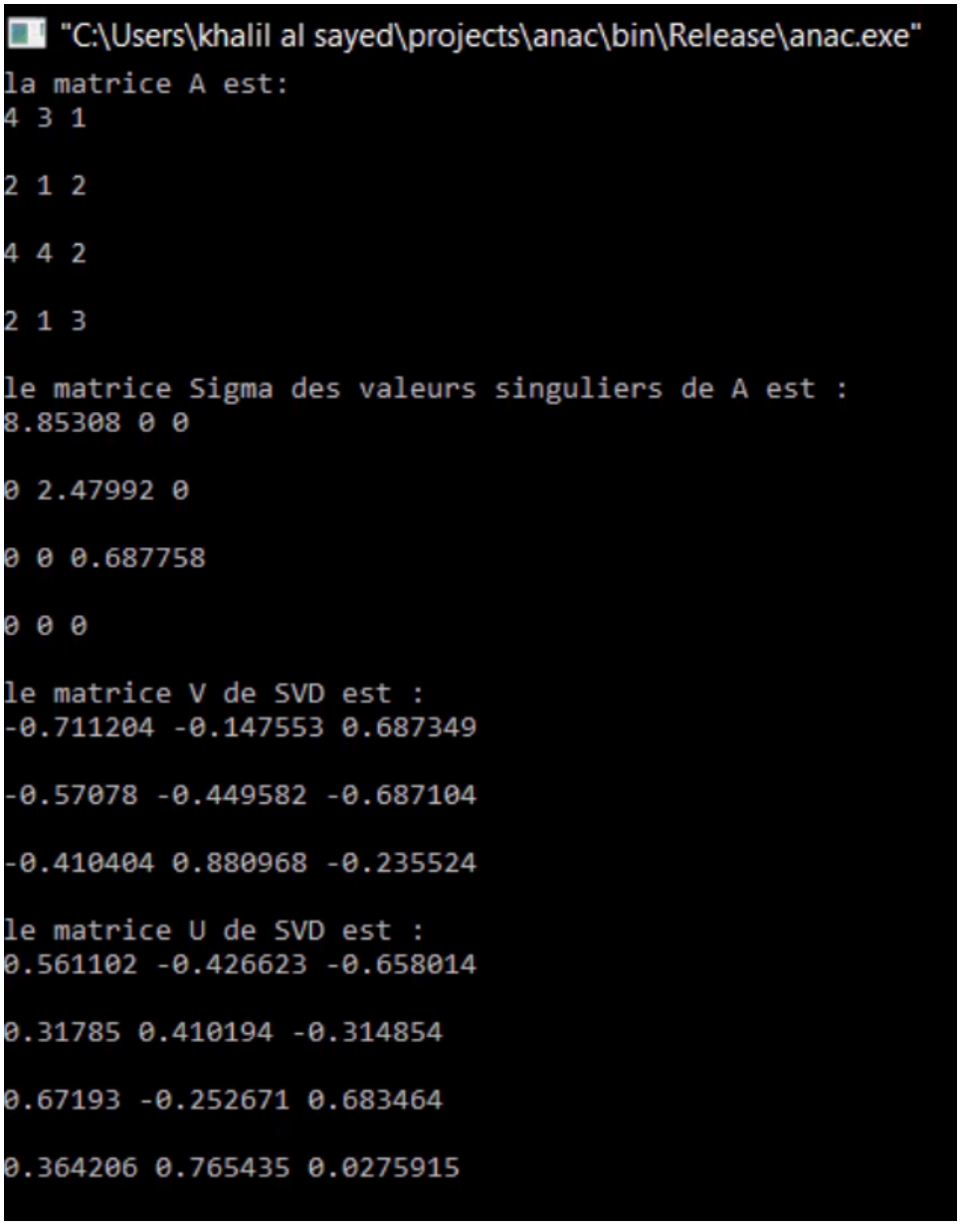
```
                cc=1;
                ss=((aa[0]-aa[1])-sqrt(pow((aa[0]-aa[1]),2)+4*pow(bb[0],2)))/(2*bb[0])
            }
            ww=cc*xx-ss*yy;
            dd=aa[k]-aa[k+1];
            zz=(2*cc*bb[k]+dd*ss)*ss;
            aa[k]=aa[k]-zz;
            aa[k+1]=aa[k+1]+zz;
            bb[k]=dd*cc*ss+(pow(cc,2)-pow(ss,2))*bb[k];
            xx=bb[k];
            if (k>0)
            {
                bb[k-1]=ww;
            }
            if (k<MM-1)
            {
                yy=-ss*bb[k+1];
                bb[k+1]=cc*bb[k+1];
            }
            for (int i=0;i<n;i++)
            {   ff=QQQU(i,k);
                gg=QQQU(i,k+1);
                QQQU(i,k)=ff*cc-gg*ss;
                QQQU(i,k+1)=ff*ss+gg*cc;
            }


        }
        if  (abs(bb[MM-1])<pow(10,-5)*(abs(aa[MM-1])+abs(aa[MM]))) // tester la conver
        {
            MM=MM-1;
        }

}
//-------------------------------------------------------------------------
// la dernière touche pour construire U et V et Sigma
