



RAPPORT DE PROJET CALCUL SCIENTIFIQUE

AINJI JOMA'A & QUENTIN POINTEAU

AVRIL 2024

Partie 1

Réponse question 1 :

Voici les temps d'exécutions obtenus pour la fonction **eig** de MATLAB et la fonction **power_v11**.

Taille	Temps eig	Temps power_v11
50	0	0.01
100	0.05	0.11
150	0	1.32
200	0.01	1.88
250	0.02	3.78
300	0.04	5.39
350	0.05	11.11
400	0.03	14.32
500	0.06	34.66

TABLE 1 - Comparaison des temps d'exécution entre eig et power_v11.

On remarque que le temps de calcul avec la fonction **eig** de MATLAB reste à peu près constant (même ordre de grandeur) alors que le temps de calcul avec **power_v11** devient de plus en plus grand quand la taille de la matrice augmente.

Réponse question 2 :

Voici l'algorithme **power_v12** proposé en comparaison avec l'algorithme fourni dans **power_v11**.

```

1  v = randn(n,1);
2  z = A*v;
3  beta = v'*z;
4  norme = norm(beta*v - z, 2) / norm(beta,2);
5  nb_it = 1;
6
7  % power_v11
8  while(norme > eps && nb_it < maxit)
9      y = A*v;
10     v = y / norm(y,2);
11     z = A*v;
12     beta = v'*z;
13     norme = norm(beta*v - z,2) / norm(beta,2);
14     nb_it = nb_it + 1;
15 end
16
17 % power_v12
18 while(norme > eps && nb_it < maxit)
19     v = z / norm(z,2);
20     z = A*v;
21     beta = v'*z;
22     norme = norm(beta*v - z,2) / norm(beta,2);
23     nb_it = nb_it + 1;
24 end

```

On remarque que le calcul de y (ligne 9) est supprimé pour laisser place directement au calcul de v (ligne 19) grâce à l'initialisation de la variable z (ligne 2). Ainsi, on passe de deux à un calcul matriciel à l'intérieur de la boucle.

Vector power method

Input: Matrix $A \in \mathbb{R}^{n \times n}$

Output: (λ_1, v_1) eigenpair associated to the largest (in module) eigenvalue.

$v \in \mathbb{R}^n$ given

$z = A \cdot v$

$\beta = v^T \cdot A \cdot v$

repeat

$z = z / \|z\|$

$\beta_{old} = \beta$

$\beta = v^T \cdot z$

until $|\beta - \beta_{old}| / |\beta_{old}| < \varepsilon$

$\lambda_1 = \beta$ and $v_1 = v$

Voici les temps d'exécutions obtenus pour la fonction **eig** de MATLAB et la fonction **power_v12**.

Taille	Temps eig	Temps power_v12
50	0.04	0.01
100	0	0.03
150	0	0.56
200	0.01	1.03
250	0.02	2.06
300	0.02	3.49
350	0.03	6.04
400	0.05	7.56
500	0.06	18.41

TABLE 2 - Comparaison des temps d'exécution entre eig et power_v11.

On remarque que pour la question 1, le temps de calcul pour **eig** reste quasiment constant alors que celui pour **power_v11** continue d'augmenter. Le constat reste le même pour **power_v12** mais il est à noter que le temps de calcul est divisé par 2 par rapport à l'algorithme proposé dans **power_v11**. Étant passé de deux à un produit matriciel, on en conclut que le produit matriciel est l'opération qui prend le plus de temps.

Réponse question 3 :

En termes de temps de calcul, les principaux désavantages de la *power method* sont ceux d'un produit matriciel classique. En effet, le produit matriciel est une opération lourde en temps de calcul mais aussi en termes d'espace mémoire si la matrice est grande. De plus, dans la partie théorique on fait tendre $p \rightarrow +\infty$ donc il faut aussi un nombre très élevé d'itérations.

Réponse question 4 :

L'une des différences entre la *power method* et la *subspace iteration method* v0 est la matrice A prise en entrée. En effet, dans la *subspace iteration method* v0, on fait l'hypothèse que la matrice $A \in \mathbb{R}^{n \times n}$ est symétrique. On peut donc se permettre d'orthonormaliser les vecteurs propres car on sait d'après le théorème spectral qu'une matrice symétrique à coefficients réels est diagonalisable dans une base orthonormée de vecteurs propres. Cependant, ce n'est pas le cas dans la *power method* car l'on ne suppose pas que la matrice A est symétrique. Ainsi, vu qu'on n'orthonormalise pas les vecteurs propres dans la *power method*, on n'a aucune garantie d'obtenir une base.

