ALGORITHMIQUE ALGÉBRIQUE

POLYTECH SORBONNE

RENDU DE PROJET

BERBAGUI Amine
HACHANI Ghassen

MAIN 4

10 mai 2022

Table des matières

1 Introduction

1.1 But du projet

Ce projet consiste en l'implémentation en langage C de différents algorithmes fondamentaux de l'algèbre numérique et de mesurer les performances de ces derniers. Nous travaillerons qu'avec des matrices carrés dans le corps finis $\mathbb{Z}/p\mathbb{Z}$ avec p un nombre premiers d'au plus 30 bits. On supposera que les matrices dont on calcul la matrice inverse sont en effet inversibles.

En effet, on peut reformuler ce dernier point au travers de la notion de matrice générique. Dans ce contexte, une matrice générique est une matrice pour laquelle toutes les valeurs propres sont distinctes. Ainsi, si l'on choisit les coefficients d'une matrice au hasard, alors c'est ce que l'on est susceptible d'obtenir. On travaillera donc avec des matrices générés aléatoirement pour éviter les divisions par zéro. Vous trouverez plus de détail sur ce site.

En plus du code bien détaillé, ce rapport éclaircira chaque algorithme implémenté ainsi que la structure du code. Enfin, on commentera les mesures de performances obtenus sur ces différents algorithmes. Vous trouverez par ailleurs le lien vers le dépôt Git de ce projet ici :

1.2 Pré-requis

Nous avons travaillés avec quelques macros qui sans explications peuvent être difficiles à comprendre. Voici toutes les macros utilisés dans le code :

```
1 // some macro to go faster
2 #define foreach(a,b,c) for(int a = b; a<c ; a++)
3 #define forneg(a,b,c) for(int a = b; a>=c ; a--)
4 #define for_(a,n) foreach(a,0,n)
5 #define DIM int n   // the size of the matrix
6 #define prime int p // the prime number
7 typedef int **matrix; // the matrix
```

Avoir `usr/bin/python3` comme interpréteur python. Si ce n'est pas le cas, veuillez indiquer le bon chemin dans le script python ! Plus de détail à la prochaine page.
Comme nous travaillons avec des modulos, toutes les opérations arithmétique modulo p comme l'addition, la soustraction, et la division se trouve dans le fichier `Tools.c`.

1.3 Compilation et exécution

Un makefile est à votre disposition pour vous faciliter la vie. Ouvrer un terminal dans le dossier du projet puis taper `make` pour compiler tous les fichiers source. Vous pouvez lancer l'exécutable en indiquant en argument la taille de la matrice **n** ainsi que le nombre premier **p** que vous voulez. Pour une matrice de taille 4 et un nombre premier assez grand, la syntaxe à respecter est celle-ci : `./project --n 4 --p 293`.
ATTENTION ! n doit être une puissance de 2.

Tout un tas de chose vont s'afficher sur le terminal dont :

- ✔ **la décomposition LU**
- ✔ **la résolution de systèmes linéaires en utilisant la décomposition LU**
- ✔ **le calcul de la matrice inverse naïve**
- ✔ **le calcul de la matrice inverse avec l'algorithme de Strassen**
- ✔ **La multiplication de 2 matrices avec l'algorithme de Strassen**

Enfin vous avez la possibilité d'exécuter les benchmarks en tapant `benchmarks n` avec **n** la puissance de 2 jusqu'où vous voulez allez. Par précaution ne mettez pas une valeur de n $>$ 12 sauf si vous avez assez de ressources. Pour avoir des résultats pertinents, allez jusqu'à n $=$ 11 (2048) ce qui peut prendre un peu de temps selon votre machine.
Exemple : `./project benchmarks 10`

Une fois exécuté, un menu s'affichera qui vous proposera de choisir quels algorithmes comparer durant le benchmark parmi l'inversion naïve, l'inversion Strassen, la multiplication naïve, la multiplication Strassen et l'inversion de Strassen avec la multiplication de Strassen. Vous verrez ensuite en temps réel le temps d'exécution pour chaque taille de matrices. Enfin, ces données sont exportés dans le dossier **benchmarks** sous forme de fichiers texte permettant de générer un **plot** à la fin du benchmark automatiquement.Le dossiers benchmarks, initialement absent se crée à la compilation et se supprime lors du `make clean`.