matrice V(n,n);
vector<double> VV;
V=Q*QQQ;
matrice U(m,n);
```

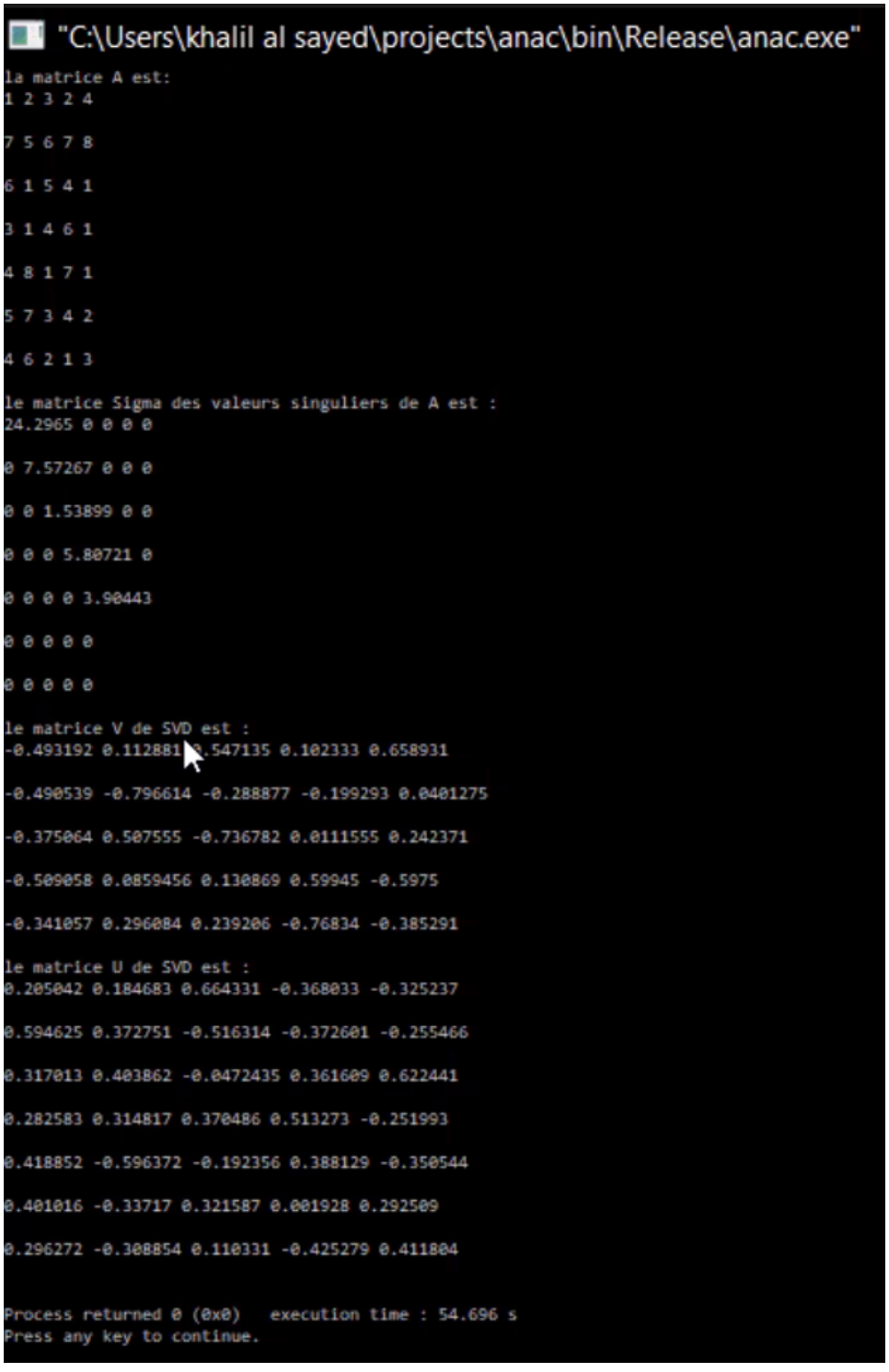
```
vector<double> UU;
U=P*QQQU;
if (a[n-1]>a[n-2]){
L=a[n-2];
a[n-2]=a[n-1];
a[n-1]=L;
VV=V.colonne(n-2);
V.egale(n-2,V.colonne(n-1));
V.egale(n-1,VV);
UU=U.colonne(n-2);
U.egale(n-2,U.colonne(n-1));
U.egale(n-1,UU);}
for (int i=0;i<n;i++)
{
    a[i]=sqrt(a[i]);
}
matrice Sigma(m,n);
for (int i=0;i<n;i++)
{
    Sigma[i][i]=a[i];
}
//-------------------------------------------------------------------------

// ouvrire ici le crochet seulement si vous voulez essayer la decomposition svd pour d

/*cout<<"le matrice Sigma des valeurs singuliers de A est : "<<Sigma<<endl;

cout<<"le matrice V de SVD est : "<<V<<endl;

cout<<"le matrice U de SVD est : "<<U<<endl;*/
//-------------------------------------------------------------------------------

// ouvrire ici le crochet seulement si vous voulez essayer l'exemple analyse textuelle

/*ofstream file1;
file1.open("C:/Users/khalil al sayed/Desktop/aa.txt",ios::in);
for (int i=0;i<n;i++)
