Projet n°3
Compression d'images à travers la factorisation SVD

Groupe n°3 - Equipe n°2
Responsable : paubreton
Secrétaire : arodrigue003
Codeurs : jvitrat

Résumé : Dans ce projet, le principal but était de programmer un ou des algorithmes permettant de faire de la compression d'images, en se servant de certaines techniques matricielles basées sur la factorisation SVD (Singular Value Decomposition). Nous allons donc vous exposer tout d'abord le travail réalisé sur les matrices de Householder, puis successivement la transformation d'une matrice en matrice bidiagonale par une méthode directe, la transformation de cette matrice bidiagonale en une matrice diagonale par une méthode itérative, puis enfin l'application de la méthode SVD sur ces matrices.

1 Transformation de Householder

Le but de cette partie est d'écrire des algorithmes qui serviront dans les parties suivantes et qui se basent sur les matrices de Householder pour projeter un vecteur sur un autre.

1.1 Projection de Householder

Soit la matrice de Householder H et un vecteur U tel que $H = Id - 2 \times U^t U$. On considère deux vecteurs X et Y de même norme et on cherche U tel que $H.X = Y$. On distingue deux cas :
— $X = Y$: dans ce cas on choisit $U = 0$. On a alors $H = Id$ et on a bien : $H.X = Id.X = X = Y$.
— $X \neq Y$: dans ce cas on choisit $U = \frac{X-Y}{\|X-Y\|}$. On a alors :

$$\begin{aligned}
H.X &= (Id - 2 \times \frac{(X-Y)^t(X-Y)}{\|X-Y\|^2}).X \\
&= X - 2 \times \frac{X^tXX - X^tYX - Y^tXX + Y^tYX}{\|X-Y\|^2} \\
&= X - 2 \times \frac{\|X\|^2X - \langle X.Y\rangle X - \|X\|^2Y + \langle X.Y\rangle Y}{\|X-Y\|^2} \\
&= X - 2 \times \frac{(\|X\|^2 - \langle X.Y\rangle)(X-Y)}{\|X-Y\|^2} \\
&= X - 2 \times \frac{(\|X\|^2 - 0.5(\|X\|^2 + \|Y\|^2 - \|X-Y\|^2)(X-Y)}{\|X-Y\|^2} \text{ or } \|X\| = \|Y\| \\
&= X - (X-Y) = Y \text{ ce qui démontre que le choix de } U \text{ est correct.}
\end{aligned}$$

1.2 Fonction optimisé

Nous avons écrit une fonction qui calcule U pour deux vecteurs X et Y donnés ainsi qu'une fonction qui va calculer le produit d'une matrice de Householder par un vecteur de manière conventionnelle. Enfin la fonction $householderProjection(X, Y)$ renvoie la matrice de Householder qui projette le vecteur X sur le vecteur Y.
Nous n'avons cependant pas réussi à écrire de fonction qui calcule de manière optimisée le produit entre une matrice de Householder et un (ou plusieurs) vecteur(s).

2 Mise sous forme Bidiagonale

Le but de cette partie est de décomposer une matrice A en trois matrices différentes : $QLeft$, BD et $QRight$ tel que $A = QLeft \times BD \times QRight$.
De plus chacune de ces matrices doit avoir des propriétés particulières, ainsi $QLeft$ et $QRight$ sont des matrices orthogonales (c'est à dire des matrices Q telle que $Q^tQ = Id$) et BD est une matrice bidiagonale (c'est à dire une matrice nulle excepté pour ses coefficients diagonaux et sur-diagonaux).

Nous avons pour cela d'abord écrit les fonctions $getColumnFromMatrix$ et $getRowFromMatrix$ qui permettent d'extraire la colonne i (respectivement la ligne i) de la matrice qui leur est passée en

entrée. Nous appliquons ensuite l'algorithme de transformation bidiagonale qui est fourni dans le sujet (on note e_i le i$^{\text{ème}}$ vecteur de la base canonique) :

