2019/2020

Projet de complexité : Produit Matriciel

BANDOUI NAZIM 161631055599
MEKBAL MOHAMED AMINE 161631055599

En mathématiques, le produit matriciel est une opération binaire qui produit une matrice à partir de deux matrices. Pour le produit de matrices, il faut que le nombre de colonnes de la première matrice soit égal au nombre de ligne de la deuxième matrice, le produit aura le nombre de lignes de la première et le nombre de colonnes de la 2eme.

$$\mathrm{M}[x,y]N[y,z] = P[x,z]$$

I – Produit matriciel classique :

I.1 – Définition :

A est une matrice x × y

B est une matrice y × z

Leur produit C noté C=AB :

$$c_{i,j} = \sum_{k=1}^{n} a_{ik}b_{kj} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj}$$

I.2 – Algorithme, Implémentation et complexité :

Algorithme :

A : tableau [x,y] ;

B : tableau [y,z] ;

Début

    /*création d'une matrice nulle C[x,z]*/

    Pour i de 0 a x-1 :

    Début

        Pour j de 0 a x-1 :

        Début

            C[i] [j]=0 ;

        Fin.

    Fin.

Pour i de 0 à x-1 :

Début

Pour j de 0 à z-1 :

Début

Pour j de 0 à y-1 :

Début

C[i] [j]=c[i] [j] +a[i] [k]*b[k] [j] ;

Fin

Fin

Fin

Fin.

Implémentation :

Solution implémentée en Python 3 :

```python
def mul1(a,b):
    m=len(a)
    p=len(a[0])
    n=len(b[0])
    result=np.zeros((m,n),dtype=int)
    for i in range(0,m):
        for j in range(0,n):
            for k in range(0,p):
                result[i][j]=result[i][j]+a[i][k]*b[k][j]
    return result
```

Complexité :

Etant donné que l'algorithme contient une boucle imbriquée qui est imbriquée dans une boucle, l'algorithme a une complexité $\Theta(n^3)$.

II – Produit de matrices par blocs:

II.1 – Définition :

Un produit de matrices par blocs peut être effectué en considérant seulement des opérations sur les sous-matrices.

Dans ce projet, nous allons diviser notre matrice en 4 blocs.

Soit A et B deux matrices de taille n× n :

$$A = \begin{bmatrix} x_0 & \cdots & x_n \\ \vdots & \ddots & \vdots \\ x_n & \cdots & x_{nk} \end{bmatrix} = \begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix}$$

$$B = \begin{bmatrix} y_0 & \cdots & y_k \\ \vdots & \ddots & \vdots \\ y_n & \cdots & y_{nk} \end{bmatrix} = \begin{bmatrix} b_1 & b_2 \\ b_3 & b_4 \end{bmatrix}$$

Ou a_i et b_i est une matrice $n/2 \times n/2$.

Leur produit C=AB :

$$C = \begin{bmatrix} c_1 & c_2 \\ c_3 & c_4 \end{bmatrix}$$

Ou :

$c1 = a_1 b_1 + a_2 b_3$

$c2 = a_1 b_2 + a_2 b_4$

$c3 = a_3 b_1 + a_4 b_3$

$c4 = a_3 b_2 + a_4 b_4$

II.2 – Algorithme, Implémentation et complexité :

Algorithme :

Tableau MultiplicationParBloc (A : tableau [x,y] , B : tableau [y,z])

Début

Si (x<=2 ou y<=2)

Alors

Retourner AB ;

Fin Si.

$c1 = \text{MultiplicationParBloc}(a_1, b_1) + \text{MultiplicationParBloc}(a_2, b_3)$

$c2 = \text{MultiplicationParBloc}(a_1, b_2) + \text{MultiplicationParBloc}(a_2, b_4)$

$c3 = \text{MultiplicationParBloc}(a_3, b_1) + \text{MultiplicationParBloc}(a_4, b_3)$

$c4 = \text{MultiplicationParBloc}(a_3, b_2) + \text{MultiplicationParBloc}(a_4, b_4)$

/* recover (a, b, c, d)=$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ */

Retourner recover (c1, c2, c3, c4) ;

Fin.

