SCIENCES SORBONNE UNIVERSITÉ

Compte-rendu du projet de FLAG

Jacques Colin et Clarke Zhou

Année 2021-2022

Sommaire

I - Introduction

Afin de pouvoir inverser toutes les matrices, nous avons choisi d'implémenter PLUQ au lieu de LU. Nous avons également réussi à implémenter l'inversion de matrice via l'algorithme de Strassen. Tous les algorithmes sont fonctionnels pour toutes les tailles de matrice.

De plus, nous avons implémenté l'algorithme de multiplication de Strassen afin de comparer les performances avec la multiplication naïve pour l'algorithme de Strassen.

Nous avons également implémenté dans la version de l'algorithme d'inversion de Strassen optimisé, une condition permettant à l'algorithme de choisir entre la multiplication de Strassen et la multiplication naïve afin de maximiser les performances.

Afin de pouvoir inverser toutes les matrices en appelant Strassen nous avons rajouté une condition qui teste si la matrice est paire ou non, si elle ne l'est pas alors Strassen va faire appel à PLUQ.

II - Code

II.1 - Arborescence des fichiers

Projet
- ↳main.c
- ↳benchmark.c
- ↳unit_test.c
- ↳tests.c
- ↳algo.c
- ↳base.c
- ↳matrix.c

Le fichier *main.c* fait appel aux fichiers *benchmark.c* et *unit_test.c* afin d'exécuter les différents types de tests. Il n'est pas nécessaire de modifier ce fichier afin de choisir les tests à exécuter.
Le fichier *benchmark.c* implémente des fonctions permettant de comparer le temps d'exécution des différents algorithmes.
Le fichier *unit_test.c* fait appel au fichier *tests.c* afin de pouvoir facilement commenter/décommenter les différents tests à exécuter.
Le fichier *tests.c* implémente l'ensemble des tests.
Le fichier *algo.c* implémente les différents types d'algorithmes demandés ainsi que des sous fonctions permettant de faire fonctionner ces algorithmes.
Le fichier *base.c* contient les opérations de base telles que l'addition, la multiplication, l'inverse modulaire.

Le fichier *matrix.c* permet la gestion d'une structure "Matrix" ainsi que l'utilisation d'opérations matricielles telles que l'addition ou la multiplication.

II.2 - Compilation et exécution du code

II.2.a - Benchmarks

Afin de compiler le programme il faut utiliser la commande:

```
$ make
```

puis lancer l'exécutable en faisant la commande:

```
./main [-prime p] [-size s] [-iteration i]
```

ou

```
./main [-p pr] [-s sz] [-i it]
```

Si aucun argument n'est mis, ou si des arguments sont manquants alors les valeurs par défaut seront :

- prime : 1069639009
- size : 128
- iteration : 1

Exemple:

```
./main -p 65537 -s 100 -i 20
./main -p 11 -s 15
./main -s 20
```

Si vous avez un doute, vous pouvez utiliser la commande :

```
$ ./main -h
```

ou

```
$ ./main -help
```

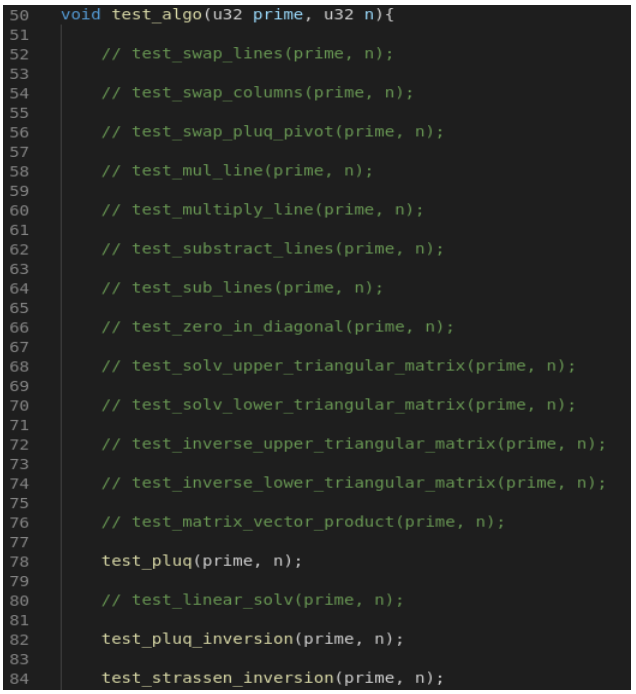
Une fois exécuté, le programme vous demandera de choisir quel benchmark lancer
Comme le montre l'image ci-dessous :

```
21101162@ppti-14-507-01:~/Documents/S2/FLAG/Projet$ ./main -p 65537 -s 64 -i 5

--- Initialisation ---
Nombre premier : 65537
Taille de la matrice : 64
Nombre d'iteration(s) : 5

--- Choix du benchmark ---
 1 - Comparaison entre la multiplication naive et la multiplication de Strassen
 2 - Comparaison entre l'inversion PLUQ et l'inversion de Strassen
 3 - Comparaison des algorithmes d'inversions de Strassen utilisant différent algorithme de multiplication
       => Multiplication naive
       => Multiplication de Strassen
       => Multiplication naive et Strassen (choisit l'un ou l'autre selon la taille de la matrice)
 4 - Temps d'execution de l'algorithme PLUQ
 5 - Quitter
```