En cas d'erreur, veuillez vérifier que le chemin de l'interpréteur python est le bon ou bien vous pouvez toujours taper la commande `make plot` directement. Ce plot est ensuite sauvegarder dans le dossier **plot**. A titre d'exemple, un plot figure déjà dans ce répertoire (finalall.png), correspondant au benchmark de tous les algorithmes (option 4) jusqu'à $n = 2048$.
Pour faire simple , compiler sous Linux pour éviter les problèmes de chemins et d'affichage d'images !

N.B : Ce projet contient une documentation **Doxygen** que vous je vous invite à visualiser en ouvrant le fichier `index.html` (situé dans le dossier **doc**) dans un navigateur !

2 Décomposition LU

2.1 Explications

Décomposition d'une matrice carré **A** en deux matrices **L** et **U** avec L une matrice triangulaire inférieure et U une matrice triangulaire supérieure.Cette décomposition est utilisée en analyse numérique pour résoudre des systèmes linéaires que nous traiterons par la suite.
Après avoir fais mes recherches, au lieu de chercher L et U tel que A = LU, j'ai préféré chercher L,U et P tel que PA = LU. En effet, la matrice P représentant la matrice de permutation, est utile si l'on veut éviter un pivot à zéro dans notre décomposition. Cette forme plus général est plus "sure" car elle effectue des permutations lorsque c'est nécessaire.

2.2 Description du code

A l'exécution, on rentre dans le `else` du main où l'on récupère les arguments rentrés,on initialise nos matrices puis on exécute `LaunchProject` qui va exécuter `RunLU`.

```
//LU decomposition
RunLU(A,P,L,U,n,p);
```

`RunLU` va exécuter la fonction `pivot` qui calcul la matrice de permutation. On calcul ensuite la matrice **PA** que l'on passe en paramètre dans la fonction `LU` de sorte à ce que cette dernière calcule bien PA = LU et non A = LU.

```
// calcul la matrice de permutation P
pivot(A,P,n);
// compute P*A
matrix PA = matmul(P,A,n,p);
```

Si $j \leq i$ donc la partie supérieure, on calcul U à l'aide des fonctions arithmétiques modulo p (sub et inv)

```
// Compute the U matrix
s = 0;
foreach(k, 0, j)
    s+= (long) ((L[j][k] * U[k][i]) % p);
tmp = sub(A[j][i],s,p);
U[j][i] = sub(Abis[j][i],s,p);
if(U[j][i] < 0) U[j][i] += p;
if(U[j][i] > p) U[j][i] -= p;
```

Idem, si on a $j > i$, on calcul la matrice L

```
// Compute the L matrix
s = 0;
foreach(k, 0, i)
    s+= (long) ((L[j][k] * U[k][i]) % p);
tmp = sub(Abis[j][i],s,p);
L[j][i] = (long) (tmp * inv(U[i][i],p)) % p;// modular inverse
if(L[j][i] < 0) L[j][i] += p;
if(L[j][i] > p) L[j][i] -= p;
```

2.3 Résultat

On s'assure bien que le produit de L par U est égal au produit de P par A puis on affiche joliement les matrices comme ceci avec le temps d'exécution :