Implémentation :

Solution implémentée en Python 3 :

```
def mul2(a,b):
    if(len(a)<=2 and len(a[0])<=2):
        return np.matmul(a,b)
    wa=xa=ya=za=[]
    wb=xb=yb=zb=[]
    wc=xc=yc=zc=[]
    wa,xa,ya,za=div4(a)
    wb,xb,yb,zb=div4(b)
    wc=np.matmul(wa,wb)+np.matmul(xa,yb)
    xc=np.matmul(wa,xb)+np.matmul(xa,zb)
    yc=np.matmul(ya,wb)+np.matmul(za,yb)
    zc=np.matmul(ya,xb)+np.matmul(za,zb)
    return mul2(recover4(wc,xc,yc,zc))
```

Complexité :

Le temps d'une addition : $\Theta (n^2)$

Le temps d'une multiplication : T (n/2)

Pour chaque itération, nous avons 4 additions et 8 multiplications

$T_{somme}=4 \Theta ((n/2)^2)= \Theta (4n^2/4)= \Theta (n^2)$

$T_{multiplication}=8 T (n/2)$

$$\mathbf{T(n)= T_{somme} + T_{multiplication} = \Theta (n^3)}$$

III – Algorithme de Strassen:

III.1 – Définition :

L'algorithme de Strassen est un algorithme calculant le produit de deux matrices carrées de taille n, proposé par Volker Strassen en 19691. La complexité de l'algorithme est en O ($n^{2,807}$) avec pour la première fois un exposant inférieur à celui de la multiplication naïve qui est en O (n^3). Par contre, il a l'inconvénient de ne pas être stable.

Soit A et B deux matrices de taille n× n :

$$A = \begin{bmatrix} x_0 & \cdots & x_n \\ \vdots & \ddots & \vdots \\ x_n & \cdots & x_{nk} \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

$$B = \begin{bmatrix} y_0 & \cdots & y_k \\ \vdots & \ddots & \vdots \\ y_n & \cdots & y_{nk} \end{bmatrix} = \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

Ou a_i et b_i est une matrice $n/2 \times n/2$.

Leur produit C=AB :

$$C = \begin{bmatrix} c_1 & c_2 \\ c_3 & c_4 \end{bmatrix}$$

Ou :

p1= a*(f-h)

p2= (a+b)*h

p3= (c+d)*e

p4= d*(g-e)

p5= (a+d)*(e+h)

p6= (b-d)*(g+h)

p7= (a-c)*(e+f)

$c1 =$ p1 + p4 – p5 + p7

$c2 = p3 + p5$

$c3 = p2 + p4$

$c4 = p1 - p2 + p3 + p6$

III.2 – Algorithme, Implémentation et complexité :

Algorithme :

```
Tableau StrassenProduit (A : tableau [x,y]  , B : tableau [y,z] )
Début
    Si (x<=2 ou y<=2)
    Alors
        Retourner AB ;
    Fin Si.
    p1= StrassenProduit (a,f-h) ;
    p2= StrassenProduit (a+b,h) ;
    p3= StrassenProduit(c+d,e) ;
    p4= StrassenProduit (d,g-e) ;
    p5= StrassenProduit (a+d,e+h) ;
    p6= StrassenProduit (b-d,g+h) ;
    p7= StrassenProduit (a-c,e+f) ;

    c1 = p1 + p4 – p5 + p7 ;
    c2 = p3 + p5
    c3 = p2 + p4
    c4 = p1 − p2 + p3 + p6
    /* recover (a, b, c, d)=[a  b; c  d] */
    Retourner recover (c1, c2, c3, c4) ;

Fin.
```

Implémentation :

Solution implémentée en Python 3 :

```
def mul3(aa,bb):
    if(len(aa)<=2 and len(aa[0])<=2):
        return np.matmul(aa,bb)
    a,b,c,d=div4(aa)
    e,f,g,h=div4(bb)

    return recover4(mul3(b-d,g+h)+mul3(a+d,e+h)+mul3(d,g-e)-
mul3(a+b,h),mul3(a,f-h)+mul3(a+b,h),mul3(c+d,e)+mul3(d,g-e),mul3(a,f-
h)+mul3(a+d,e+h)-mul3(c+d,e)-mul3(a-c,e+f))
```

Complexité :