Pour changer les arguments, il suffit de quitter et recommencer l'étape précédente.

II.2.b - Tests unitaires

Pour tester les fonctions une par une, il faut décommenter les commentaires dans le fichier unit_test.c, comme le montre l'image ci-dessous :



```
50  void test_algo(u32 prime, u32 n){
51
52      // test_swap_lines(prime, n);
53
54      // test_swap_columns(prime, n);
55
56      // test_swap_pluq_pivot(prime, n);
57
58      // test_mul_line(prime, n);
59
60      // test_multiply_line(prime, n);
61
62      // test_substract_lines(prime, n);
63
64      // test_sub_lines(prime, n);
65
66      // test_zero_in_diagonal(prime, n);
67
68      // test_solv_upper_triangular_matrix(prime, n);
69
70      // test_solv_lower_triangular_matrix(prime, n);
71
72      // test_inverse_upper_triangular_matrix(prime, n);
73
74      // test_inverse_lower_triangular_matrix(prime, n);
75
76      // test_matrix_vector_product(prime, n);
77
78      test_pluq(prime, n);
79
80      // test_linear_solv(prime, n);
81
82      test_pluq_inversion(prime, n);
83
84      test_strassen_inversion(prime, n);
```

Vous pouvez sélectionner plusieurs tests à la fois, mais cela a plus de chances d'échouer. La génération des matrices est aléatoire, cela augmente la probabilité que la matrice générée ne soit pas inversible et fasse échouer les tests. Il faudra alors relancer le programme.
Nous vous conseillons de choisir des matrices de petite taille afin que l'affichage soit plus lisible.