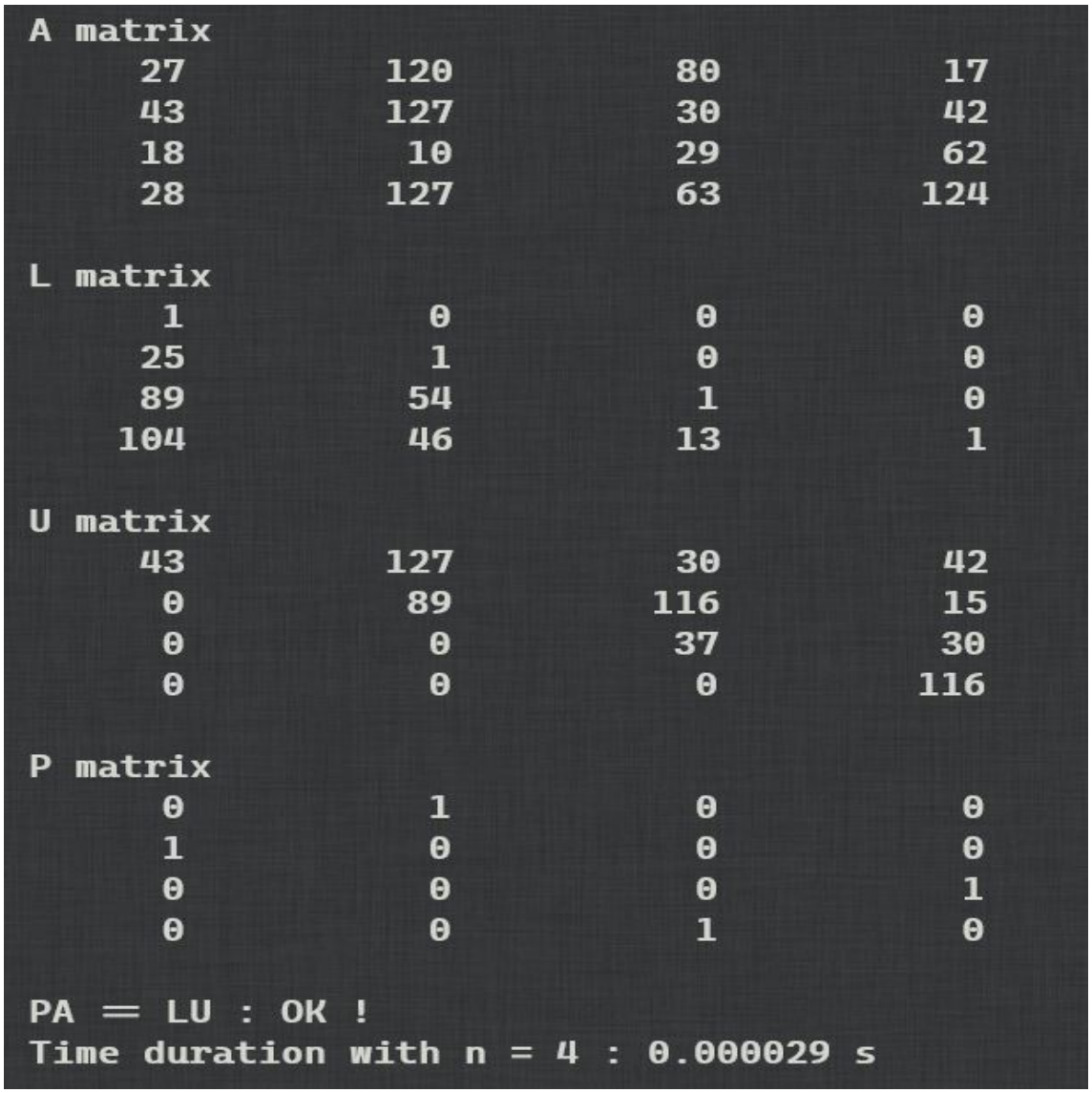


FIGURE 1 – Affichage décompostion LU

3 Résolution de systèmes linéaires

3.1 Explications

Passons maintenant à la résolution de systèmes linéaires en utilisant la décomposition LU. L'idée est la suivante : On cherche à trouver x tel que $Ax = B$. Or $A = LU$ donc on a $LUx = B$.

En posant $Z = Ux$, on se retrouve donc avec un système de deux équations à 2 inconnues (Z et X) :

$$\begin{cases} LZ = B \\ UX = Z \end{cases}$$

3.2 Description du code

Le lancement depuis le main se fait de la même manière que pour LU à la différence qu'on initialise les vecteurs B,Z et X.

```
// for the linear system solving
int B[n]; // Solutions vector
int Z[n]; // LZ = B
int X[n]; // UX = Z
```

De même dans la fonction `RunLS`, on génére une solution aléatoire mod p, on initialise Z et X et on appelle `LinearSystem`.

```
// creation of a random solution vector modulo p
generateSolution(B,n,p);
// Initialization of Z and X with 0
InitVector(Z,X,n);
```

