Projet 3
Compression d'image à travers la factorisation SVD

Groupe 1 - Equipe 2 - 12397

Responsable : bboudeau
Secrétaire : ndo
Codeurs : mbailleul, tberard, ldujardin

Résumé : Le but de ce projet consiste à programmer un algorithme permettant de faire de la compression d'images en utilisant des techniques matricielles basée sur la factorisation SVD.
La première partie s'intéresse à mettre en place les fonctions utilitaires pour manipuler les matrices de Householder, nécessaires à la réalisation des deux parties suivantes. La seconde partie, elle, transforme la matrice A en une matrice bidiagonale BD par une méthode directe. Ensuite, la troisième partie va consister à transformer une matrice bidiagonale BD en une matrice diagonale S par une méthode itérative. Enfin, la dernière partie applique ces transformations à l'algorithme de compression d'image par transformation SVD.

1 Transformation de Householder

La matrice de Householder H est une matrice carrée définissant une symétrie hyperplane sous la forme $H = Id - 2 * U.\ {}^tU$, où U est un vecteur colonne. Pour un système de la forme $H.X = Y$ avec X et Y deux vecteurs donnés, d'après le chapitre 2 du cours, U est égale à $\frac{X-Y}{||X-Y||}$ (1). Par la suite, il est aisé d'écrire un algorithme utilisant (1) pour obtenir H.

Cependant, pour exécuter un produit Householder-vectoriel, il est possible d'optimiser le coût de ce produit en passant par la décomposition de H. En effet $HX = X - 2 * U.\ {}^tU.X$, or ${}^tU.X$ est un scalaire et se détermine en temps linéaire, et ainsi, il n'y a qu'une différence de vecteur à réaliser, le tout en coût linéaire. Ainsi, étendue sur une matrice M de dimension $n * m$, le coût du produit Householder-matriciel est en $O(n * m)$, contre $O(n * n * m)$ pour un produit matriciel classique.

2 Mise sous forme bidiagonale

Dans cette partie, on cherche à déterminer une base dans laquelle une matrice $A \in M_{n,m}(R)$ est bidiagonale. Pour se faire, on implémente dans le fichier `bidiagonale.py` l'algorithme qui nous a été fourni.
On note Q_{left} la matrice de changement de base de départ et Q_{right} la matrice de changement de base d'arrivée. On obtient donc l'invariant $Q_{left} \times BD \times Q_{right} = A$ avec BD une matrice bidiagonale.
Dans la fonction de bidiagonalisation, on doit vérifier l'égalité ci-dessus à chaque itération de la boucle for (mais pas nécessairement que BD soit bidiagonale). On vérifie donc cet invariant de boucle lors de nos tests avec la fonction `test_inv_boucle(A, seuil)` qui détermine si l'invariant de boucle ci-dessus est vérifié selon un seuil de tolérance `seuil` du point de vue de la norme matricielle quadratique.
On lance la fonction de test sur des matrices aléatoires, générées par la fonction `matrix_generate` et on fait varier le seuil pour lequel on peut considérer que la norme de la matrice est assez petite pour considérer la norme nulle et donc que la matrice est nulle via la règle de séparation sur la norme matricielle.
Algorithmiquement, on considère que l'invariant vrai si et seulement si $||Q_{left} \times BD \times Q_{right} - A|| < seuil$

Nous venons de voir que la matrice retournée par `bidiagonale` est bien la matrice entrée en

Valeur du seuil	Retour du test
10^{-3}	True
10^{-6}	True
10^{-9}	True
10^{-13}	True
10^{-14}	False

TABLE 1 – Résultats des tests de l'invariant de boucle de la fonction `bidiagonale`

paramètre écrite dans une autre base. On souhaite finalement tester si la fonction `bidiagonale` renvoit bien une matrice bidiagonale. Pour se faire, la fonction `test_bidiag(N, n, seuil)` génère N matrices de taille $n * n$ et teste si l'appel de `bidiagonale` sur chacune d'entre elle retourne bien une matrice bidiagonale, avec un seuil d'erreur `seuil` du point de vue de la norme matricielle quadratique.