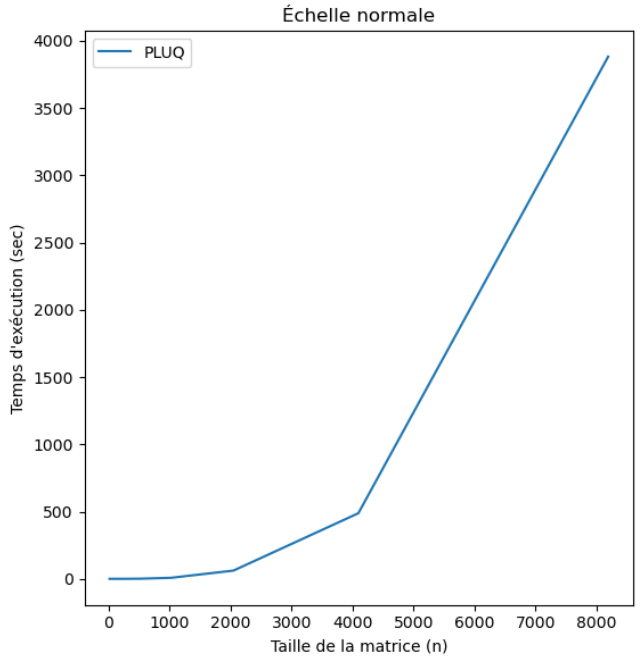
III - Benchmarks

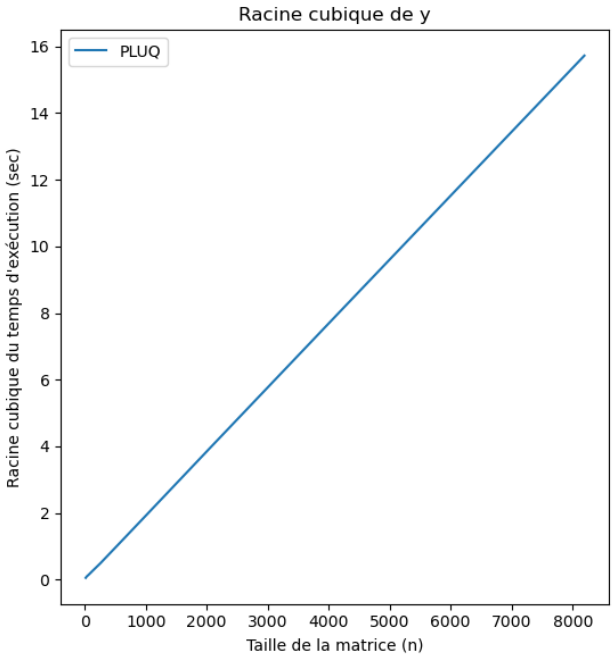
Les tests ont été effectués sur les machines de la PPTI avec le nombre premier 1 069 639 009.

III.1 - Décomposition PLUQ

n	PLUQ (sec)
16	0.000211
32	0.000986
64	0.003539
128	0.019378
256	0.122064
512	0.956809
1024	7.630618
2048	61.241443
4096	487.776084
8192	3882.437955
Complexité	$O(n^3)$

Temps d'exécution de la décomposition PLUQ par rapport à la taille de la matrice





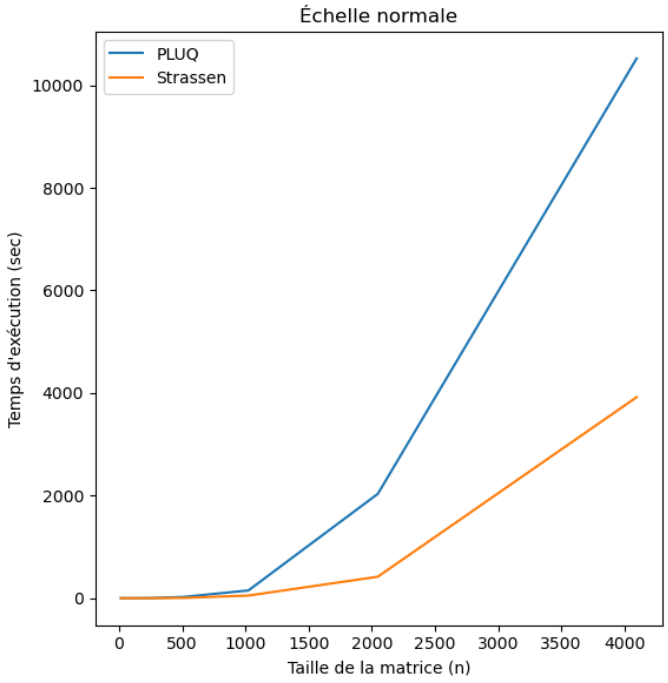
La complexité théorique de l'algorithme PLUQ est de $O(n^3)$. Lorsque la taille de la matrice double, le temps d'exécution est multiplié par 8 car $\frac{(2n)^3}{n^3} = \frac{8n^3}{n^3} = 8$

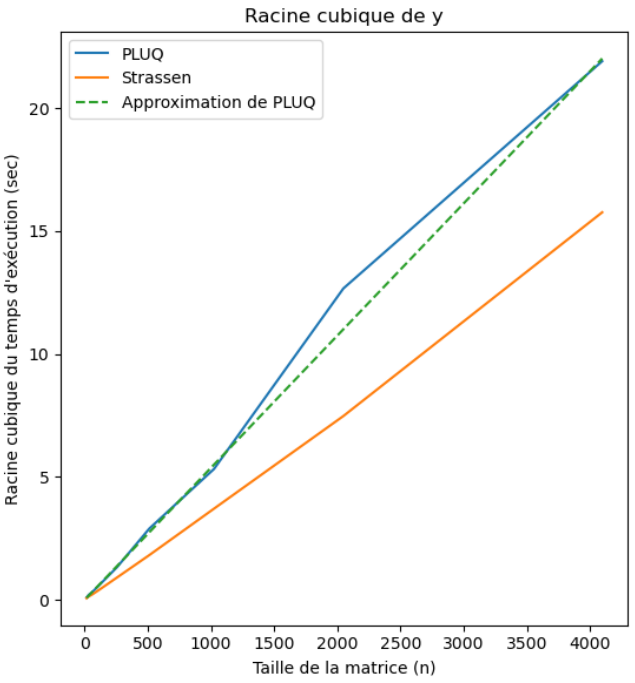
En pratique on trouve aussi que le temps d'exécution est multiplié par 8 quand la taille de la matrice double. La complexité théorique est donc vérifiée.