$T_{somme}=4\ \Theta\ ((n/2)^2)= \Theta\ (4n^2/4)= \Theta\ (n^2)$

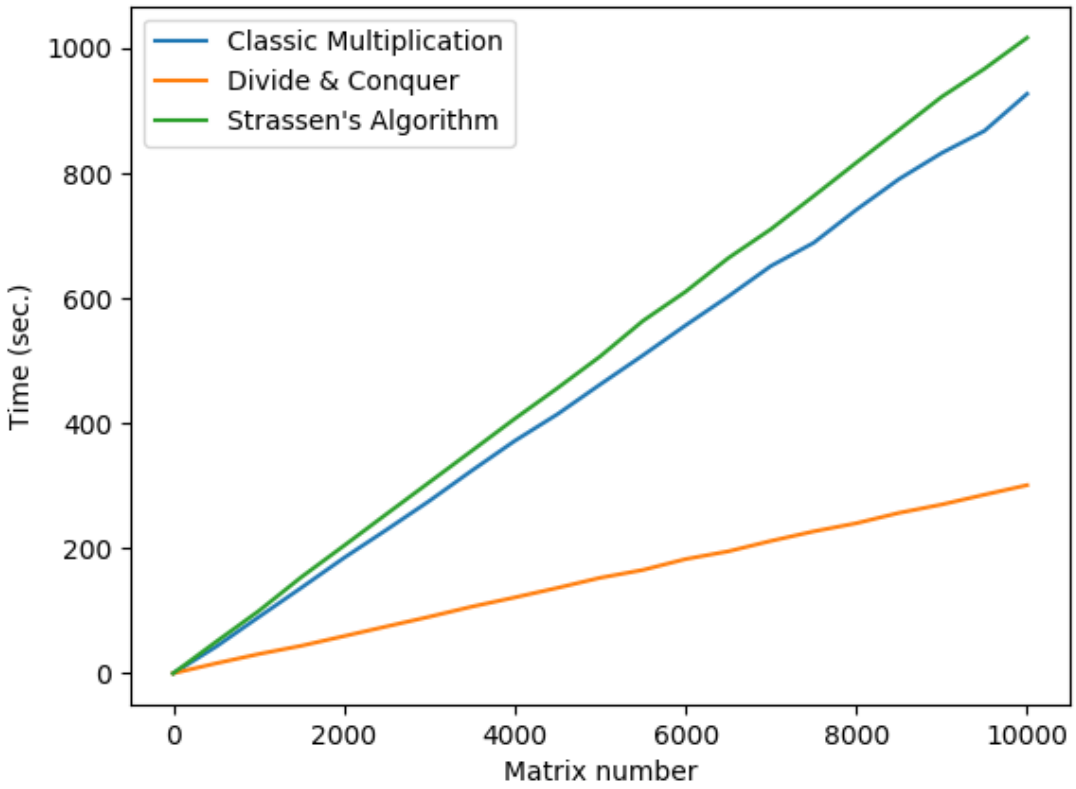
$T_{multiplication}=7\ T\ (n/2)$

$T(n)= T_{somme} + T_{multiplication} = \Theta\ (n^{\log_2 7})$

$2.8<\log_2 7<2.81$

$$\mathbf{T(n)= \Theta\ (n^{2.81})}$$

IV– Comparaison entre les 3 méthodes :



-Analyse :

Le graphe ci-dessous représente la variation du temps d'exécution en millisecondes en fonction du changement du nombre de multiplication effectuées par le produit matriciel classique, le produit matriciel par blocs et l'algorithme de Strassen.

Nous remarquons une croissance du temps d'exécution pour le produit matriciel par blocs atteignant près de 300 secondes (5min.) pour 10000 multiplications.

Nous remarquons aussi une **forte** croissance du temps d'exécution pour le produit classique et l'algorithme de Strassen atteignant 900 et 1000 secondes (15min. et 16.66min) (respectivement) pour 10000 multiplications. Les courbes des deux méthodes convergent entre elles.

Nous remarquons que le produit de matrices par bloc est beaucoup plus rapide que l'algorithme de Strassen malgré une pire complexité. Cela est dû au fait que l'algorithme de Strassen a une meilleur complexité en théorie seulement, en pratique beaucoup de temps est perdu pour plusieurs raisons ce qui rends son temps d'exécution quasi équivalent à la méthode classique.

Pour le cas d'une matrice n × n tel que : $\mathbf{n \neq 2^k}$

Il suffit de remplacer n par n' qui est le plus petit $\mathbf{2^k}$ suivant et de mettre des zéros dans les emplacements vide de la matrice n' × n' (création d'une matrice creuse)

Exemple :

A est une matrice de taille n × n ou n=5 :

On remplace 5 par le $\mathbf{2^k}$ suivant et donc **n=8**

	1 2 3 4 5			**1 2 3 4 5 0 0 0**
	6 7 8 9 1			**6 7 8 9 1 0 0 0**
A=	**2 3 4 5 6**	**Devient**	**A=**	**2 3 4 5 6 0 0 0**
	7 8 9 1 2			**7 8 9 1 2 0 0 0**
	3 4 5 6 7			**3 4 5 6 7 0 0 0**
				0 0 0 0 0 0 0 0
				0 0 0 0 0 0 0 0
				0 0 0 0 0 0 0 0

<u>Test et impact :</u>

Nous allons effectuer des tests sur différentes taille de matrices et analyser les résultats, nous exécuterons dans chaque cas un maximum de 2000 instructions.