Données : A une matrice de taille $n \times m$
Résultat : $QLeft$, BD et $QRight$ tel que $A = QLeft \times BD \times QRight$
début
  $Qleft \leftarrow Id_n$, $BD \leftarrow A$, $QRight \leftarrow Id_m$
  pour tous les $i \in [0, n[$ **faire**
    $u \leftarrow BD[i : n, i]$, $v \leftarrow \|u\| \times e_i$
    $Q1 \leftarrow householderProjection(u, v)$
    $QLeft \leftarrow QLeft \times Q1$, $BD = Q1 \times BD$
    si *ij(m-2)* **alors**
      $u \leftarrow BD[i, (i+1) : m]$, $v \leftarrow \|u\| \times e_{i+1}$
      $Q2 \leftarrow householderProjection(u, v)$
      $QRight \leftarrow Q2 \times QRight$, $BD = BD \times Q2$
  pour tous les $i \times j \in [0, n[\times[0, m[$ *telle que* $i \neq j$ *ou* $i + 1 \neq j$ **faire**
    $BD[i, j] \leftarrow 0$
  Retourner $(QLeft, BD, QRight)$

Algorithme 1 : Algorithme de décomposition bidiagonale

A la fin de l'algorithme, on met manuellement des zéros dans les cases de la matrice qui ne font pas partie de la diagonale ou de la sur-diagonale afin d'améliorer la stabilité des algorithmes qui vont réutiliser la matrice BD par la suite.

Pour tester l'algorithme, nous générons une matrice aléatoire A de taille $n \times m$ où $n, m \in [0, 50[$, on applique ensuite la transformation BD et on vérifie que :
— $A = QLeft \times BD \times QRight$
— $QLeft$ est une matrice orthogonale
— $QRight$ est une matrice orthogonale
— BD est une matrice bidiagonale
Il suffit d'exécuter le fichier *Test_Forme_Bidiag.py* afin de réaliser ces tests.

3 Transformation QR

Dans cette partie, le but est de décomposer la matrice bidiagonale précédemment générée en trois matrices U, S et V de sorte que $BD = U \times S \times V$ et que S soit une matrice diagonale.

3.1 Avec numpy

Nous avons utilisé la fonction *numpy.linalg.qr* afin d'obtenir la décomposition QR d'une matrice de taille $n \times m$ en le produit d'une matrice orthogonale de taille $n \times n$ et d'une matrice triangulaire supérieur de taille $n \times m$.

3.2 Convergence

Pour vérifier la convergence de cet algorithme, nous avons tracé l'évolution du nombre de termes extra-diagonaux de R non nuls en fonction du nombre d'itérations de l'algorithme de transformation QR (cf figure 1). L'algorithme vérifie de plus à chaque étape que la décomposition obtenue est bien égale à la matrice initiale.

3.3 Matrices bidiagonales

Nous n'avons pas réussi à démontrer l'invariant : "Les matrices S, R_1 et R_2 sont toujours bidiagonales", nous l'admettrons donc dans toute la suite du projet.

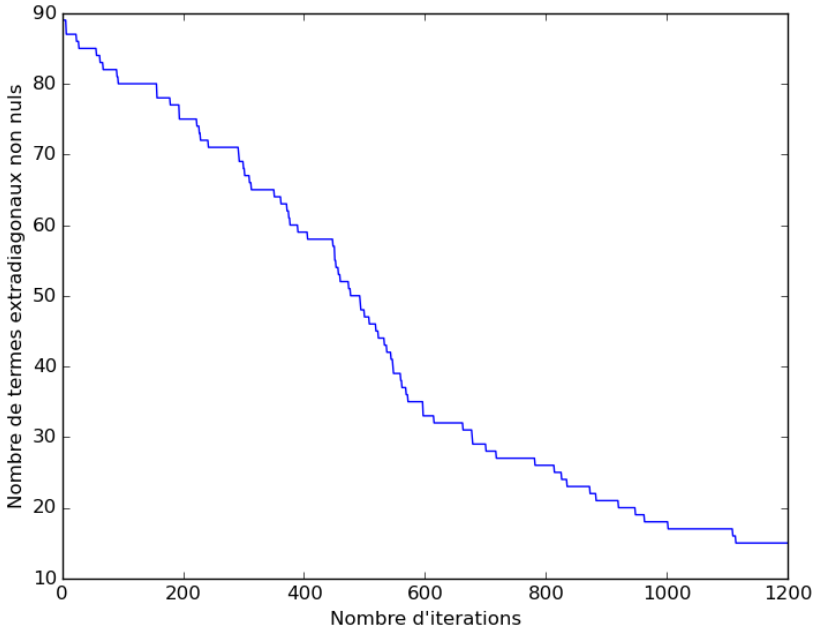


FIGURE 1 – Vitesse de convergence de R vers une matrice diagonale en fonction du nombre d'itérations