{
    file1<<a[i]<<'\t';
}
```

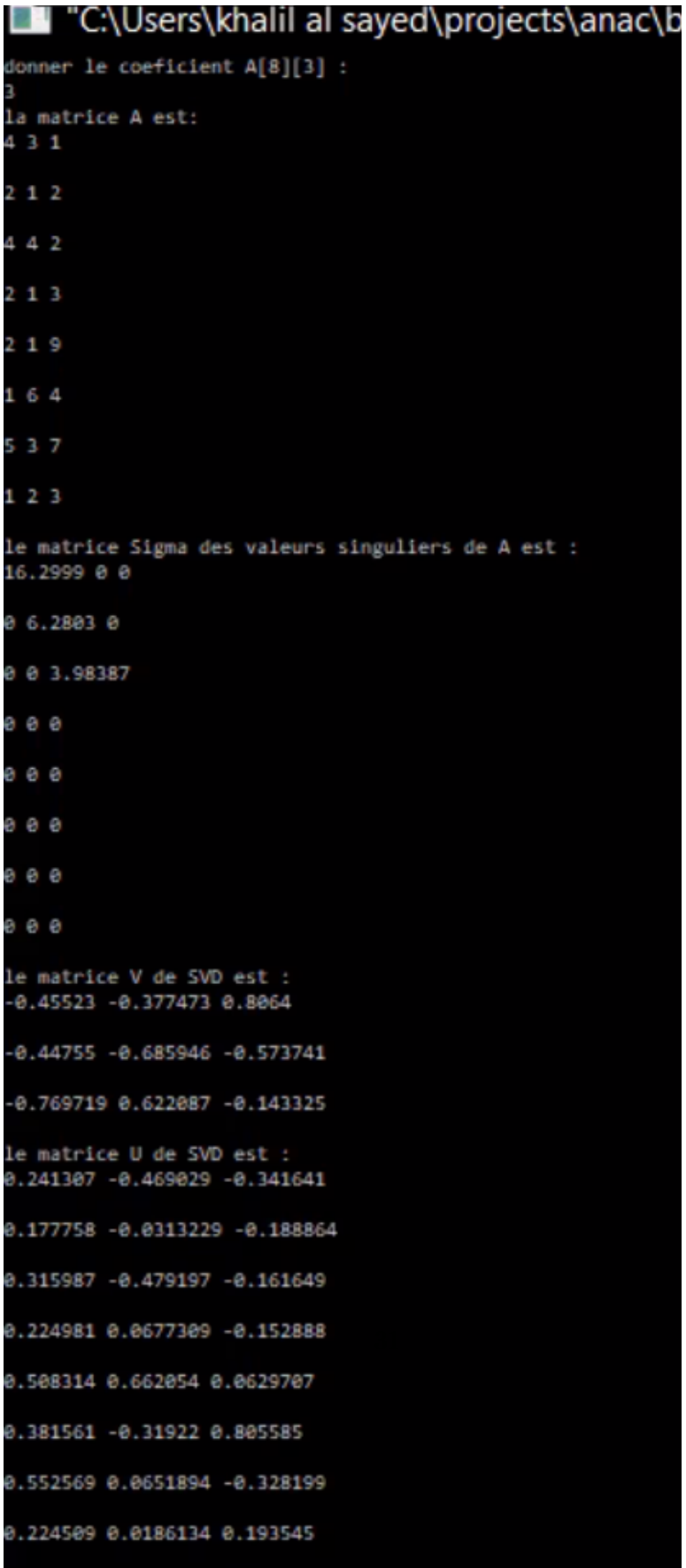
```
ofstream file2;
file2.open("C:/Users/khalil al sayed/Desktop/VV.txt",ios::in);
for (int i=0;i<n;i++)
{
    for (int j=0;j<n;j++)
    {
        file2<<V[i][j]<<'\t';
    }
    file2<<'\n';

}


ofstream file3;
file3.open("C:/Users/khalil al sayed/Desktop/UU.txt",ios::in);
for (int i=0;i<m;i++)
{
    for (int j=0;j<n;j++)
    {
        file3<<U[i][j]<<'\t';
    }
    file3<<'\n';

}

*/
```

**Analyse sur certaines matrices :**

**Remarque :** il faut que la matrice $U$ est de taille $m \times m$ mais moi dans mon programme C++ je prend seulement les $n$ première vecteur $u_i$ .


"C:\Users\khalil al sayed\projects\anac\bin\Release\anac.exe"
la matrice A est:
4 3 1
2 1 2
4 4 2
2 1 3
le matrice Sigma des valeurs singuliers de A est :
8.85308 0 0
0 2.47992 0
0 0 0.687758
0 0 0
le matrice V de SVD est :
-0.711204 -0.147553 0.687349
-0.57078 -0.449582 -0.687104
-0.410404 0.880968 -0.235524
le matrice U de SVD est :
0.561102 -0.426623 -0.658014
0.31785 0.410194 -0.314854