Valeur de N	Valeur de n	Valeur du seuil	Retour du test de bidiagonalité
10^1	2	10^{-9}	True
10^1	4	10^{-13}	True
10^2	4	10^{-13}	True
10^2	4	10^{-14}	False
10^1	2	10^{-14}	False
10^1	2	10^{-15}	False

TABLE 2 – Résultats des tests de matrices retournées par la fonction de bidiagonalisation

Les résultats obtenus sur la table 1 montrent qu'il faut prendre un seuil d'au moins 10^{-13} puisqu'on remarque que dès qu'on met le seuil en dessous de 10^{-13} la norme matricielle peut renvoyer un résultat supérieur à 10^{-14}
Les résultats obtenus sur la table 2 montrent quant à eux que pour considérer un élément de la matrice comme étant nul, il faut fixer un seuil de 10^{-13} puisqu'on voit que si on fixe un seuil de 10^{-14} ou plus petit, on a des matrices qui ne sont pas considérées comme étant bidiagonales.
Le fait que les deux valeurs des seuils soient à 10^{-13} est a priori une simple coïncidence.
Après d'autres versions de tests, nous nous sommes rendus compte que notre algorithme de bidiagonalisaton ne fonctionnait que dans le cas de matrices carrées et dans le cas où les matrices contiennent plus de lignes que de colonnes mais pas dans le cas où la matrice a plus de colonnes que de lignes. Cela est dû à la manière de parcourir la matrice dans la fonction `bidiagonale`.

3 Transformations QR

Cette partie a pour but d'implémenter la deuxième étape de la décomposition SVD à savoir appliquer un certain nombre de fois la décomposition QR. Le but étant de transformer une matrice bidiagonale A en entrée, en une matrice diagonale S tout en fournissant les matrices de changement de base U et V telles que $U \times S \times V = A$.

3.1 Première implémentation

Pour cela nous avons traduit le pseudo code de cette étape de la SVD, la fonction python ainsi créée demande deux valeurs d'entrées :

— Une matrice A avec comme pré-condition d'être bidiagonale.

— Un entier `NMax` qui correspond au nombre de répétitions de la transformation QR par la fonction.

À priori on ne connaît pas la valeur de `NMax` à fournir à la fonction pour que celle-ci renvoie une matrice diagonale. C'est pour cette raison que nous avons effectué des tests sur la convergence de la matrice de sortie vers une matrice diagonale.

3.2 Convergence de S

Afin de vérifier si une matrice est diagonale, la fonction `isDiagonal` a été implémentée. Pour éviter que ce test de "diagonalité" ne soit trop strict et donc qu'une matrice avec des éléments extra-diagonaux extrêmement petits (de l'ordre de 10^{-12} par exemple) soit considérée non diagonale, 2 paramètres seuils ont été ajoutés aux paramètres d'entrées de la fonction. Cette dernière prend donc une matrice A et deux valeurs seuils `valueThreshold` et `returnThreshold`. La fonction calcule : $\frac{\texttt{sum}}{\texttt{count}}$ avec `sum` la somme des éléments extra-diagonaux supérieurs à `valueThreshold` et `count` leur nombre.

Enfin, la fonction vérifie si cette valeur calculée est inférieure à `returnThreshold`, si c'est le cas la fonction renvoie 0, ce qui correspond à une matrice diagonale sinon elle renvoie la valeur calculée qui est un indicateur de combien la fonction est proche d'être diagonale ou du moins considérée diagonale.

La fonction `isDiagonal` est alors utilisée dans une fonction adaptée des décompositions QR successives mais à chaque décomposition QR elle retient la valeur prise par `isDiagonal(S)` dans le but d'évaluer la convergence de S vers une matrice diagonale. Cette fonction, associée à une fonction de génération de matrices aléatoires nous a permis d'afficher cette courbe de convergence pour 11 matrices de tailles différentes et aléatoires selon le nombre d'itérations de la décomposition QR. (Figures 1 et 2)

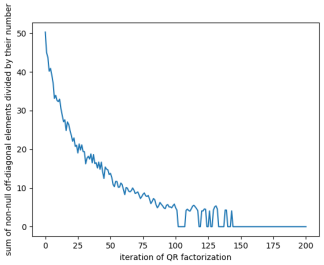

FIGURE 1 – valeurs de `isDiagonal` pour chaque itération de QR sur 1 matrice de taille 80

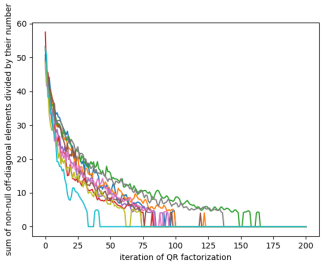

FIGURE 2 – valeurs de `isDiagonal` pour chaque itération de QR sur 10 matrices de taille aléatoire entre 10 et 100

Après environ 170 décompositions QR toutes les matrices sont diagonales. Ici `valueThreshold` = 10^{-13} et `returnThreshold` = 4. Sur les figures 1 et 2 on remarque qu'un comportement particulier apparaît juste avant la convergence : les matrices ne sont plus considérées diagonales après l'avoir été. Cela est dû à `returnThreshold`, les matrices vont continuer de changer après avoir été considérées diagonales et sont donc susceptibles de repasser au dessus de ce seuil. Cependant cela n'empêche pas

leur convergence au final puisqu'après quelques oscillations elles restent considérées diagonales. Les matrices convergent donc rapidement vers des matrices diagonales.

