

Diviser pour Régner : Méthode de Strassen

1 Rappels sur les tableaux numpy

Une fois n'est pas coutume, on utilise les tableaux `numpy` (principalement pour leurs facilités de slicing).

Toutes les manipulations citées ci-dessous sont autorisées dans ce TP.

```
[1]: import numpy as np

      #rappel sur les tableaux numpy
      s=[[2, 14, 25, 30],\
        [3 ,15 ,28 ,30 ],\
        [7 ,15 ,32 ,43 ],\
        [20 ,28 ,36 ,58 ]]
```

```
[2]: #création de tableau numpy
      a=np.array(s)
      print(a)
```

```
[[ 2 14 25 30]
 [ 3 15 28 30]
 [ 7 15 32 43]
 [20 28 36 58]]
```

```
[3]: #slicing
      print(a[2:4,0:2])
```

```
[[ 7 15]
 [20 28]]
```

```
[4]: #somme
      print(a+a)
```

```
[[ 4 28 50 60]
 [ 6 30 56 60]
 [14 30 64 86]
 [40 56 72 116]]
```

```
[5]: #produit par un scalaire  
print(3*a)
```

```
[[ 6 42 75 90]  
 [ 9 45 84 90]  
 [21 45 96 129]  
 [60 84 108 174]]
```

```
[6]: #concaténations  
a = np.ones((2,2))  
b = 2 * np.ones((2,2))  
c = 3 * np.ones((2,2))  
print(c)
```

```
[[3. 3.]  
 [3. 3.]]
```

```
[7]: #concaténation horizontale  
print(np.concatenate((a,b,c),axis=1))
```

```
[[1. 1. 2. 2. 3. 3.]  
 [1. 1. 2. 2. 3. 3.]]
```

```
[8]: #concaténation verticale  
print(np.concatenate((a,b),axis=0))
```

```
[[1. 1.]  
 [1. 1.]  
 [2. 2.]  
 [2. 2.]]
```

```
[9]: #dimensions  
print(np.concatenate((a,b),axis=0).shape) #4 lignes deux colonnes
```

```
(4, 2)
```

L'opérateur * utilisé sur des tableaux effectue un produit coefficient par coefficient qui n'est pas le produit matriciel.

```
[10]: m = 2 * np.ones((2,2))  
print(m*m) #autorisé dans ce TP
```

```
[[4. 4.]  
 [4. 4.]]
```

Pour faire le produit de deux matrices **a** et **b** représentées par des tableaux, on peut utiliser le symbole **a@b** (mais ceci est **interdit** dans ce TP).

```
[11]: m = 2 * np.ones((2,2))
      print(m@m) #interdit dans ce TP
```

```
[[8. 8.]
 [8. 8.]]
```

2 Produit matriciel

Les matrices sont représentées par des tableaux `numpy`.

2.1 Algorithme naïf

2.1.1 Fonction de multiplication

Ecrire la fonction `mult(N,M)` qui calcule le produit NM par l'algorithme naïf.

```
[12]: *****
      def mult(N, M):
          # N: p x n, M: n x m (aucune vérification)
          assert len(N[0]), len(M)
          p, n, m = len(N), len(N[0]), len(M[0])

          C = np.zeros((m,p))
          for i in range(p):
              for j in range(m):
                  for k in range(n):
                      C[i,j] += N[i,k] * M[k,j]
          return C
```

```
[13]: a=np.array([[1,2,3],[4,5,6]])
      b=np.array([[1,-1],[2,2],[0,3]])
      print(mult(a,b))
```

```
[[ 5. 12.]
 [14. 24.]]
```

```
[14]: print(a@b) #pour vérifier
```

```
[[ 5 12]
 [14 24]]
```

2.1.2 Complexité

Etablir la complexité de l'algorithme naïf de produit matriciel.

```
*****
```

Triple boucle donc $O(npm)$ avec les conventions du code.

```
*****
```

2.2 Produit par blocs

Dans cette section, les matrices sont carrées de tailles $2^k \times 2^k$.

Peut-on améliorer la complexité du produit en adoptant une méthode « diviser pour régner » ?

Posons $M = \begin{pmatrix} A_1 & B_1 \\ C_1 & D_1 \end{pmatrix}$ et $N = \begin{pmatrix} A_2 & B_2 \\ C_2 & D_2 \end{pmatrix}$ où tous les blocs sont de taille $2^{k-1} \times 2^{k-1}$.

Rappel : Le produit MN , calculé par blocs, donne

$$MN = \begin{pmatrix} A_1A_2 + B_1C_2 & A_1B_2 + B_1D_2 \\ C_1A_2 + D_1C_2 & C_1B_2 + D_1D_2 \end{pmatrix}$$

Cela donne 8 produits et 4 additions.

2.2.1 Complexité du produit par bloc

Calculer la complexité du produit récursif par bloc.

L'algorithme conduit à une récurrence de la forme

$$C(n) = 8C(\lceil n/2 \rceil) + O(n^2)$$

On a l'habitude. Posons $n = 2^k$. Alors :

$$\begin{aligned} C(k) &= 8C(k-1) + 4^k = 8(8 \times C(k-2) + 4^{k-1}) + 4^k \\ &= 8^3C(k-3) + 8^0 \times 4^k + 8^1 \times 4^{k-1} + 8^2 \times 4^{k-2} = \dots \\ &= 8^k C(0) + 8^k \sum_{i=0}^{k-1} \left(\frac{4}{8}\right)^i \\ &\leq 8^k + 8^k \frac{\left(\frac{1}{2}\right)^k - 1}{-\frac{1}{2}} = O(8^k) = O(n^{\log_2(8)}) = O(n^3) \end{aligned}$$

C'est la complexité de l'algorithme naïf.