0.67193 -0.252671 0.683464
0.364206 0.765435 0.0275915


"C:\Users\khalil al sayed\projects\anac\bin\Release\anac.exe"
la matrice A est:
1 2 3 2 4
7 5 6 7 8
6 1 5 4 1
3 1 4 6 1
4 8 1 7 1
5 7 3 4 2
4 6 2 1 3
le matrice Sigma des valeurs singuliers de A est :
24.2965 0 0 0 0
0 7.57267 0 0 0
0 0 1.53899 0 0
0 0 0 5.80721 0
0 0 0 0 3.90443
0 0 0 0 0
0 0 0 0 0
le matrice V de SVD est :
-0.493192 0.112881 0.547135 0.102333 0.658931
-0.490539 -0.796614 -0.288877 -0.199293 0.0401275
-0.375064 0.507555 -0.736782 0.0111555 0.242371
-0.509058 0.0859456 0.130869 0.59945 -0.5975
-0.341057 0.296084 0.239206 -0.76834 -0.385291
le matrice U de SVD est :
0.205042 0.184683 0.664331 -0.368033 -0.325237
0.594625 0.372751 -0.516314 -0.372601 -0.255466
0.317013 0.403862 -0.0472435 0.361609 0.622441
0.282583 0.314817 0.370486 0.513273 -0.251993
0.418852 -0.596372 -0.192356 0.388129 -0.350544
0.401016 -0.33717 0.321587 0.001928 0.292509
0.296272 -0.308854 0.110331 -0.425279 0.411804
Process returned 0 (0x0) execution time : 54.696 s
Press any key to continue.