Néanmoins cette convergence dépend de la taille de la matrice. En effet, après avoir testé pour une matrice deux fois plus grande à savoir 200, une convergence autour de la 320 ème itération a été observée, ce qui reste acceptable. La convergence semble donc avoir une complexité en `O(n)` où n est le maximum entre la longueur et la largeur de la matrice bidiagonale donnée.

3.3 Invariants

Après cela, pour vérifier que la fonction de décomposition SVD soit correcte il faut vérifier que tout au long de son exécution on ait $U \times S \times V = A$.

La vérification de cet invariant a été effectuée à l'aide d'une fonction intermédiaire `isNormZero` qui vérifie si la norme d'une matrice est proche de zéro, ce qui nous permet de considérer nulle une norme très proche de zéro.

Après chaque décomposition QR, la fonction de test de l'invariant vérifie que la norme de $U \times S \times V - A$ est proche de zéro grâce à `isNormZero`, si elle ne l'est pas assez vis-à-vis du seuil, la fonction s'arrête et renvoie faux, si après `NMax` décompositions QR l'invariant est encore vérifié alors elle renvoie vrai.

Afin de vérifier plus rigoureusement cela, la fonction de création de matrice bidiagonales aléatoires a été utilisée afin d'effectuer ce test sur 10 matrices aléatoires. Nous avons remarqué pendant ces tests que le seuil doit être supérieur ou égal à 10^{-13} sinon l'invariant n'est pas considéré comme étant respecté.

3.4 Optimisation

Le sujet nous indique qu'il est possible d'optimiser la décomposition QR dans le cas de matrices bidiagonales. Pour pouvoir appliquer cette optimisation il faut d'abord vérifier que les décompositions QR effectuées lors de la SVD ont été faites sur des matrices qui respectent la caractéristique de bidiagonalité. Pour cela, la même fonction avec seuil, `isBD`, de la partie 2 a été utilisée. L'invariant testé a été : **S, R1 et R2 sont bidiagonales**. La fonction de décompositions QR successives a été adaptée afin d'appeler `isBD` sur S, R1 et R2 après chaque décomposition QR. Comme pour les tests précédents, 10 matrices bidiagonales générées aléatoirement puis ont été données à la fonction de test et ce qu'il en retourne est que quelque soit le seuil de tolérance, si la matrice en entrée est strictement bidiagonale (avec des 0 "purs" hors-bidiagonale) alors l'invariant est respecté. Sinon, si l'on donne une matrice générée par la fonction `bidiagonale` de la partie 2, on remarque que le seuil minimum accepté est 10^{-8}. Cela est dû au fait que la matrice fournie est bidiagonale avec un seuil de 10^{-13}, et on ajoute à cela la décomposition QR qui va sûrement toucher aux éléments hors bidiagonale de la matrice et ainsi altérer la bidiagonalité de la matrice.

En ce qui concerne la méthode d'optimisation de la décomposition QR sur des matrices bidiagonales, nous ne l'avons pas trouvé.

Pour ce qui est de la complexité, on sait que la complexité d'une décomposition QR est $f(QR) = O(\frac{4}{3}n)$ (f est la fonction complexité). Ici, on a une complexité générale qui est :

$$f(SVD) = NMax \times [2 \times f(QR)] = O(NMax \times \tfrac{8}{3}n)$$

On a omis les complexités des produits matriciels et des transpositions car inférieures à celle de QR.

3.5 Propriété sur S

Usuellement, la décomposition SVD demande à ce que les éléments de la matrice S soient positifs et ordonnés de manière décroissante. Cette propriété va être appliquée à notre matrice S et va modifier la matrice U par conséquence grâce à la fonction `apply_propriety`. Cette fonction va dans un premier temps calculer le produit scalaire, noté Y, entre U et S qui ont été obtenues par la décomposition SVD. Ensuite, elle va rendre positive et trier dans l'ordre décroissant les éléments de S et étant donné que la matrice S converge vers une matrice diagonale, il est aisé d'obtenir son inverse S^{-1} en inversant chaque élément de la diagonale. De ce fait, nous calculons ensuite la matrice U tel que $U = Y \times S^{-1}$. Nous avons effectué quelques tests sur diverses matrices U et S afin de vérifier le bon fonctionnement de notre fonction.

4 Application à la compression d'image

Le but de cette partie est de compresser une image par transformation SVD. Les images manipulées, étant en couleurs, correspondent à des matrices de triplets RGB. Il a alors fallu implémenter une fonction `color_extraction` qui permet d'extraire les composantes rouge, verte et bleue de ces images avant de travailler sur la décomposition.