Dans ce cas, la matrice A va converger vers une matrice dont toutes les colonnes sont des vecteurs propres associés à la valeur propre la plus grande.

Réponse question 5 :

La matrice H est définie par le calcul matriciel suivant :

$$H = V^T \cdot A \cdot V$$

Or $V \in \mathbb{R}^{n \times m}$ donc $V^T \in \mathbb{R}^{m \times n}$ et $A \in \mathbb{R}^{n \times n}$. Ainsi, on a $H \in \mathbb{R}^{m \times m}$. En outre, comme la valeur de m est généralement très petite puisque qu'elle correspond au nombre de composantes principales que nous souhaitons

avoir pour réaliser une ACP, ce n'est donc pas un problème de calculer l'intégrale décomposition spectrale de H .

Réponse question 6 :

C.f. code dans le fichier `subspace_iter_v0.m`.

Réponse question 7 :

```
1 function [ V, D, n_ev, it, itv, flag ] = subspace_iter_v1( A, m, percentage, eps, maxit )
2
3     % calcul de la norme de A (pour le critère de convergence d'un vecteur (gamma))
4     normA = norm(A, 'fro');
5
6     % trace de A
7     traceA = trace(A);
8
9     % valeur correspondant au pourcentage de la trace à atteindre
10    vtrace = percentage*traceA;
11
12    n = size(A,1);
13    W = zeros(m,1);
14    itv = zeros(m,1);
15
16    % numéro de l'itération courante
17    k = 0;
18    % somme courante des valeurs propres
19    eigsum = 0.0;
20    % nombre de vecteurs ayant convergés
21    nb_c = 0;
22
23    % indicateur de la convergence
24    conv = 0;
25
26    % on génère un ensemble initial de m vecteurs orthogonaux
27    Vr = randn(n, m);
28    Vr = mgs(Vr);
29
30    % rappel : conv = (eigsum >= trace) / (nb_c == m)
31    while (~conv && k < maxit)
32
33        k = k+1;
34        %% Y <- A*V
35        Y = A*Vr;
36        %% orthogonalisation
37        Vr = mgs(Y);
38
39        %% Projection de Rayleigh-Ritz
40        [Wr, Vr] = rayleigh_ritz_projection(A, Vr);
41
42        %% Quels vecteurs ont convergé à cette itération
43        analyse_cvg_finie = 0;
44        % nombre de vecteurs ayant convergé à cette itération
45        nbk_k = 0;
46        % nb_c est le dernier vecteur à avoir convergé à l'itération précédente
47        i = nb_c + 1;
48
49        while(~analyse_cvg_finie)
50            % tous les vecteurs de notre sous-espace ont convergé
```

```

51     % on a fini (sans avoir obtenu le pourcentage)
52     if(i > m)
53         analyse_cvg_finie = 1;
54     else
55         % est-ce que le vecteur i a convergé
56
57         % calcul de la norme du résidu
58         aux = A*Vr(:,i) - Wr(i)*Vr(:,i);
59         res = sqrt(aux'*aux);
60
61         if(res >= eps*normA)
62             % le vecteur i n'a pas convergé,
63             % on sait que les vecteurs suivants n'auront pas convergé non plus
64             % => itération finie
65             analyse_cvg_finie = 1;
66         else
67             % le vecteur i a convergé
68             % un de plus
69             nbc_k = nbc_k + 1;
70             % on le stocke ainsi que sa valeur propre
71             W(i) = Wr(i);
72
73             itv(i) = k;
74
75             % on met à jour la somme des valeurs propres
76             eigsum = eigsum + W(i);
77
78             % si cette valeur propre permet d'atteindre le pourcentage
79             % on a fini
80             if(eigsum >= vtrace)
81                 analyse_cvg_finie = 1;
82             else
83                 % on passe au vecteur suivant
84                 i = i + 1;
85             end
86         end
87     end
88 end
89
90 % on met à jour le nombre de vecteurs ayant convergés
91 nb_c = nb_c + nbc_k;
92
93 % on a convergé dans l'un de ces deux cas
94 conv = (nb_c == m) | (eigsum >= vtrace);
95
96 end
97
98 if(conv)
99     % mise à jour des résultats
100     n_ev = nb_c;
101     V = Vr(:, 1:n_ev);
102     W = W(1:n_ev);
103     D = diag(W);
104     it = k;
105 else
106     % on n'a pas convergé
107     D = zeros(1,1);
108     V = zeros(1,1);
109     n_ev = 0;
110     it = k;
111 end