"C:\Users\khalil al sayed\projects\anac\b
donner le coeficient A[8][3] :
3
la matrice A est:
4 3 1
2 1 2
4 4 2
2 1 3
2 1 9
1 6 4
5 3 7
1 2 3
le matrice Sigma des valeurs singuliers de A est :
16.2999 0 0
0 6.2803 0
0 0 3.98387
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
le matrice V de SVD est :
-0.45523 -0.377473 0.8064
-0.44755 -0.685946 -0.573741
-0.769719 0.622087 -0.143325
le matrice U de SVD est :
0.241307 -0.469029 -0.341641
0.177758 -0.0313229 -0.188864
0.315987 -0.479197 -0.161649
0.224981 0.0677309 -0.152888
0.508314 0.662054 0.0629707
0.381561 -0.31922 0.805585
0.552569 0.0651894 -0.328199
0.224509 0.0186134 0.193545

### A.2.2 classes supplémentaire o.h

```
#ifndef O_H_INCLUDED
#define O_H_INCLUDED
#include <iostream>
#include <fstream>
#include <assert.h>
#include <vector>
#include <cmath>
#include <iostream>
#include <ostream>

using namespace std;

// Surcharge operateur + : somme de 2 vecteurs
vector<double> sommevecteur (const vector<double>& , const vector<double>& );
// Surcharge operateur - : difference de 2 vecteurs
vector<double> differencevecteur(const vector<double>& , const vector<double>& );
// Surcharge operateur * : produit scalaire
double produitvecteur (const vector<double>& , const vector<double>& );
// Surcharge operateur * : produit vecteur x cte
vector<double> produitcons (const vector<double>& , const double& );
// Surcharge operator <<
ostream& operator << (ostream& , const vector<double> & );


// Lire un vecteur depuis un fichier
template <class T>
void lire_vect(string nom_fichier, vector<T> & vect){
    ifstream file;
    T val;
    file.open(nom_fichier,ios::in); // ouverture du fichier
    assert(!file.fail()); // verifier ouverture correcte du fichier
    while (!file.eof()){ // on lit tant qu'on est pas à la fin du fichier
        file >> val;
        vect.push_back(val);
    }
    file.close();
    //return vect;
    }
```

```
// Ecrire un vecteur dans un fichier
template <class T>
void ecrire_vect(string nom_fichier, vector<T> & vect){
    ofstream file;
    file.open(nom_fichier,ios::out); // ouverture du fichier
    assert(!file.fail()); // verifier ouverture correcte du fichier
    for (int i=0; i<vect.size(); i++)
        file << " " << vect[i];
    file.close();
}
//-------------------------------------------------------------------------
int fact(int);
//---------------------------------------------------------
double sign(double); // fonction sign
void givens(double ,double ,double &,double &); //fonction givens


#endif // O_H_INCLUDED
```

### A.2.3 classe supplémentaire o.cpp

```
#include <iostream>
#include <ostream>
#if !defined (__CASSERT_H)
#include <cassert>
#endif
#include <iostream>
#include <fstream>
#include <assert.h>
#include <vector>
#include <cmath>
#include <iostream>
#include <ostream>
#include <vector>
#include "o.h"
using namespace std;

// Surcharge operateur + : somme de 2 vecteurs
vector<double> sommevecteur (const vector<double>& v1, const vector<double>& v2){
    assert((v1.size()>0)&&(v1.size()==v2.size()));
```

```
    vector<double> w(v1.size(),0);
    for (int i=0; i<v1.size(); i++)
         w[i]= v1[i]+v2[i];
    return w;
}
// Surcharge operateur - : difference de 2 vecteurs
vector<double> differencevecteur(const vector<double>& v1, const vector<double>& v2){
    assert((v1.size()>0)&&(v1.size()==v2.size()));
    vector<double> w(v1.size(),0);
    for (int i=0; i<v1.size(); i++)
         w[i]= v1[i]-v2[i];
    return w;
}
// Surcharge operateur * : produit scalaire
double produitvecteur (const vector<double>& v1, const vector<double>& v2){
    assert((v1.size()>0)&&(v1.size()==v2.size()));
    double ps=0.0;
    for (int i=0; i<v1.size(); i++)
         ps += v1[i]*v2[i];
    return ps;
}
// Surcharge operateur * : produit vecteur x cte