Les graphes ci-dessous représentent la variation du temps d'exécution en millisecondes en fonction du changement du nombre de multiplication effectuées par le produit matriciel classique, le produit matriciel par blocs et l'algorithme de Strassen.

Figure 1 : nous avons pris des matrices 5x5 qui seront transformer en matrices 8x8 comme montrer précédemment.

Nous remarquons une croissance du temps d'exécution pour le produit matriciel par blocs atteignant 10 secondes pour 2000 multiplications.

Nous remarquons aussi une **forte** croissance du temps d'exécution pour le produit classique et l'algorithme de Strassen atteignant 45 et 50 secondes (respectivement) pour 2000 multiplications. Les courbes des deux méthodes convergent entre elles.

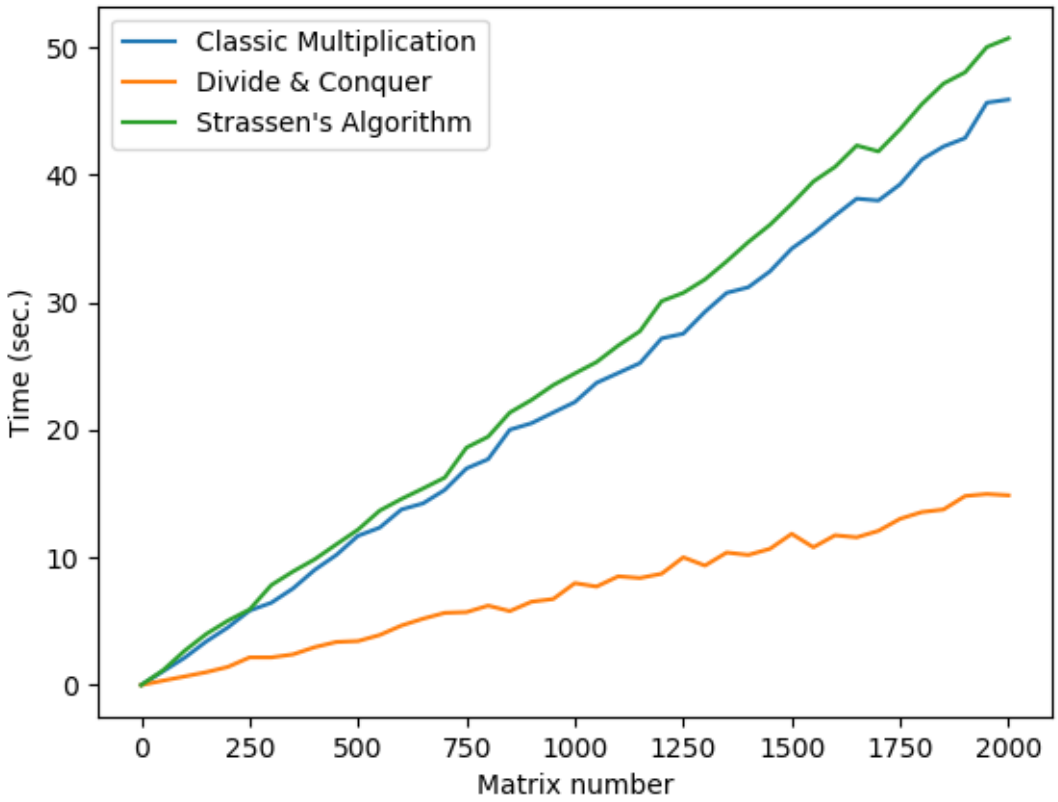


Figure 1

Figure 2 : nous avons pris des matrices 5x5 qui seront transformé en matrices 8x8 comme montrer précédemment.

Nous remarquons une croissance du temps d'exécution pour le produit matriciel par blocs atteignant 50 secondes pour 2000 multiplications.

Nous remarquons aussi une **forte** croissance du temps d'exécution pour le produit classique et l'algorithme de Strassen atteignant 175 et 200 secondes (respectivement) pour 2000 multiplications. Les courbes des deux méthodes convergent entre elles.