Ensuite, nous avons implémenté une fonction `compression_rank` qui prend en paramètre A qui est une matrice de triplets RGB (la matrice d'une image) et un rang k et qui renvoie la matrice d'une image compressée au rang k. Dans cette fonction, nous appliquons `color_extraction` sur la matrice A et une fois les matrices des composantes de couleurs obtenues, nous effectuons une décomposition SVD sur chacune d'elles. Dans le but d'obtenir une compression au rang k, nous mettons à zéro les termes diagonaux d'indice strictement supérieur à k des matrices S obtenues par la décomposition SVD. Pour finir, nous recomposons les matrices des composantes rouge, verte et bleue avec les matrices S modifiées et nous les assemblons pour reconstituer une matrice de triplets RGB qui correspond alors à la matrice de l'image compressée. Néanmoins la compression d'une image de grande taille par notre fonction est très coûteuse et prend du temps du fait que la bidiagonalisation n'est, dans notre cas, pas très optimisée.

Les résultats obtenus pour la compression d'une image de taille 250x250 pour différents rang k est visible Figure 3.

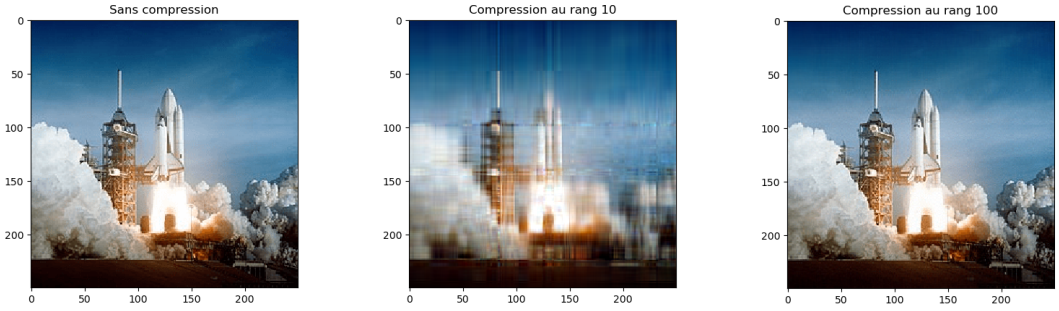

FIGURE 3 – Compression d'une image

4.1 Gain de place

Soit $A = U \times S \times V \in \mathcal{M}_n$ une matrice représentant une image, et $A_k \in \mathcal{M}_n$ celle représentant l'image compressée au rang k.

On a $A_k = US_kV = \begin{pmatrix} U_{1,1} & U_{1,2} \\ U_{2,1} & U_{2,2} \end{pmatrix} \begin{pmatrix} S'_k & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} V_{1,1} & V_{1,2} \\ V_{2,1} & V_{2,2} \end{pmatrix} = \begin{pmatrix} U_{1,1}S'_kV_{1,1} & U_{1,1}S'_kV_{1,2} \\ U_{2,1}S'_kV_{1,1} & U_{2,1}S'_kV_{1,2} \end{pmatrix}$

Seul les blocs $U_{1,1}$, $U_{2,1}$, $V_{1,1}$, $V_{1,2}$ et S'_k interviennent dans le calcul de A_k. Le nombre de coefficient nécessaire de U, S_k et V est $2nk + k$. Le gain de place de la compression est donc de $n^2 - k(2n + 1)$ coefficients. Par conséquent, pour que l'image compressée ne soit pas de taille supérieure à celle de l'image original, il faut avoir $k \leq \frac{n^2}{2n+1}$.

4.2 Efficacité de la compression

Nous avons testé l'efficacité de la compression sur l'image de la Terre, Figure 4, en traçant la distance entre cette image et l'image compressée en fonction du rang de compression, Figure 5. Cette distance définie par $dist(A, B) = ||A - B||$, et $||M|| = \sqrt{Tr({}^tRR) + Tr({}^tGG) + Tr({}^tBB)}$ avec R, G, B, les 3 composantes de couleur de l'image M.

FIGURE 4 – Terre

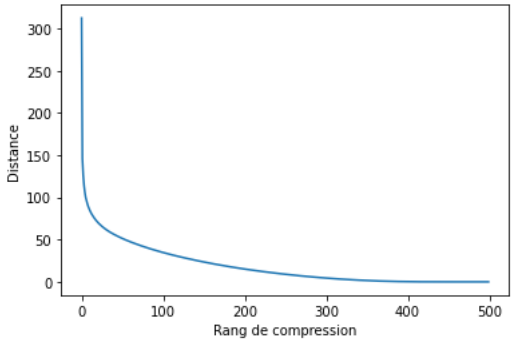

FIGURE 5 – Efficacité de la compression