vector<double> produitcons (const vector<double>& v, const double& cte){
    assert(v.size()>0);
    vector<double> w(v.size(),0);
    for (int i=0; i<v.size(); i++)
         w[i]= v[i]*cte;
    return w;
}
// Surcharge operator <<
ostream& operator << (ostream& s, const vector<double> & vect){
    assert(vect.size()>0);
    cout << " "  << endl;
    for (int i=0; i<vect.size(); i++)
        s << vect[i] << " ";
    s << endl;
    return s;
}

//-------------------------------------------------------
```

```
int fact(int k)
    {
        int c=1;
        for (int i=1; i<=k; i++)
        {
            c=c*i;
        }
        return c;
    }
//-------------------------------------------------------
// la fonction sign
double sign(double d){
if (d==0){
    return 0;
} else if (d>0){
    return 1;
} else {
    return -1;}
}
//-------------------------------------------------------------------
// la fonction de givens
void givens(double a,double b,double &c,double &s ){
double r;
if (b==0)
{
    c=1;
    s=0;
}
else if(abs(b)>abs(a)){
    r=-(a/b);
    s=1/sqrt(1+pow(r,2));
    c=s*r;
}
else {
    r=-(b/a);
    c=1/sqrt(1+pow(r,2));
    s=c*r;
}
```

```
}
```

### A.2.4 classe supplémentaire matrice.h

```
#ifndef MATRICE_H_INCLUDED
#define MATRICE_H_INCLUDED
#if !defined (__VECTOR_H)
#include <vector>
#endif
#if !defined (__FSTREAM_H)
#include <fstream>
#endif

using namespace std;
class matrice
{
protected:
    int size1;
    int size2;
    vector<double>* matrix;
public:
    matrice()
    {
        this->size1 = this->size2 = 0;
    };
    matrice(int,int);
    matrice(const matrice&);
    ~matrice();
    int dim1() const
    {
        return this->size1;   // retourne la 1ere dimension de la matrice
    }
    int dim2() const
    {
        return this->size2;   // retourne la 2eme dimension de la matrice
    }
    vector<double>& operator [] (int) const;  // retourne une ligne complete
    double& operator () (int,int) const;      // retourne un coefficient
    matrice& operator = (const matrice&); // surcharge de = (affectation)
```

```
    void  egale(const int , const vector<double>&); // Surcharge = (affectation) vecte
    matrice transpose();    // transpose une matrice
    vector<double> colonne(const int);    // une colonne de la matrice
    matrice operator+(const matrice&); // somme de 2 matrices
    matrice operator-(const matrice&); // difference de 2 matrices
    matrice operator*(matrice&); // produit de 2 matrices, pas const matrice& à cause
    matrice operator*(const double&); // matrice x cte
    vector<double> operator*(const vector<double>&); // produit A*v
    friend vector<double> operator*(const vector<double>&, matrice&); // produit v^t *
    friend ostream& operator << (ostream&, const matrice &);        // surcharge << po
    friend istream& operator >> (istream&, const matrice &);        // surcharge >> po
    void save_matr(const char*); // ecriture dans un fichier
    void read_matr(const char *);  // lecture dans un fichier
};



#endif // MATRICE_H_INCLUDED
```

### A.2.5 classe supplémentaire matrice.cpp

```
#include "matrice.h"
#include <iostream>
#include <ostream>
#if !defined (__CASSERT_H)
#include <cassert>
#endif
#include "o.h"

using namespace std;
// Constructeur v1
matrice::matrice(int n, int m)
{
    assert((n>0)&&(m>0));
    this->size1 = n;
    this->size2 = m;
    this->matrix = new vector<double>[n];
    vector<double> v(m,0);
    for (int i=0; i<n ; i++)
        this->matrix[i]=v;
```

```
}

//-------------------------------------------------------
// Constructeur par copie
matrice::matrice(const matrice & mat)
{
  assert((mat.size1)&&(mat.size2));
  this->size1=mat.size1;
  this->size2=mat.size2;
  this->matrix=new vector<double>[size1];
  for (int i=0; i<mat.size1 ; i++)
      this->matrix[i]=mat.matrix[i];
}
