CC : Multiplication de matrices

I. Rappels d'algèbre linéaire

Matrices et endomorphismes

1. Soit $(v_0,v_1,\dots,v_{n-1}) \in V^n$ telle que cette identité soit satifaite. Montrons que c'est une base :
   - Montrons tout d'abord qu'elle est libre : supposons qu'il existe $(\lambda_0,\lambda_1,\dots,\lambda_{n-1}) \in \mathbb{R}^n$ tels que $\sum_i \lambda_i v_i = 0$.
     Soit $u \in \mathcal{L}(V)$ et soit M telle que la propriété soit satifaite. On définit M' par $m'_{ij} = m_{i,j} + \lambda_i$ et on constate que c'est une autre matrice qui satisfait la même relation. Par unicité, $M = M'$ donc les λ_i sont tous nuls.
     On déduit qu'elle est libre.
   - Montrons qu'elle est génératrice : Soit $x \in V$. Comme la famille est libre, $v_0 \neq 0$ et soit $u \in \mathcal{L}(V)$ telle que $u(v_0) = x$. (Il existe de telles applications linéaires, on peut par exemple en construire en completant v_0 en une base de V, et on choisissant que l'image de tous les autres vecteurs de cette base soit nulle.)
     alors il existe M qui satisfait l'égalité précédente et en particulier $x = u(v_0) = \sum_i m_{i,0} v_i$ donc x est combinaison linéaire des v_i.
   - **Remarque** : comme la famille a $n = \dim V$ éléments, il était en fait inutile de montrer qu'elle est génératrice, la liberté suffisait pour dire que c'est une base.
2. (a) $\mathcal{B}_0 = (1,X,X^2,\dots,X^n)$.
   (b) On obtient les coefficients des matrices en calculant les coordonnées de $D(v_j)$ et $T(v_j)$:
   - $D(X^j) = \begin{cases} jX^{j-1} & \text{si } j > 0 \\ X^n & \text{si } j = 0 \end{cases}$ donc la matrice de D est $\begin{pmatrix} 0 & 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 2 & 0 & \dots & 0 \\ 0 & 0 & 0 & 3 & \dots & 0 \\ \vdots & & \ddots & \ddots & \ddots & \\ 0 & \dots & \dots & 0 & 0 & n \\ 1 & 0 & \dots & \dots & 0 & 0 \end{pmatrix}$.
   - $T(X^j) = jX^j$ donc la matrice de T est $\begin{pmatrix} 0 & 0 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 2 & 0 & \dots & 0 \\ \vdots & & \ddots & \ddots & \ddots & \\ 0 & \dots & \dots & 0 & n-1 & 0 \\ 0 & \dots & \dots & 0 & 0 & n \end{pmatrix}$

Implémentation en python

3. Fonctions en python pour ces matrices :

```
def matrice_de_D(n): #P'+X^n P(0)
    return [ [j*(j==i+1)+(i==n and j==0) for j in range(n+1)]   for i in range(n+1)]

def matrice_de_T(n): # X P'
    return [ [i if i==j else 0 for j in range(n+1)]   for i in range(n+1)]
```

Produit de matrices

4. Soit $(\alpha_0, \alpha_1, \ldots, \alpha_{n-1}) \in \mathbb{R}^n$, calculons $u \circ v(\sum_i \alpha_i v_i)$:

$$u(v(\sum_i \alpha_i v_i)) = u(\sum_i \underbrace{\sum_j b_{ij}\alpha_j}_{\beta_i} v_i) = \sum_i \sum_k a_{i,k}\beta_k v_i = \sum_i \underbrace{\sum_k a_{ik}b_{kj}}_{c_{ij}} \alpha_j v_i$$

ce qui donne l'expression annoncée pour la matrice C.

II. Premier algorithme

5. Le première fonction est la bonne. En effet pour la première ligne (c'est à dire la première sous-liste, qui correspond à l'itération ou `i=0`), en prennant en compte la formule précédente, elle calcule $m_{0,0}$, puis $m_{0,1}$, etc.
   Au contraire, la deuxième fonction renvoit pour la première ligne (c'est à dire la première sous-liste, qui correspond à l'itération ou `j=0`), en prennant en compte la formule précédente, elle calcule $m_{0,0}$, puis $m_{1,0}$, etc. ce qui aurait du être mis en première colonne et non pas en première ligne.

6. Il y a n^3 multiplications, car on imbrique trois boucles de n intérations, (donc au total n^3 itérations) avec une multiplication à chaque fois.