```

```

112
113     % on indique comment on a fini
114     if(eigsum >= vtrace)
115         flag = 0;
116     else if (n_ev == m)
117         flag = 1;
118     else
119         flag = -3;
120     end
121 end
122 end

```

La première étape consistant à générer aléatoirement un ensemble de m vecteurs propres orthonormés $V \in \mathbb{R}^{n \times m}$ correspond aux lignes 27 et 28.

$k = k + 1$ se trouve à la ligne 33.

Le calcul de Y tel que $Y = A \cdot V$ se trouve à la ligne 35.

L'orthonormalisation des colonnes de Y se trouve à la ligne 37.

La projection de Rayleigh-Ritz appliqué à la matrice A et aux vecteurs orthonormés V correspond à l'appel de la fonction ligne 40. Fonction dont le code est le suivant :

```

1  function [ W, V ] = rayleigh_ritz_projection( A, V )
2
3  H = V'*(A*V);
4
5  [VH, DH] = eig(H);
6  [W, indice] = sort(diag(DH), 'descend');
7
8  V = V*VH(:, indice);
9
10 end

```

Le stockage des couples propres qui ont convergé ainsi que la mise à jour de la somme des vecteurs propres correspond aux lignes 42 à 91.

Réponse question 8 :

A est une matrice carré de taille $n \times n$:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

Le coût de la multiplication matricielle de A par elle-même est donc égal à :

$$(2n - 1)(n^2) \sim n^3$$

En effet, pour chaque multiplication d'une ligne par une colonne on fait n multiplications et $n - 1$ additions, donc un total de $2n - 1$ flops. Le produit ligne-colonne est répété $n \times n$ fois pour faire le calcul de tous les éléments de la matrice. Par conséquent, le nombre total de flops est de $(2n - 1)(n^2) \sim n^3$.

Ceci représente le coût d'un seul produit de A par elle-même. Par suite, le produit de A par elle-même p fois (i.e le calcul de A^p) coûtera :

$$(2n - 1)^p (n^{2p}) \sim n^{3p}$$

V est quant à elle une matrice de taille $n \times m$:

$$V = \begin{bmatrix} v_{11} & v_{12} & \cdots & v_{1m} \\ v_{21} & v_{22} & \cdots & v_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ v_{n1} & v_{n2} & \cdots & v_{nm} \end{bmatrix}$$

Jusqu'à présent, lorsqu'on voulait calculer $A^p \cdot V$, on commençait par déterminer A^p puis au final on calculait $A^p \times V$. Ceci coûte à peu près n^{3p} flops (comme expliqué plus haut).

Afin de réduire le coût de calcul de $A^p \cdot V$ on pourrait commencer par calculer $A \cdot V = A \cdot V$ qui aurait un coût de $2n^2m$ flops. Puis multiplier $A \cdot V$ par A , p fois.

En utilisant cette autre méthode méthode, le coût total de calcul de $A^p \cdot V$ serait de l'ordre de :

$$2n^{2p} \times m^p$$

Ceci réduit considérablement le prix de ce produit matriciel, et particulièrement lorsque m devient petit devant n .

Réponse question 9 :

C.f. code dans le fichier `subspace_iter_v2.m`.

Réponse question 10 :

Bien que l'augmentation de la valeur de p permet une convergence plus rapide vers les vecteurs propres recherchés, si cette dernière est beaucoup trop élevée, ceci peut conduire à des résultats erronés.

En effet, lorsque la valeur de p dépasse un certain seuil (que nous ne connaissons pas à l'avance), l'algorithme risque de converger vers une matrice qui ne contient que les vecteurs propres associés à une seule et unique valeur propre. Ceci n'est évidemment pas ce que l'on recherche.

Par conséquent, la bonne pratique serait de trouver un compromis entre l'accélération du temps de calcul et la précision des résultats que l'on obtient.

Réponse question 11 :

Nous avons sur la figure 1 une image contenant le vecteur qv qui représente la qualité de chacun des couples (valeur propre, vecteur propre).