Forward substitution pour trouver Z

```
foreach(i,0,n) // Finding Z forward substitution
{
    sum = 0;
    foreach(j,0,i)
        sum += (long)(L[i][j]*Z[j]) % p;
    tmp = sub(B[i],sum,p);
    Z[i]= (long) ( tmp * inv(L[i][i],p)) % p;
    if(Z[i] < 0) Z[i] += p;
}
```

Backward substitution pour trouver X

```
forneg(i,n-1,0) // Finding X  backward substitution
{
    sum = 0;
    forneg(k,n-1,i)
        sum += (long)((U[i][k]*X[k]) % p);
    tmp = sub(Z[i],sum,p);
    X[i]= (long) ( tmp * inv(U[i][i],p)) % p;
}
```

3.3 Résultat

Pareil que pour la décomposition LU, on s'assure dans la fonction `CorrectionLS` que l'on a bien l'égalité $PA = LU$.

```
*************** Linear system solving ***************
1 :Solution vector
B0 = 73
B1 = 30
B2 = 43
B3 = 93

2 : Solve LZ = B for Z

Z1 = 73
Z2 = 21
Z3 = 124
Z4 = 6

3 : Solve UX = Z for X

X1 = 76
X2 = 37
X3 = 116
X4 = 91

Checking :

LZ
 73  30  43  93
UX
 73  21 124   6

LZ == B : OK !
UX = Z : OK!
Time duration with n = 4 : 0.000003 s
```

FIGURE 2 – Affichage résolution système linéaire

4 Inversion de matrices : version naive

4.1 Explications

On s'intéresse à présent au calcul de matrices inverse. Pour rappel, une matrice inversible est une matrice carrée A pour laquelle il existe une matrice B de même taille n telle que le produit AB ou BA vaut la matrice identité. Le but est donc de se servir de notre décomposition LU pour calculer la matrice inverse de A. Or on a :
$PA = LU \iff A = LUP^{-1} \iff A^{-1} = L^{-1}U^{-1}(P^{-1})^{-1} \iff A^{-1} = L^{-1}U^{-1}P$ sachant que `InverseMatrix` calcul seulement $A^{-1} = L^{-1}U^{-1}$. On multiplie donc la matrice `Inv` par P pour ainsi tomber sur la bonne matrice inverse.

4.2 Description du code

De la même manière que précédemment, on lance la fonction `LaunchProject` du main qui elle-même lance la fonction `RunNaiveInverse`.

```
// Naive Inverse
RunNaiveInverse(A,n,p);
```

Dans `RunNaiveInverse` on initialise la matrice inverse puis comme on exécute la fonction `InverseMatrix`.Cette fonction calcul la matrice inverse en utilisant deux autres fonctions qui sont `computeZ` et `ComputeInverse`. La 1ère calcule les valeurs du vecteur Z pour chaque ligne à l'aide de la matrice L et de la matrice identité tandis que la seconde trouve la matrice inverse à l'aide de U et de Z.

```
for(int i = 0; i < n; i++)
if(i != row)
    tmp += L[row][i] * Z[i][col]; // product row of L by column of Z
sum = tmp % p; // modulo p
int result = sub(I[row][col],sum,p); // substract sum to the identity matrix
result = (result * inv(L[row][row],p)) % p; // compute the modular inverse
```

Enfin, comme expliqué en explications, on oublie pas de multiplier `Inv` par la matrice de permutation pour retrouver le bon résultat dans `RunNaiveInverse`

```
// As the matrix L and U correspond to PA = LU, Inv needs to be multiplied by P
matrix res = matmul(Inv,P,n,p);
```

Ainsi dans `InverseMatrix`, on a plus qu'a appeler nos deux fonctions dans 2 boucles imbriqués chacune. `row` et `col` ici sont mis en paramètre des deux fonctions pour que chaque fonction traite qu'une seule ligne/colonne à la fois bien qu'on aurait pu tout faire directement.

```
// compute z
for(int col = 0; col < n; col++)
    for(int row = 0; row < n; row++)
        Z[row][col] = computeZ(L,I,Z,col, row,n,p);
// compute inverse
for(int col = 0; col < n; col++)
    for(int row = n - 1; row >= 0; row--)
        inverse[row][col] = computeInverse(inverse,U,Z,col,row,n,p);
```