III.2 - Inverse modulaire : PLUQ vs Strassen

n	PLUQ (sec)	Strassen multiplication naïve (sec)
16	0.001064	0.000385
32	0.008710	0.002716
64	0.038638	0.013127
128	0.295928	0.095926
256	2.359965	0.759839
512	24.485409	6.079963
1024	151.202569	51.243688
2048	2034.221337	418.822340
4096	10521.903457	3919.305271
Complexité	$O(n^3)$	$O(n^3)$

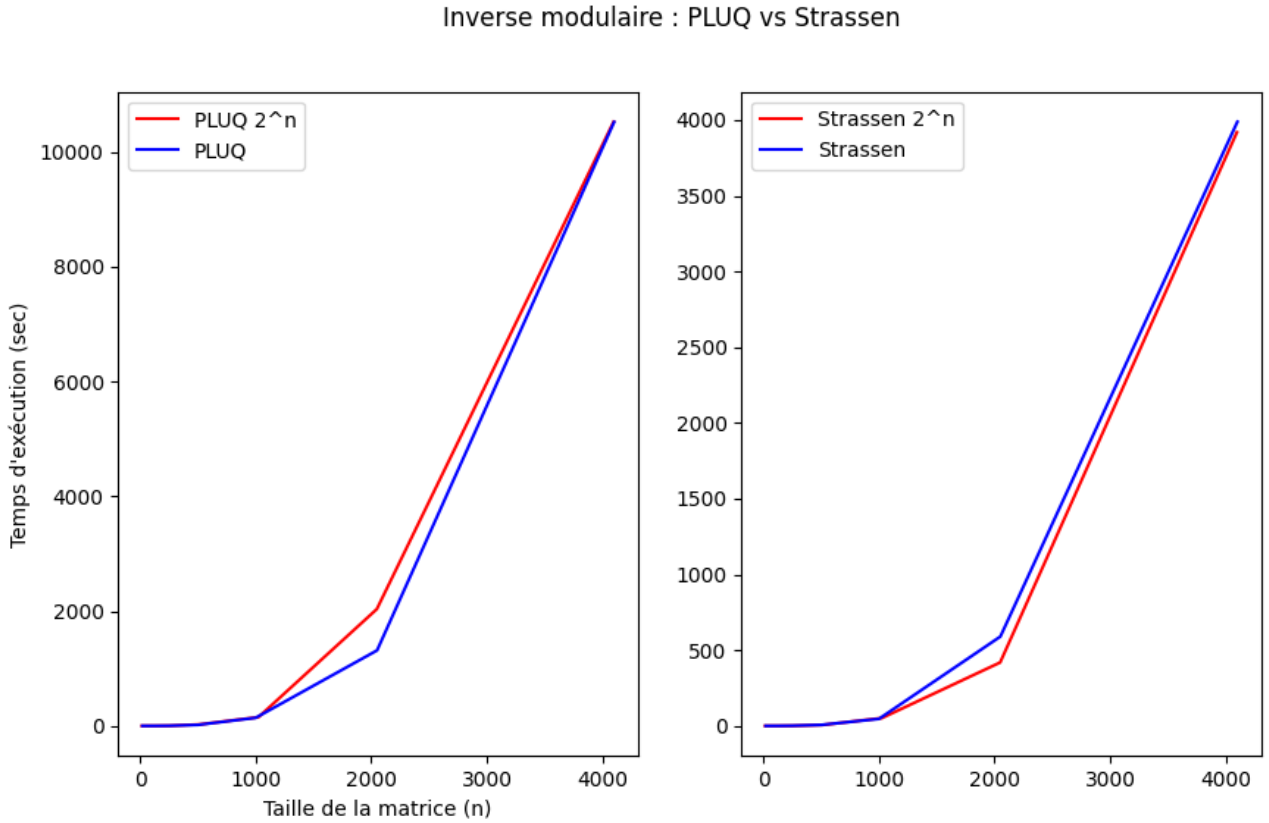
Temps d'exécution de l'inverse modulaire PLUQ vs Strassen par rapport à la taille de la matrice