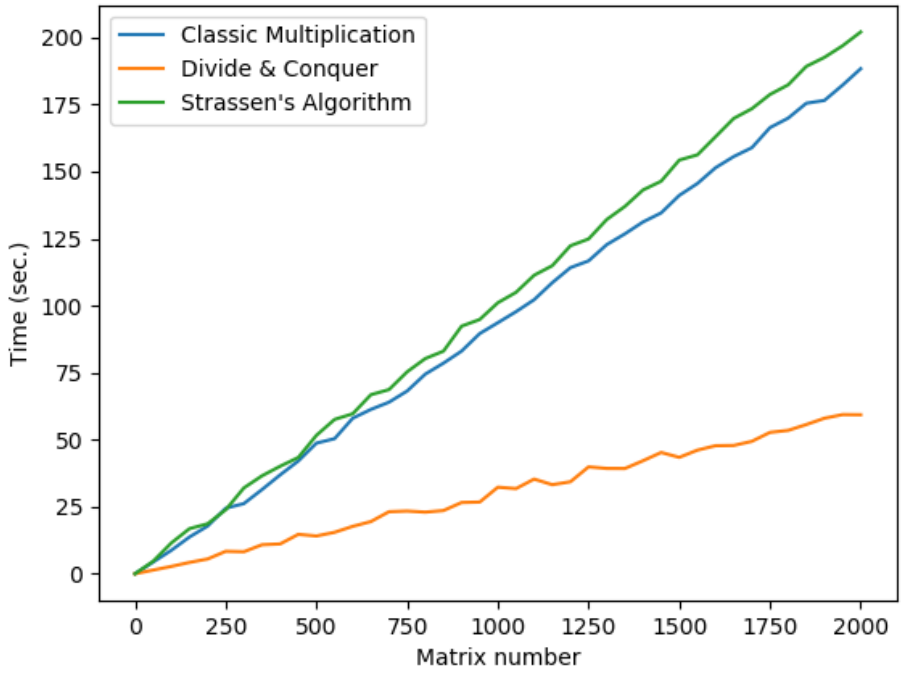


Figure 2

<u>Conclusion : Le produit par blocs est mieux adapté aux matrices creuses que l'algorithme de Strassen.</u>

Analysons maintenant le taux de croissance du temps additionnel :

Figure 3 : nous avons pris des matrices 9x9 qui seront transformé en matrices 16x16 comme montrer précédemment.

Nous remarquons une croissance du temps d'exécution pour le produit matriciel par blocs atteignant un peu moins de 200 secondes pour 2000 multiplications.

Nous remarquons une croissance du temps d'exécution pour le produit matriciel par blocs atteignant 500 secondes pour 2000 multiplications.

Nous remarquons une **forte** croissance du temps d'exécution pour le produit matriciel par blocs atteignant 900 secondes pour 2000 multiplications.