2.3 Algorithme de Strassen

Il correspond aux formules suivantes (qu'on laisse vérifier au lecteur curieux) :

$$\begin{aligned} X &= M_1 + M_2 - M_4 + M_6 \\ Y &= M_4 + M_5 \\ Z &= M_6 + M_7 \\ T &= M_2 - M_3 + M_5 - M_7 \end{aligned}$$

où les blocs M_i sont définis par

$$\begin{aligned} M_1 &= (B_1 - D_1)(C_2 + D_2) & M_5 &= A_1(B_2 - D_2) \\ M_2 &= (A_1 + D_1)(A_2 + D_2) & M_6 &= D_1(C_2 - A_2) \\ M_3 &= (A_1 - C_1)(A_2 + B_2) & M_7 &= (C_1 + D_1)A_2 \\ M_4 &= (A_1 + B_1)D_2 \end{aligned}$$

Dans ce cas on a $MN = \begin{pmatrix} X & Y \\ Z & T \end{pmatrix}$.

2.3.1 Blocs

Ecrire la fonction `blocs(a)` qui prend en paramètre une matrice représentée par un tableau de taille (supposée) $2^k \times 2^k$ et la décompose en 4 blocs.

```
[15]: def blocs(M):  
      n = len(M)  
      return M[:n//2, :n//2], M[:n//2, n//2:], \  
             M[n//2:, :n//2], M[n//2:, n//2:]
```

```
[16]: a = np.array([[j*4 + i for i in range(4)] for j in range(4)])  
      print(a)
```

```
[[ 0  1  2  3]  
 [ 4  5  6  7]  
 [ 8  9 10 11]  
 [12 13 14 15]]
```

```
[17]: blcs = blocs(a)  
      for bl in blcs:  
          print(bl, end="\n-----\n")
```

```
[[0 1]  
 [4 5]]  
-----  
[[2 3]  
 [6 7]]  
-----  
[[ 8  9]  
 [12 13]]  
-----  
[[10 11]  
 [14 15]]  
-----
```

2.3.2 Code

Ecrire la fonction récursive

```
def strassen(M,N)
```

qui implante cet algorithme pour des matrices de taille $2^k \times 2^k$.

On peut utiliser les fonctionnalités des tableaux numpy pour le slicing et la somme. Mais le produit des matrices numpy est bien entendu interdit ici.

```
[18]: *****  
      def strassen(M,N):  
          if len(M)==1:  
              return np.array([[M[0,0]*N[0,0]]])
```

```

A_1,B_1,C_1,D_1=blocs(M)
A_2,B_2,C_2,D_2=blocs(N)
M_1=strassen((B_1 - D_1 ),(C_2 + D_2 ))
M_5=strassen(A_1,(B_2-D_2))
M_2=strassen((A_1 + D_1 ),(A_2 + D_2 ))
M_6=strassen(D_1,(C_2-A_2))
M_3=strassen((A_1 - C_1 ),(A_2 +B_2))
M_7=strassen((C_1+D_1),A_2)
M_4=strassen((A_1+B_1),D_2)
X=M_1+M_2-M_4+M_6
Y=M_4+M_5
Z=M_6+M_7
T=M_2-M_3+M_5-M_7
AA = np.concatenate((X,Y),axis=1)
BB = np.concatenate((Z,T),axis=1)
CC = np.concatenate((AA,BB),axis=0)
return CC

```

```

[19]: a = np.array([[j*4 + i for i in range(4)] for j in range(4)])
      print(strassen(a,a))

```

```

[[ 56  62  68  74]
 [152 174 196 218]
 [248 286 324 362]
 [344 398 452 506]]

```

```

[20]: print(a@a) #pour comparaison

```

```

[[ 56  62  68  74]
 [152 174 196 218]
 [248 286 324 362]
 [344 398 452 506]]

```

2.3.3 Complexité

Etablir la complexité de la méthode récursive de Strassen.

Avec $n = 2^k$, on a une complexité de la forme $C(k) = 7C(k-1) + O(n^2)$ où $O(n^2)$ représente les différentes additions entre matrices, les concaténations et l'extraction des blocs. Simplifions en $C(k) = 7C(k-1) + 4^k$.

Alors

$$\begin{aligned}
C(k) &= 7C(k-1) + 4^k = 7(7 \times C(k-2) + 2^{k-1}) + 4^k \\
&= 7^3 C(k-3) + 7^0 \times 4^k + 7^1 \times 4^{k-1} + 7^2 \times 4^{k-2} = \dots \\
&= 7^k C(0) + 7^k \sum_{i=0}^{k-1} \left(\frac{4}{7}\right)^i \\
&\leq 7^k + 7^k \frac{\left(\frac{4}{7}\right)^k - 1}{-\frac{5}{7}} = O(7^k) = O(n^{\log_2(7)}) \simeq O(n^{2.8})
\end{aligned}$$

C'est donc mieux que l'algorithme naïf qui est en $O(n^3)$.

La complexité est croissante en la taille de la donnée (admis). Or on peut toujours encadrer un nombre n par deux puissances de 2 consécutives. La complexité pour n quelconque est donc aussi en $O(n^{2.8})$