//-------------------------------------------------------
// Destructeur
matrice::~matrice()
{
  if (this->size1||this->size2) delete[] this->matrix;
  this->size1=0;
  this->size2=0;
}
// Retourne la ligne i
vector<double>& matrice::operator [] (int i) const
{
  assert((i>=0)&&(i<this->size1));
  return this->matrix[i];
}
// Retourne le coefficient (i,j)
double& matrice::operator () (int i, int j) const
{
  assert((i>=0)&&(i<this->size1)&&(j>=0)&&(j<this->size2));
  return this->matrix[i][j];
}
// Surcharge = (affectation)
matrice& matrice::operator = (const matrice& mat)
{
  assert((mat.size1>0)&&(mat.size2>0));
  if ((!this->size1)||(!this->size2))
    {this->size1=mat.size1;
     this->size2=mat.size2;
```

```
     this->matrix=new vector<double>[size1];}
  assert((this->size1==mat.size1)&&(this->size2==mat.size2));
  for (int i=0; i<this->size1 ; i++)
    this->matrix[i]=mat.matrix[i];
  return (*this);
  }
  // Surcharge = (affectation) vecteur
void matrice::egale(const int k, const vector<double>& mat)
{



  for (int i=0; i<this->size1 ; i++)
    this->matrix[i][k]=mat[i];

  }
  // Transposee d'une matrice
matrice matrice::transpose()
{
    matrice T(this->size2,this->size1);
    for (int i=0; i<this->size2; i++)
        for (int j=0; j<this->size1; j++)
            T(i,j) = matrix[j][i];
    return T;
}
//---------------------------------------------------------
// Retourner la colonne j d'une matrice
vector<double> matrice::colonne (const int j)
{
    assert((j>=0)&&(j<this->size2));
    vector<double> vcol(this->size1);
    for (int i=0; i<this->size1; i++)
        vcol[i] = this->matrix[i][j];
    return vcol;
}
//---------------------------------------------------------

// Surcharge operateur + non symetrique : somme de 2 matrices
matrice matrice::operator + (const matrice& A)
{
```

```
        assert((this->size1>0)&&(this->size1==A.size1));
        assert((this->size2>0)&&(this->size2==A.size2));
        matrice C(this->size1,this->size2);
        for (int i=0; i<this->size1; i++)
    {
       C[i] = sommevecteur(this->matrix[i],A[i]);
    }

        return C;
}
//-------------------------------------------------------

// Surcharge operateur - non symetrique : difference de 2 matrices
matrice matrice::operator - (const matrice& A)
{
    assert((this->size1>0)&&(this->size1==A.size1));
    assert((this->size2>0)&&(this->size2==A.size2));
    matrice C(this->size1,this->size2);
    for (int i=0; i<this->size1; i++)
        C[i] = differencevecteur(this->matrix[i],A[i]);
    return C;
}
//-------------------------------------------------------
// Surcharge operateur * non symetrique : produit de 2 matrices
matrice matrice::operator * (matrice& A) // pas const A à cause de la fonction transpo
{
    assert((this->size1>0)&&(this->size2>0)&&(A.size1>0)&&(A.size2>0));
    assert(this->size2==A.size1);
    matrice TA=A.transpose();
    matrice C(this->size1,A.size2);
    for (int i=0; i<this->size1; i++)
         for (int j=0; j<A.size2; j++)
             C(i,j) = produitvecteur(this->matrix[i],TA[j]);
    return C;
}
//-------------------------------------------------------
// Surcharge operateur * non symetrique : matrice x cte
matrice matrice::operator * (const double& cte)
{
        assert((this->size1>0)&&(this->size2>0));
```

```
        matrice C(this->size1,this->size2);
        for (int i=0; i<this->size1; i++)
             C[i] = produitcons(this->matrix[i],cte);
        return C;
}
//-------------------------------------------------------
// Surcharge operateur * non symetrique : produit A*v
vector<double> matrice::operator * (const vector<double>& v)
{
    assert((this->size1>0)&&(this->size2>0)&&(this->size2==v.size()));
    vector<double> w(this->size1);
    for (int i=0; i<this->size1; i++)
        w[i] = produitvecteur(this->matrix[i],v);
    return w;
}