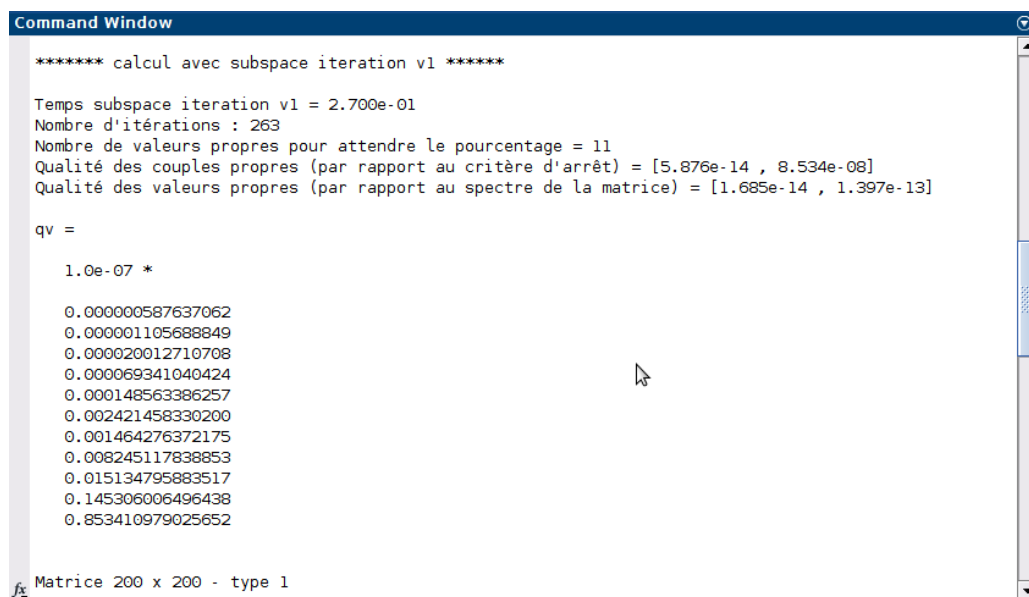


Figure 1: Les erreurs associées à l'algorithme v1.

On remarque que l'erreur de calcul du vecteur propre associée à la valeur propre la plus grande (*i.e.* la première composante du vecteur qv) est beaucoup plus petite ($\sim 10^{-14}$) que celle associée au dernier vecteur propre ($\sim 10^{-8}$).

Soit λ_1 la valeur propre la plus grande et v_1 son vecteur propre associé. On note \tilde{v}_1 la valeur expérimentale du vecteur propre v_1 calculée par l'algorithme.

On sait que v_1 sera le premier vecteur à converger parce qu'il est associé à la valeur propre la plus grande. Or lors des itérations suivantes qui sont dédiées à la convergence des autres vecteurs propres qui n'ont pas encore convergé, celui-ci continue de se rapprocher de sa vraie valeur v_1 .

Ceci est dû au fait que pour cette version de l'algorithme, le vecteur n'est pas exclu des calculs lorsqu'il atteint le seuil de convergence attendu (10^{-8}). L'erreur devient de plus en plus petite pour celui-ci et de même pour les autres dès qu'ils auront convergé au fur et à mesure.

Quant au dernier vecteur, on remarque que la dernière composante du vecteur qv (qui est son erreur associée) est de l'ordre de (10^{-8}) puisque dès que ce vecteur converge l'algorithme s'arrête. Le critère de convergence étant donc une erreur de l'ordre de (10^{-8}) pour tous les vecteurs.

Pour la version 2 de l'algorithme, comme les vecteurs ayant convergé rentrent toujours dans les calculs, on obtient un résultat similaire.

Réponse question 12 :

On rappelle que V est la matrice qui contiendra, après convergence, les vecteurs propres de A .