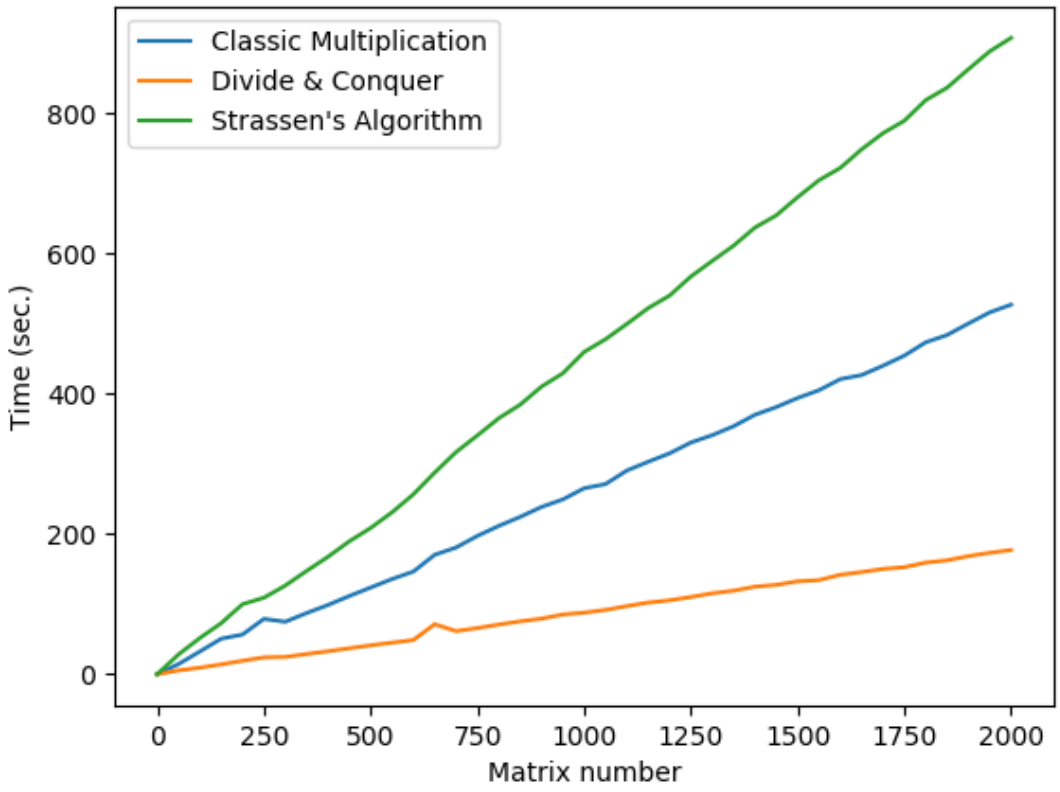


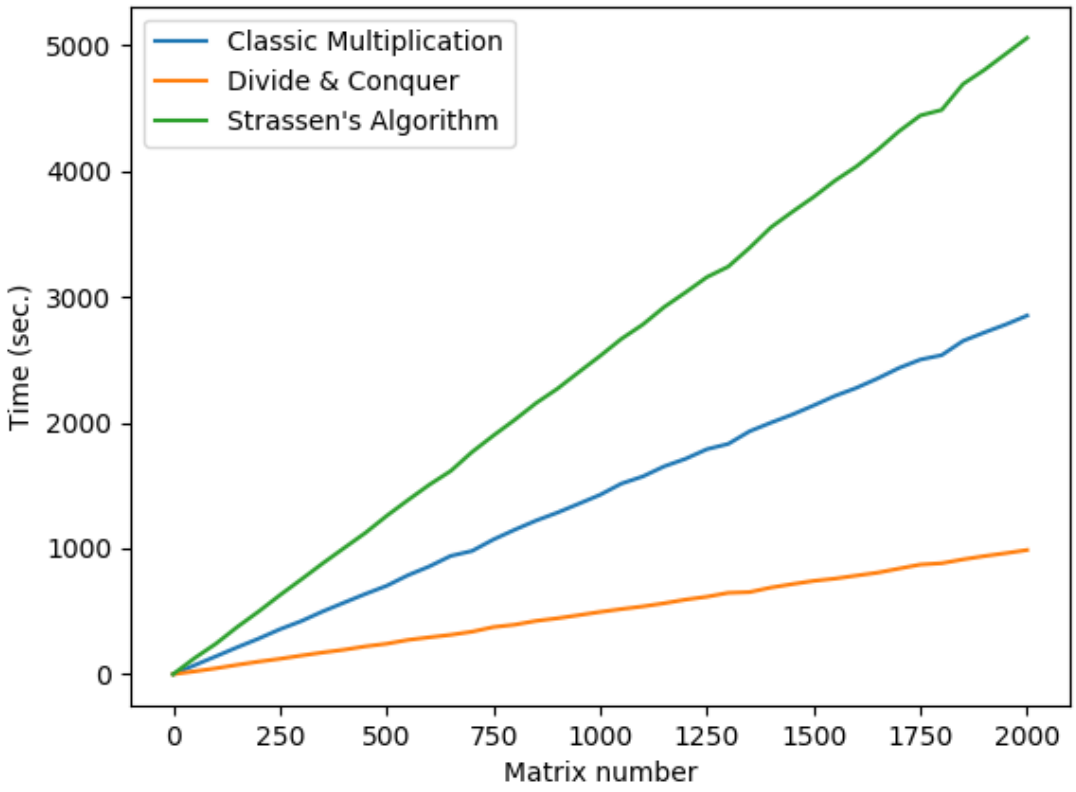
Figure 3

Figure 4 : nous avons pris des matrices 16x16

Nous remarquons une croissance du temps d'exécution pour le produit matriciel par blocs atteignant un peu moins de 900 secondes pour 2000 multiplications.

Nous remarquons une croissance du temps d'exécution pour le produit matriciel par blocs atteignant 3700 secondes pour 2000 multiplications.

Nous remarquons une **forte** croissance du temps d'exécution pour le produit matriciel par blocs atteignant 5100 secondes pour 2000 multiplications.



Ceci confirme la conclusion précédente.