3.4 Algorithme optimisé

D'après l'algorithme utilisé et les invariants de la question précédente ${}^tS$ est bidiagonale inférieure, de même pour ${}^tR_1$. Ainsi le but de cette question est de créer un algorithme de décomposition QR qui soit optimisé pour décomposer des matrices bidiagonales inférieures (on les note BDI). Nous avons écrit une fonction qui à partir des termes $BDI[i,j]$ et $BDI[i-1,j]$ renvoie les coefficients (cs,sn) de la matrice de Givens qui permet d'annuler $BDI[i,j]$. La matrice de Givens associée a ces coefficients est notée G(i,j,cs,sn). On va générer une à une les matrices de Givens pour annuler chacun des termes sur la diagonale de la matrice BDI. On obtient ainsi l'algorithme suivant :

Données : BDI une matrice bidiagonale inférieur de taille $n \times m$
Résultat : Q et R tel que $BD = Q \times R$
1 **début**
2 $\quad Q \leftarrow Id_n$
3 $\quad$ **pour tous les** $i \in [0, n[$ **faire**
4 $\quad\quad cs, sn \leftarrow givens([BDI[i,i], BDI[i+1,i])$
5 $\quad\quad BDI \leftarrow G(i+1, i, cs, sn) \times BDI$
6 $\quad\quad Q \leftarrow Q \times^t G(i+1, i; cs, sn)$
7 $\quad$ **Retourner** (Q, BDI)

Algorithme 2 : Algorithme de décomposition QR pour une matrice bidiagonale

On peut remarquer que cet algorithme peut être optimisé, en effet la multiplication à gauche par une matrice de Givens ne change que deux lignes de la matrice multipliée, de plus dans ce cas précis où la matrice est bidiagonale seules quatre opérations sont nécessaires. Ainsi on change la ligne (5) de l'algorithme pour directement effectuer ces quatre opérations et ne pas faire de produit matriciel.
De même la multiplication à droite par une matrice de Givens ne change que deux colonnes, on fait donc le calcul de ces colonnes à la main plutôt que de faire le produit matriciel dans la ligne (6) de l'algorithme.

On obtient ainsi un algorithme avec une complexité en $\Theta(n^2)$ ce qui est mieux que la version avec des calcul matriciels qui aurait une complexité de $\Theta(n^4)$. On continue cependant d'utiliser *numpy.linalg.qr* car cette fonction est plus rapide (notamment car *linalg* est une bibliothèque compilé et que notre fonction est une fonction interprétée en python). Enfin cet algorithme possède tout de même le défaut de ne pouvoir traiter des matrices où $n < m$.

Pour effectuer tous les tests des algorithmes de cette partie (on test la convergence de la décomposition

qr, l'exactitude de celle-ci ainsi que l'exactitude de l'algorithme optimisé), il suffit d'exécuter le fichier $Test\_transformation\_qr.py$.

4 Application à la compression d'image

Dans cette partie, nous allons finalement appliquer les algorithmes vu précédemment à une image, dans le but de la "compresser". Nous allons utiliser une image fournie représentant un décollage de fusée (image PNG, 400x300).

4.1 Algorithme de traitement de l'image

Nous avons tout d'abord récupéré l'image concernée. La bibliothèque MatPlotLib nous permet donc de la récupérer sous forme d'une matrice de triplets. En effet comme l'image est codée en RGB, nous avons à traiter chaque matrice de couleurs indépendamment l'une des autres.
Nous avons donc découpé la matrice de l'image en trois nouvelles matrices différentes. Chaque matrice contient la matrice de valeurs Rouges (R), Vertes (G) ou Bleues (B) tirée de l'image.