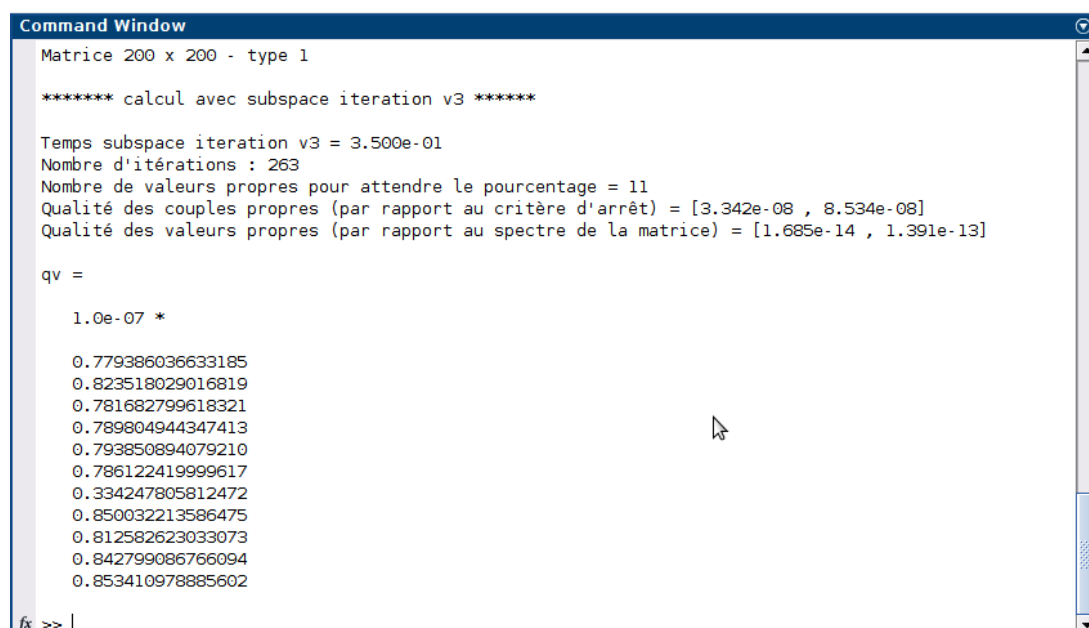
On note nc le nombre de colonnes de V qui a convergé après l'itération i .

Dans la version 3 de l'algorithme, on divise les colonnes de V en 2 parties. Ceci nous donne alors $V = [V_c, V_{nc}]$ où V_c a nc colonnes et V_{nc} en a $m - nc$.

La différence entre cette version et les précédentes est que, à chaque itération, on remplace le produit $A \cdot V$ par un produit de la forme $[V_c, A \cdot V_{nc}]$. En d'autres termes, lorsqu'un vecteur converge, il est figé et ne sera plus multiplié par A lors des itérations suivantes.

Par conséquent, à la fin de l'algorithme, les erreurs seront toutes de l'ordre de (10^{-8}) parce que dès que ce seuil sera atteint, les vecteurs seront figés. Les premiers vecteurs ne pourront donc plus se rapprocher de plus en plus des vraies valeurs durant les itérations suivantes.

Ce résultat est bien confirmé par la figure 2.



```
Command Window
Matrice 200 x 200 - type 1

***** calcul avec subspace iteration v3 *****

Temps subspace iteration v3 = 3.500e-01
Nombre d'itérations : 263
Nombre de valeurs propres pour attendre le pourcentage = 11
Qualité des couples propres (par rapport au critère d'arrêt) = [3.342e-08 , 8.534e-08]
Qualité des valeurs propres (par rapport au spectre de la matrice) = [1.685e-14 , 1.391e-13]

qv =

1.0e-07 *

0.779386036633185
0.823518029016819
0.781682799618321
0.789804944347413
0.793850894079210
0.786122419999617
0.334247805812472
0.850032213586475
0.812582623033073
0.842799086766094
0.853410978885602
```

Figure 2: Les erreurs associées à l'algorithme v3.

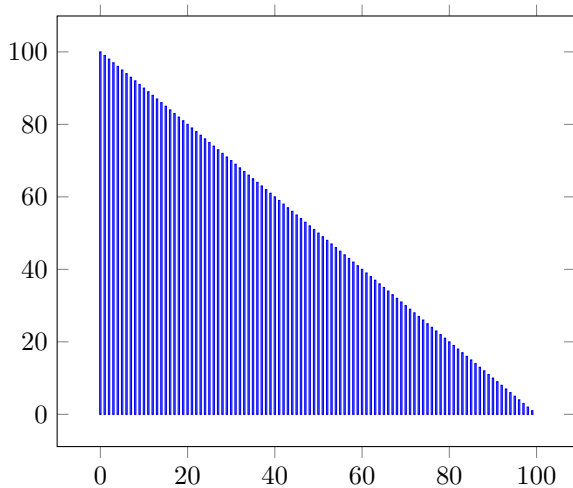
Réponse question 13 :

C.f. code dans le fichier `subspace_iter_v3.m`.