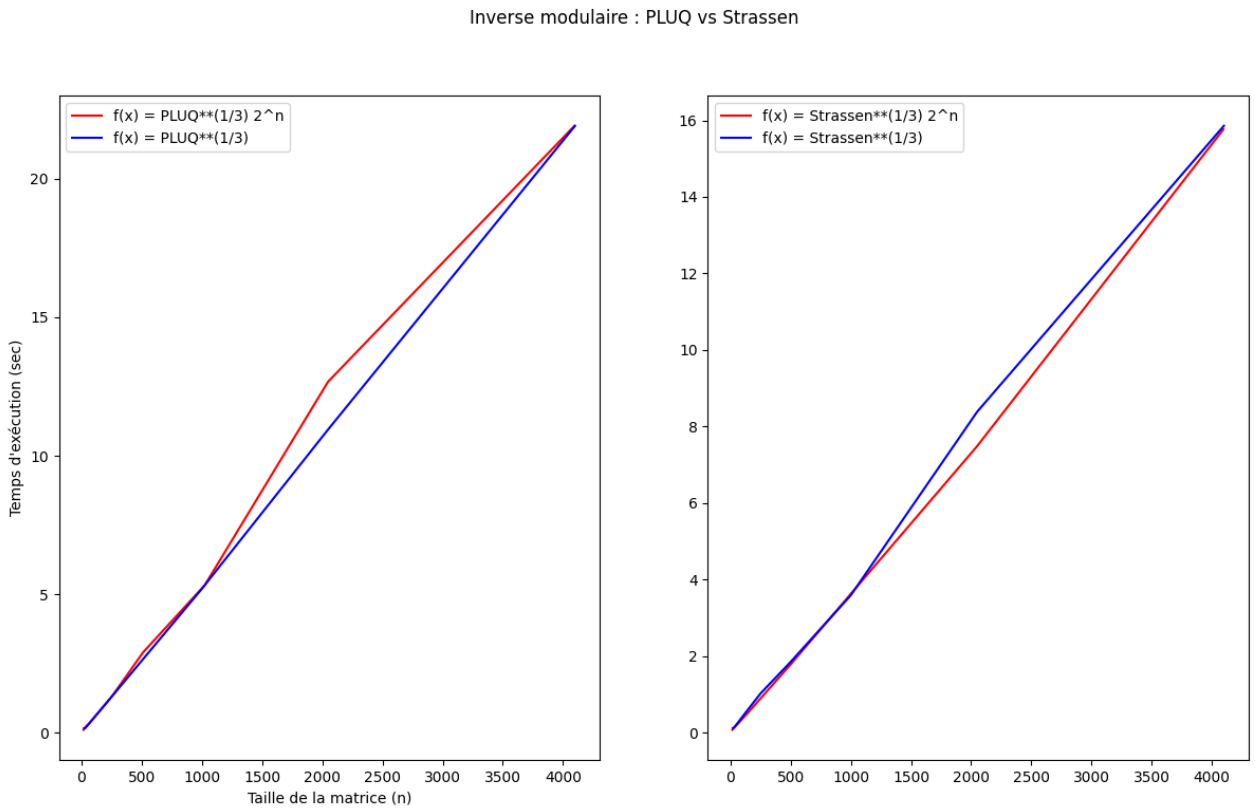




Interprétation : Pour les matrices de taille 2^n, on remarque que l'inversion de Strassen est toujours plus efficace que l'inversion en utilisant la décomposition PLUQ.

n	PLUQ (sec)	Strassen multiplication naïve (sec)
20	0.003581	0.001579
30	0.003977	0.002083
70	0.048803	0.025511
130	0.306964	0.156228
250	2.181757	1.071181
500	17.511983	6.309771
1000	140.266764	46.214870
2050	1314.749193	589.692217
4100	10517.119411	3988.598750
Complexité	$O(n^3)$	$O(n^3)$





A l'aide du graphe nous pouvons constater que la complexité est inchangée peu importe la taille de la matrice, pour les deux algorithmes.