Une fois ceci fait, on applique à chacune des matrices R, G et B obtenues la décomposition SVD. Nous récupérons donc la matrice bidiagonale associée, puis y appliquons une transformation QR. À la matrice S de la décomposition QR que nous obtenons, nous réalisons la compression au rang k. Il est à noter que nous "normalisons" les valeurs en les bornant et les remplaçant si jamais elles dépassent ces bornes.
Finalement nous renvoyons le produit des matrices (QL, U, S modifiée, V et QR).
Pour finir totalement, nous reconstituons l'image en réalisant l'exact inverse de l'algorithme proposé ci-dessus.

Nous obtenons donc les résultats suivants. Depuis l'image originale :

Nous obtenons avec k la compression de valeurs successivement 10, 50 et 100 :



4.2 Gain de place et efficacité

Résumons ce que nous faisons :
— Nous partons d'une matrice M que nous décomposons grâce à la méthode SVD en $M = U \times S \times V$
— Nous compressons la matrice S, c'est à dire que seuls les k premiers éléments diagonaux de la matrice sont conservés

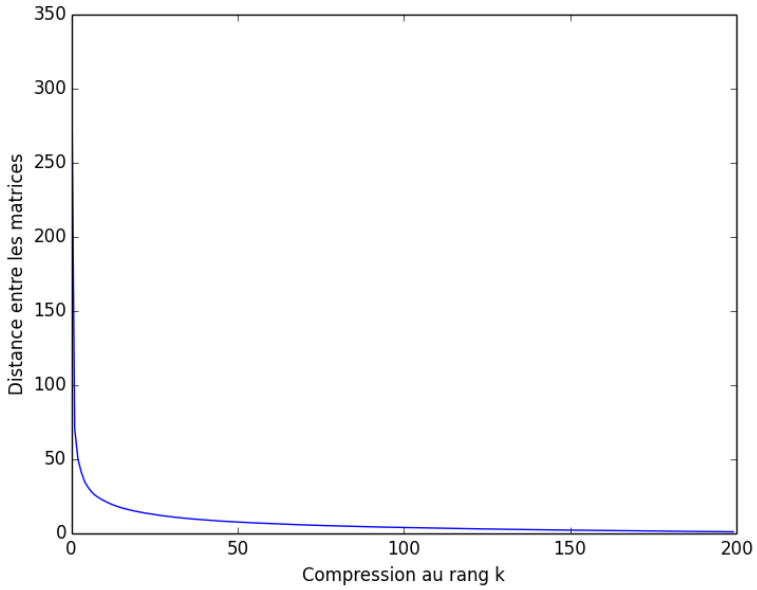
— À partir de cette nouvelle matrice S compressée, nous recalculons la matrice M (avec le même produit).

Considérons donc que la matrice M soit de taille $n \times n$. Comme l'on a recalculé S pour obtenir M, on peut négliger les $(n-k)$ dernières colonnes de la matrice U et les $(n-k)$ dernières lignes de la matrice V. De ce fait nous gagnons $2(n-k)$ de mémoire, en plus des k gagnés en supprimant les k dernières valeurs de la matrice S.
Donc nous gagnons $n^2 - 2(n-k) - k$ de mémoire. Par ailleurs, l'algorithme reste donc rentable tant que $k \leq \frac{n^2}{2(n+1)}$.

L'efficacité de cette compression va se calculer de manière relativement simple : on tente de voir la distance qui sépare la matrice de départ et la matrice d'arrivée. Nous réaliserons donc la norme de la soustraction de ces deux matrices.

$$Efficacite(Matrice\_depart, Matrice\_arrivee) = ||Matrice\_depart - Matrice\_arrivee||$$

Nous réalisons ce calcul sur toutes les matrices trouvées avec la compression du rang 0 au rang 200. Nous obtenons ainsi le graphique suivant :



Nous pouvons donc observer que plus le taux de compression augmente, plus la distance qui sépare les deux images se réduit. Finalement à une compression supérieure à 200 on obtient quasiment l'image de départ.

5 Conclusion

C'est un projet qui nous aura heurté à une certaine difficulté, mais qui aura été très intéressant. En effet, découvrir comment fonctionnent à l'intérieur les mécanismes de compression d'image et l'utilisation de la méthode SVD nous aura permis d'agrandir un peu plus notre horizon sur ce qui concerne la culture mathématique et informatique.