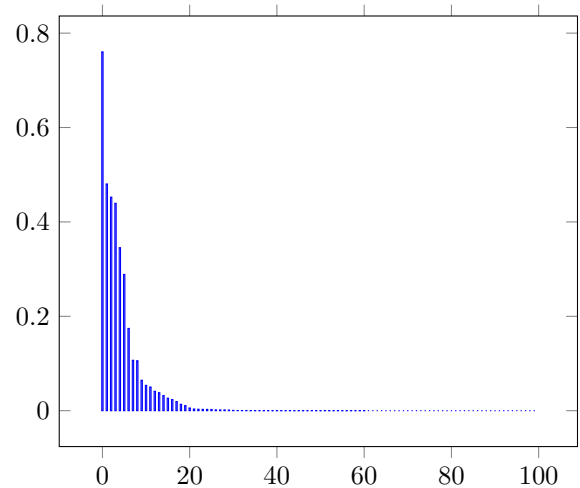
Réponse question 14 :

Dans la figure 3, on peut observer la distribution des valeurs propres pour les différents types de matrices fournies dans le code. Les 4 graphiques ci-dessus ont été tracés à partir des valeurs propres d'une matrice de taille $n = 100$. Les quatre distributions sont les suivantes :

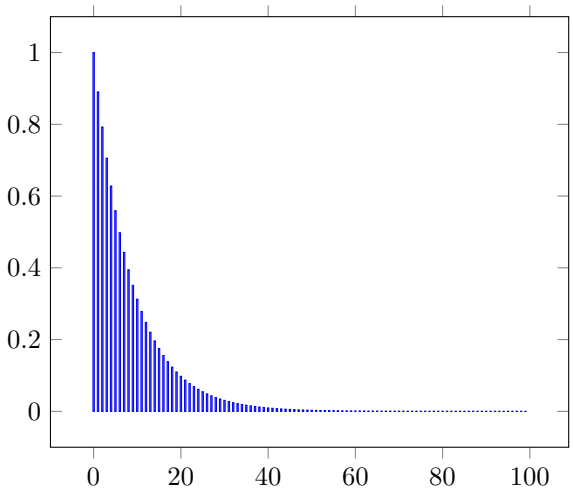
1. Première matrice : Les valeurs propres sont des entiers succesifs allant de 1 à n .
2. Deuxième matrice : Pour cette matrice, on commence par tirer aléatoirement une valeur x entre 0 et 1. Ensuite, on multiplie cette valeur x par $\log(\text{cond}) = \log(10^{-10})$. Finalement, on prend l'exponentielle de cette valeur. En d'autres termes, pour $i \in \{1..n\}$, $\lambda_i = \exp(\log(10^{-10})) * x$. N.B. : Contrairement aux autres matrices, les valeurs propres de la matrice 2 changent à chaque compilation. Par conséquent, le graphique donné par la figure 3(b) donne la forme globale de la répartition, mais les valeurs exactes changeront à chaque fois que le code sera lancé.
3. Troisième matrice : Pour la matrice 3, pour $i \in \{1..n\}$, $\lambda_i = 10^{\frac{-5(i-1)}{(n-1)}}$.
4. Quatrième matrice : Finalement, pour la matrice 4, pour $i \in \{1..n\}$, $\lambda_i = \frac{1-\frac{1}{10^2}}{n-1} * (n-i) + \frac{1}{10^2}$.



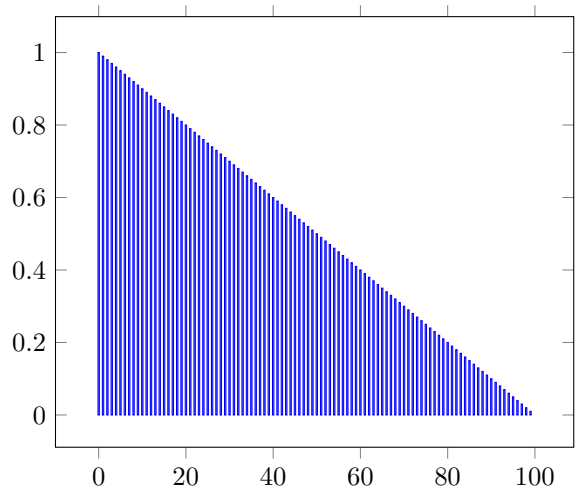
(a) Matrice 1



(b) Matrice 2



(c) Matrice 3



(d) Matrice 4

Figure 3: Répartition des valeurs propres pour les différents types de matrices

Réponse question 15 :