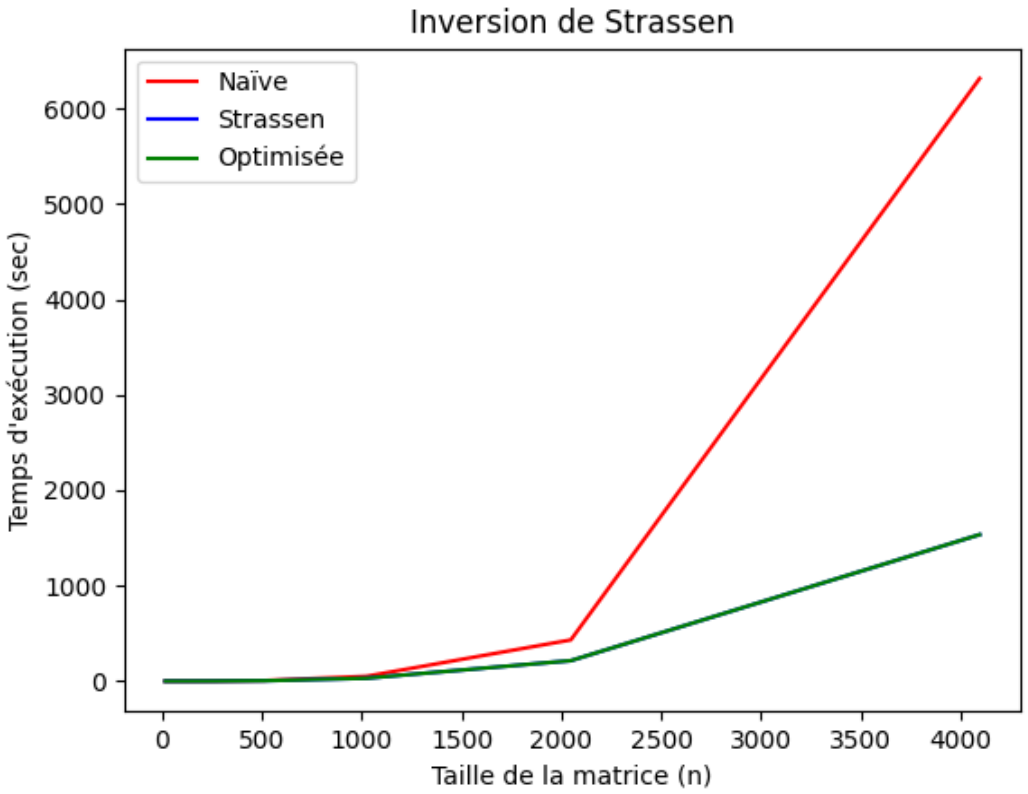
III.3 - Inverse modulaire : Strassen

Comparaison des différentes inversions modulaires de Strassen.