4.3 Résultat

Une méthode assez simple pour vérifier que notre matrice inverse est correcte est de la multiplier par la matrice A de départ et voir si l'on tombe bien sur la matrice identité. C'est ce que fait la fonction `correctionInv`. Voici un exemple de ce que l'on obtient :

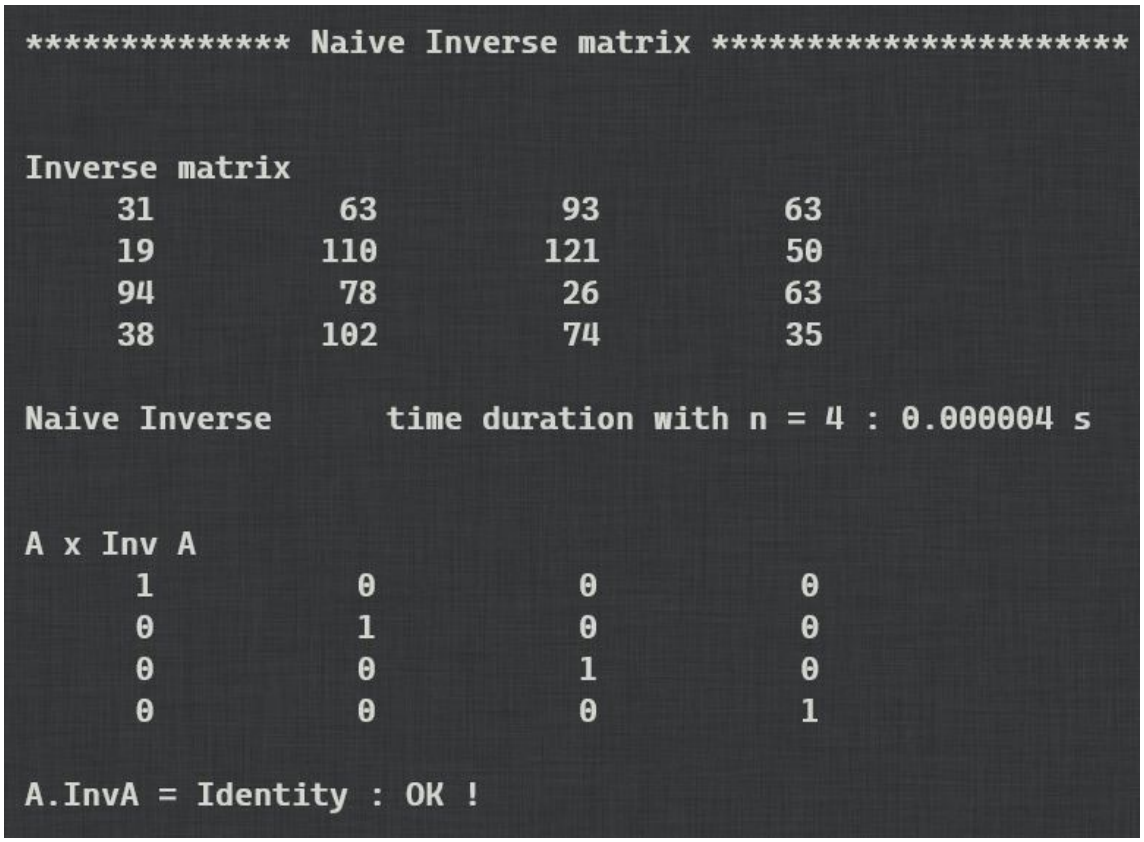


FIGURE 3 – Affichage inverse naïve

5 Inversion de matrices via l'algorithme de Strassen

5.1 Explications

On souhaite maintenant implémenter un autre algorithme permettant entre autres de calculer l'inverse d'une matrice. Il s'agit de l'algorithme d'inversion de Strassen. Ne trouvant pas cet algorithme bien détaillé sur les slides du cours, l'implémentation se base sur ce papier.
L'idée est la suivante. Si l'on a une matrice carré A partitionné comme suit