Type de matrice	Taille	Temps v0	Temps v1	Temps v2	Temps v3	Temps eig
1	50	$2, 7.10^{-1}$	3.10^{-2}	1.10^{-2}	4.10^{-2}	3.10^{-2}
2	50	$2, 1.10^{-1}$	3.10^{-2}	\	1.10^{-2}	0
3	50	$1, 5.10^{-1}$	2.10^{-2}	\	1.10^{-2}	0
4	50	7.10^{-1}	1.10^{-1}	1.10^{-2}	$1, 2.10^{-1}$	2.10^{-2}
1	200	6, 95	$4, 1.10^{-1}$	8.10^{-2}	3.10^{-1}	4.10^{-2}
2	200	$4, 9.10^{-1}$	4.10^{-2}	\	3.10^{-2}	1.10^{-2}
3	200	$9, 2.10^{-1}$	4.10^{-2}	2.10^{-2}	$1, 8.10^{-1}$	2.10^{-2}
4	200	6, 36	$9, 8.10^{-1}$	$3, 8.10^{-1}$	6.10^{-1}	1.10^{-2}
1	500	18	\	\	\	7.10^{-2}
2	500	$7, 2.10^{-1}$	5.10^{-2}	3.10^{-2}	5.10^{-2}	8.10^{-2}
3	500	1, 6	$1, 1.10^{-1}$	4.10^{-2}	$1, 3.10^{-1}$	2.10^{-2}
4	500	20	\	\	\	7.10^{-2}

TABLE 3 - Temps d'exécution des différents algorithmes en fonction des différents types et tailles de matrices.

Type de matrice	Taille	Nb itérations v0	Nb itérations v1	Nb itérations v2	Nb itérations v3
1	50	437	37	2	37
2	50	42	2	\	2
3	50	57	4	\	4
4	50	431	37	2	37
1	200	1961	263	14	263
2	200	72	8	\	8
3	200	229	15	2	15
4	200	1971	265	14	265
1	500	5793	\	\	\
2	500	180	17	2	17
3	500	654	44	3	44
4	500	5836	\	\	\

TABLE 4 - Nombre d'itérations des différents algorithmes en fonction des différents types et tailles de matrices.

On sait la vitesse de convergence d'un algorithme qui calcule les valeurs propres λ_i tq $i \in \{1..n\}$ est inversement proportionnel au rapport $\left| \frac{\lambda_{i+1}}{\lambda_i} \right|$.

En d'autres termes, plus ce rapport est proche de 1 (et donc plus les valeurs propres sont proches entre elles) plus l'algorithme mettra du temps à converger. Et plus ce rapport est proche de 0 plus la convergence sera atteinte rapidement.

N.B : Ce rapport est compris entre 0 et 1 parce que les valeurs propres sont classées par ordre décroissant.

D'après les deux tableaux ci-dessus on remarque que la matrice pour laquelle les algorithmes convergent le plus rapidement(en temps de calcul et en nombre d'itérations) est la matrice 3. Ceci est dû au fait que le rapport $\left| \frac{\lambda_{i+1}}{\lambda_i} \right|$ est petit devant 1. Au contraire, on peut remarquer que pour les matrices 1 et 4, qui ont une répartition de valeurs propres similaires et pour lesquelles ce rapport est proche de 1.

Quant à la matrice 2, et vu que ces valeurs sont générées aléatoirement les temps de calcul varient mais on peut remarquer que les algorithmes pour la matrice 2 étaient plus rapides que pour les matrices 1 et 4 mais moins rapides que pour la matrice 3.

Finalement, on remarque que pour des petites tailles ($n = 50$) les algorithmes ne convergent pas pour les matrices 2 et 3 lorsqu'ils convergent bien pour des tailles de matrices élevées ($n = 500$). Inversement, pour des grandes tailles ($n = 500$) les algorithmes ne convergent pas pour les matrices 1 et 4 lorsqu'ils convergent pour des petites tailles ($n = 50$). Ceci est dû au fait que plus n est grand plus le rapport $\frac{n}{n+1}$ se rapproche de 1 et par suite ralentit la convergence jusqu'à arriver à un point où celle-ci n'est plus atteinte.