III. Matrices par blocs

Produit de matrices par blocs

7. Soient $M' = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$, $M'' = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$ et $M = M' \cdot M''$. On va calculer les coefficients $m_{i,j}$ de M en utilisant que les coefficients m'_{ij} de M' sont égaux à $a_{i,j}$ ou $b_{i,j-p}$ ou $c_{i-m,j}$ ou $d_{i-m,j-p}$ selon que i et j soient plus grands ou plus petit que n (et on fera de même pour M'') :

$$m_{i,j} = \sum_{k=0}^{n+p-1} m'_{i,k}m''_{k,j} = \sum_{k=0}^{n-1} m'_{i,k}m''_{k,j} + \sum_{k=n}^{n+p-1} m'_{i,k}m''_{k,j}$$
$$= \begin{cases} \sum_{k=0}^{n-1} a_{i,k}e_{k,j} + \sum_{k=n}^{n+p-1} b_{i,k-n}g_{k-n,j} & \text{si } i < n \text{ et } j < n \\ \sum_{k=0}^{n-1} a_{i,k}f_{k,j-m} + \sum_{k=n}^{n+p-1} b_{i,k-n}h_{k-n,j-n} & \text{si } i < n \text{ et } j \geqslant n \\ \sum_{k=0}^{n-1} c_{i-n,k}e_{k,j} + \sum_{k=n}^{n+p-1} d_{i-n,k-n}g_{k-n,j} & \text{si } i \geqslant n \text{ et } j < n \\ \sum_{k=0}^{n-1} c_{i-n,k}f_{k,j-n} + \sum_{k=n}^{n+p-1} d_{i-n,k-n}h_{k-n,j-n} & \text{si } i \geqslant n \text{ et } j \geqslant n. \end{cases}$$

Cela signifie précisément que $M = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$.

Matrices de taille $2^k \times 2^k$

8. Fonction `sous_matrice` :

```
def sous_matrice(A,p,q):
    return [[A[i,j] for j in range(q)] for i in range(p)]
```

Premier algorithme de type "Diviser pour mieux régner"

9. Fonction `blocs` :

```
def blocs(M):
    n=len(M)//2
    A=return [[M[i][j] for j in range(n)] for i in range(n)]
    B=return [[M[i][j] for j in range(n,2*n)] for i in range(n)]
    C=return [[M[i][j] for j in range(n)] for i in range(n,2*n)]
    D=return [[M[i][j] for j in range(n,2*n)] for i in range(n,2*n)]
    return (A,B,C,D)
```

10. (a) Pour un produit de matrices de tailles $2^p \times 2^p$, la fonction `mul0` calcule 8 produits de matrices de tailles $2^{p-1} \times 2^{p-1}$, et on ne fait pas de multiplication supplémentaire de coefficients. D'où la relation de récurence $N_p = 8\,N_{p-1}$.

    (b) Pour un produit de matrices de tailles 1×1, `mul0` ne fait qu'une multiplication de coefficients. Donc $N_0 = 1$. D'où (avec la relation de récurrence) $\forall p, N_p = 8^p$.

11. Le programme de la partie II aurait donné $(2^p)^3$ ce qui revient au même. Les deux versions ont donc le même temps de calcul (en tout cas le même nombre de multiplications de coefficients).

Deuxième algorithme de type "Diviser pour mieux régner"

12. Montrons qu'on a bien $M_3 + M_5 = AF + BH$:

$$M_3 + M_5 = A \cdot (F - H) + (A + B) \cdot H = AF - AH + AH + BH = AF + BH$$

13. Montrons qu'on a bien $M_2 + M_4 = CE + DG$:

$$M_2 + M_4 = (C + D) \cdot E + D \cdot (G - E) = CE + DE + DG - DE = CE + DG$$

14. Nouvelle fonction `mul0` :

```
def mul0(M1,M2):
    if len(M1)==1 : #c'est à dire que k=0
        return [[M1[0][0]*M2[0][0]]]
    (A,B,C,D)=blocs(M1)
    (E,F,G,H)=blocs(M2)
    m1,m2,m3=mul0(A+D,E+H),mul0(C+D,E),mul0(A,F-H)
    m4,m5,m6=mul0(D,G-E),mul0(A+B,H),mul0(C-A,E+F)
    m7=mul0(B-D,G+H)
    return mat_blocs( m1+m4-m5+m7 ,  m3+m5,
                      m2+m4 ,       m1-m2+m3+m6  )
```

Remarque : Si on veut vraiment que le programme s'execute correctement en python, il aurait aussi fallu définir la somme de matrice (pour la version précédente de `mul0` aussi, d'ailleurs)

15. (a) Pour un produit de matrices de tailles $2^p \times 2^p$, cette nouvelle fonction `mul0` calcule 7 produits de matrices de tailles $2^{p-1} \times 2^{p-1}$, et on ne fait pas de multiplication supplémentaire de coefficients. D'où la relation de récurence $N'_p = 7\,N'_{p-1}$.

(b) Pour un produit de matrices de tailles 1×1, `mul0` ne fait qu'une multiplication de coefficients. Donc $N'_0 = 1$. D'où (avec la relation de récurrence) $\forall p, N'_p = 7^p$.

(c) Ce programme est plus rapide que les deux autres car 7^p est beaucoup plus petit que 8^p.