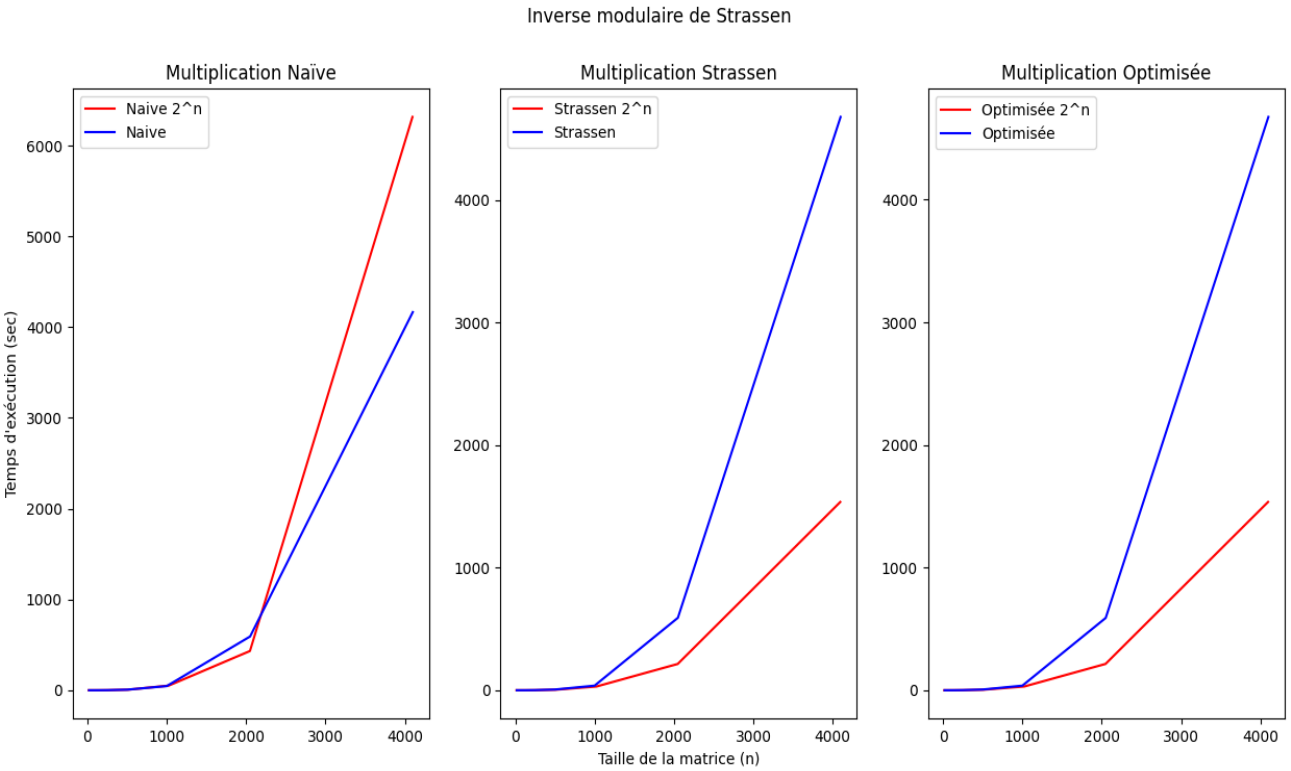
n	Strassen multiplication naïve (sec)	Strassen multiplication Strassen (sec)	Strassen multiplication optimisée (sec)
16	0.000393	0.000398	0.000398
32	0.002486	0.002478	0.001998
64	0.012750	0.011803	0.011969
128	0.094852	0.085005	0.093832
256	0.755116	0.606513	0.625951
512	6.074515	4.314100	4.355313
1024	50.876876	30.463557	30.535113
2048	432.109365	214.498836	214.620047
4096	6314.935657	1535.468129	1535.459451

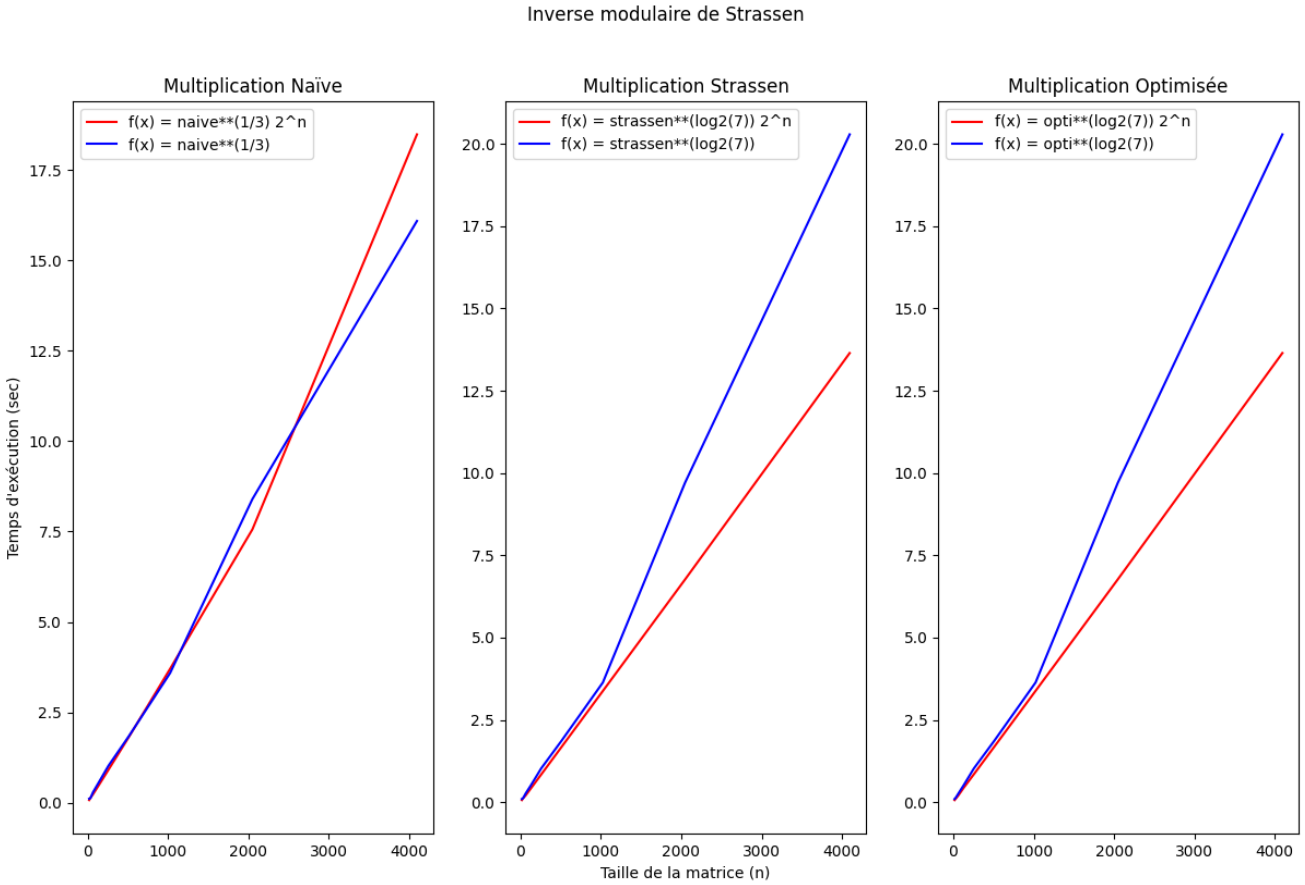
Complexité	$O(n^3)$	$O(n^{log2(7)})$	$O(n^{log2(7)})$