Partie 2

Soit A la matrice qui représentera notre image. A est de taille $q \times p$ une matrice de taille $p \times q$ avec $q > p$, les images que nous utilisons sont donc en format portrait. Grâce à une SVD nous pouvons l'exprimer comme le produit de trois matrices :

$$A = U \Sigma V^T$$

où

$$U = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1q} \\ u_{21} & u_{22} & \cdots & u_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ u_{q1} & u_{q2} & \cdots & u_{qq} \end{bmatrix}$$

est une matrice de taille $q \times q$,

$$\Sigma = \begin{bmatrix} \sigma_{11} & 0 & \cdots & 0 \\ 0 & \sigma_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_{qp} \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

est une matrice de taille $q \times p$, et

$$V^T = \begin{bmatrix} v_{11} & v_{12} & \cdots & v_{1p} \\ v_{21} & v_{22} & \cdots & v_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ v_{p1} & v_{p2} & \cdots & v_{pp} \end{bmatrix}$$

est la transposée d'une matrice de taille $p \times p$.

Réponse question 1 :

Les matrices, U_k, Σ_k, V_k sont telles que :

- U_k est de taille $q \times k$.
- Σ_k est une matrice diagonale de taille $k \times k$.
- V_k est de taille $p \times k$.

Réponse question 2 :

Le but de cette partie est de reconstruire la matrice I_k à partir des k valeurs propres dominantes de la matrice $M = I^T I$.

Ces valeurs propres, ainsi que les vecteurs propres associés nous permettront de calculer U_k, Σ_k et V_k^T . Ceci nous permettra alors de reconstituer l'image en calculant $I_k = U_k \Sigma_k V_k^T$.

Le but est donc de trouver la plus petite valeur de k qui contient assez d'information pour reconstruire l'image. Ceci se formalise mathématiquement par une variable qu'on appelle pourcentage et qui représente le pourcentage de clarté que nous cherchons pour l'image.

Or, un des paramètres des fonctions `power_method` et `subspace_iter` que nous avons implémentées est le pourcentage. Par suite, en modifiant ce pourcentage, l'algorithme se charge lui-même de nous fournir la plus petite matrice de valeurs propres qui contient le plus d'informations; ce qui revient à donner la valeur optimale de k . Nous ne donnons donc plus k en dur dans le code, celui-ci est égal à la taille de la matrice D des valeurs propres.

Après avoir implémenté cet algorithme dans le fichier "ReconstructionImage.m" (voir code fourni). Nous avons testé de reconstruire l'image fournie avec les différents algorithmes implémentés. Pour chacun de ces algorithmes, et pour des pourcentages fixés, nous avons noté les tailles des différentes matrices D et V (les matrices de valeurs et vecteurs propres).

En effet, quand les matrices sont petites, cela veut dire que nous avons réussi à reconstruire la même image (et donc avec la même clarté) avec des espaces mémoires et des temps de calculs différents.

On trouve alors :

Algorithme	Pourcentage	Max Nb vp	Taille de la matrice D
eig	0.99	200	944×944
subspace_iter_v0	0.99	200	200×200
subspace_iter_v1	0.99	200	50×50
subspace_iter_v2	0.99	200	50×50
subspace_iter_v3	0.99	200	50×50

TABLE 5 - Comparaison des différents algorithmes pour la reconstruction d'image.

On remarque que les tailles des matrices de valeurs propres pour v1, v2 et v3 sont de 50×50 . Cela veut dire que, bien que l'algorithme pouvait aller jusqu'à 200 valeurs propres (Max Nb vp) il s'est arrêté à 50. Cela veut dire que 99 % de l'information se trouve dans les 50 premières valeurs propres. Quant à l'algorithme v0, il a dû continuer de tourner jusqu'à arriver à 200 vps. Cela veut dire que même après avoir calculé 200 vps il n'avait pas 99% de l'information. L'image fournie par v0 est donc plus floue que celle fournie par v1 avec les mêmes paramètres. (voir code fourni).

Quant à l'algorithme eig, la taille est de 944 parce que ce dernier ne prend pas de pourcentage, il calcule toutes les valeurs propres et donc la taille de la matrice fournie est la même que celle de l'imgae. Par contre, il n'est pas du tout nécessaire d'aller jusqu'à calculer 944 vps pour eig. Dans ce cas là on aurait pu fixer nous même la valeur de k.