//-------------------------------------------------------
// Surcharge operateur * symetrique : produit v^t * A
vector<double> operator * (const vector<double>& v, matrice& A) // pas const A à cause
{
        assert((A.size1>0)&&(A.size2>0)&&(A.size1==v.size()));
        vector<double> w(A.size2);
        matrice TA=A.transpose();
        for (int i=0; i<A.size2; i++)
             w[i] = produitvecteur(v,TA[i]);
        return w;
}
//-------------------------------------------------------
// Surcharge operator << pour une matrice
ostream& operator << (ostream& s, const matrice & mat)
{
    assert((mat.size1>0)&&(mat.size2>0));

    for (int i=0; i<mat.size1; i++)
            s << mat[i];
    return s;
}
//-------------------------------------------------------
// Surcharge operator >> pour une matrice
istream& operator >> (istream& s, const matrice & mat)
{
```

```
    assert((mat.size1>0)&&(mat.size2>0));
    cout << "Entrer au clavier une matrice avec " << mat.size1 <<" lignes et " << mat.s
    for (int i=0; i<mat.size1; i++)
        for (int j=0; j<mat.size2; j++)
            s>>mat(i,j);
    return s;
}
//--------------------------------------------
// Sauvegarde d'une matrice dans un fichier
void matrice :: save_matr(const char *Nomfich)
{
    ofstream fichier;
    fichier.open(Nomfich); // ouverture du fichier

    assert(this->size1>0 && this->size2>0);
    fichier  << this->size1 << " " << this->size2 << "\n"<<"\n"<<"\n";
    for (int i=0; i<this->size1 ; i++)
        fichier << this->matrix[i];
    fichier.close();           // fermature du fichier
}
//--------------------------------------------
// lecture d'une matrice dans un fichier
// on va obtenir une matrice de la bonne taille
void matrice :: read_matr(const char *Nomfich)
{
    ifstream fichier;
    fichier.open(Nomfich); // ouverture du fichier
    if (!fichier){
        cerr << "Probleme d'ouverture de fichier " << endl;
        exit(1);
    }

    int m,n;
    if (!fichier.eof())
      fichier >> m >> n;
    this->size1=m;
    this->size2=n;

    for (int i=0;i<this->size1;i++){
        for (int j=0;j<this->size2;j++){
```

```
            fichier >> this->matrix[i][j];
            if (fichier.eof())
                cout << " Erreur dans le fichier, matrice incomplete \n ";
        }
    }
    fichier.close();   // fermeture du fichier

}
// Surcharge operateur *  remarque : vous pouvez ignorez cette fonction et utiliser di
double prodv (const vector<double>& v1, const vector<double>& v2){
    assert((v1.size()>0)&&(v1.size()==v2.size()));
    double ps=0.0;
    for (int i=0; i<v1.size(); i++)
         ps += v1[i]*v2[i];
    return ps;
}
```

### A.3 Code : Méthode de puissance inverse avec shift

```
from math import sqrt
from math import pi
import matplotlib.pyplot as plt

import numpy as np
from scipy import linalg as sl
from math import *


def puissanceinverse (A, mu,x0 ) :

    size=A.shape[0]
    e=0.0001
    X=x0
    i=0
    val=0
    prev_val=1
    while (abs(val-prev_val)>e ):
        prev_val=val

        V=X/np.linalg.norm(X,1)
```

```
        X=sl.solve((A-mu*np.identity(size)),V)
        val=1./float(np.dot(X.T,V))
        i+=1
    return V,val+mu

M3 =np.array(((
(40 ,38.833, 0),
(38.833, 39.0504, 1.52785),
(0, 1.52785, 5.9496),

)))


mu=3.96
#q0=np.array((((1,0,0,0))))

q0=np.array((((1,0,0))))
V1,val1=puissanceinverse(M3,mu,q0)
print "La première valeur propre egale à %f pour mu=%f"%(val1,mu)

mu=3.97
V2,val2=puissanceinverse(M3,mu,q0)
print "La deuxième valeur propre egale à %f pour mu=%f"%(val2,mu)
mu=3.975
V3,val3=puissanceinverse(M3,mu,q0)
print "La troisième valeur propre egale à %f pour mu=%f"%(val3,mu)
e,f=sl.eig(M3)
print "Les valeurs propre par sl.eig=%s"%e
print "La valeur propre obtenue %f est proche à la valeur \
propre calculé par sl.eig %f"%(val1,e[2])
```