n	**Strassen multiplication naïve (sec)**	**Strassen multiplication Strassen (sec)**	**Strassen multiplication optimisée (sec)**
20	0.001371	0.001369	0.001371
30	0.001993	0.002737	0.003915
70	0.024410	0.024558	0.023889
130	0.151757	0.150088	0.150690
250	1.067649	1.065582	1.065784
500	6.300116	5.813254	5.801076
1000	46.171218	37.435353	37.441221
2050	592.286172	590.999967	589.709857
4100	4165.243657	4676.334282	4675.383973
Complexité	$O(n^3)$	$O(n^{log2(7)})$	$O(n^{log2(7)})$



On constate que la multiplication de Strassen est bien plus efficace que la version naïve pour une taille de matrice supérieure à 2048.


Inverse modulaire de Strassen
Multiplication Naïve
Multiplication Strassen
Multiplication Optimisée
Naive 2^n
Naive
Strassen 2^n
Strassen
Optimisée 2^n
Optimisée
Temps d'exécution (sec)
Taille de la matrice (n)


Inverse modulaire de Strassen
Multiplication Naïve
Multiplication Strassen
Multiplication Optimisée
f(x) = naive**(1/3) 2^n
f(x) = naive**(1/3)
f(x) = strassen**(log2(7)) 2^n
f(x) = strassen**(log2(7))
f(x) = opti**(log2(7)) 2^n
f(x) = opti**(log2(7))
Temps d'exécution (sec)
Taille de la matrice (n)

On remarque que pour des matrices de taille 2^n, il y a un réel gain de temps à utiliser la multiplication de Strassen au lieu de la multiplication naïve dans l'inversion de Strassen.
Cependant pour une matrice paire qui n'est pas multiple de 2^n, on peut voir qu'il n'y a pas de gain, on peut également apercevoir que la multiplication naïve est plus performante avec des matrices de grande taille.

On remarque également qu'il n'y a quasiment aucune différence entre la multiplication de Strassen avec la multiplication optimisée, car les gains lors de l'utilisation de la multiplication naïve dans la multiplication optimisée est très minime voire inexistante par rapport à l'utilisation de la multiplication de Strassen. De plus le nombre d'appels à la multiplication naïve est très faible cela montre l'inefficacité de choisir entre les deux algorithmes.

IV - Conclusion

On constate une différence notable de temps entre les algorithmes d'inversion de PLUQ et de Strassen implémentant la multiplication naïve pour des matrices de grande taille. Malgré une complexité identique, ils possèdent un coefficient directeur différent.
Pour l'inversion de Strassen, utiliser la multiplication de Strassen est beaucoup plus efficace que d'utiliser la multiplication naïve.

Nous nous sommes également assuré qu'il n'y ait aucune fuite mémoire lors de l'exécution du code.

Pour améliorer le code, on pourrait paralléliser le code avec OpenMP et MPI, ou encore améliorer les opérations de base en limitant l'utilisation du modulo qui est très coûteuse.