$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$, $A_{11} \in \mathbb{R}^{kxk}$, avec A_{11} et A régulier, alors on peut écrire $X = A^{-1}$ avec :

$$X = \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} = \begin{bmatrix} A_{11}^{-1} + A_{11}^{-1}A_{12}S^{-1}A_{21}A_{11}^{-1} & -A_{11}^{-1}A_{12}S^{-1} \\ -S^{-1}A_{21}A_{11}^{-1} & S^{-1} \end{bmatrix}$$

où $S = A_{22} - A_{21}A_{11}^{-1}A_{12} = (A/A_{11})$ est appelé le **complément de Schur**.
Voici les étapes de l'algorithme avec les 7 matrices R temporaires qui nous servirons à construire X_{11}, X_{12}, X_{21} et X_{22}

1. $R_1 = A_{11}^{-1}$
2. $R_2 = A_{21}R_1$
3. $R_3 = R_1A_{12}$
4. $R_4 = A_{21}R_3$
5. $R_5 = R_4 - A_{22}$
6. $R_6 = R_5^{-1}$
7. $X_{12} = R_3R_6$
8. $X_{21} = R_6R_2$
9. $R_7 = R_3X_{21}$
10. $X_{11} = R_1 - R_7$
11. $X_{22} = -R_6.$

5.2 Description du code

Tout se passe dans la fonction `InvStrassen`. Si n vaut 1, alors on retourne l'inverse modulaire de a

```
if(n == 1)
    matrix a = init_matrix(n);
    int value = A[0][0];
    a[0][0] =  inv(value,p);
    return a;
```

On pose $k = n/2$ puis on partitionne notre matrice en 4 sous-matrices avec la fonction `SubMatrix`. Elle prend en argument A, la taille n, ainsi que la position de la sous-matrice (1 en haut a gauche, 2 en haut à droite et ainsi de suite).

```
//split A into four sub-matrices
matrix A11 = SubMatrix(A,n,1);
matrix A12 = SubMatrix(A,n,2);
matrix A21 = SubMatrix(A,n,3);
matrix A22 = SubMatrix(A,n,4);
```

On applique ensuite toute les étape de l'algorithme puis on assemble nos 4 sous-matrices pour former la matrice inverse.

```
// Strassen steps
matrix R1  = InvStrassen( A11,k ,p);
matrix R2  = matmul( A21, R1, k, p);
matrix R3  = matmul( R1, A12 ,k ,p);
matrix R4  = matmul( A21, R3 ,k ,p);
matrix R5  = SubstractMatrix( R4, A22,k, p);
matrix R6  = InvStrassen( R5, k, p);
matrix X12 = matmul( R3, R6 , k, p);
matrix X21 = matmul( R6, R2 , k, p);
matrix R7  = matmul( R3, X21, k, p);
matrix X11 = SubstractMatrix( R1, R7, k, p);
matrix X22 = ChangeSign( R6,k ,p);

// Assemble the complete matrix
matrix res = assembly(X11,X12,X21,X22,n);
```

5.3 Résultat

On aurait pu faire une fonction qui vérifie que la matrice inverse est correct mais comme c'est déjà le cas pour l'inversion naïve, on compare la matrice inverse des deux algorithmes voir si elles sont identiques ce qui est bien le cas.

```
********** Strassen's Inversion Algorithm **********
A

    83      188      171      281
    41      199      168      185
   171      242      177      192
    26       30       67      216
Inverse matrix:

   133      112       65       87
   263      271      183       20
   114      258      231      223
   121      151      260      235
Strassen's inverse time duration with n = 4 : 0.000013 s
```

FIGURE 4 – Affichage inverse Strassen

6 Algorithme de multiplication de Strassen

6.1 Explications

Toujours avec Strassen, on cherche maintenant à implémenter un algorithme plus connu qui est l'algorithme de multiplication. Cet algorithme permet d'effectuer le produit de deux matrices de taille n avec une complexité en $O(n^{2.807})$ contre $O(n^3)$ pour la multiplication naïve, donc devrait avoir de meilleures performances. L'implémentation se base sur celle de ce site.
En effet lors de la multiplication naïve, si l'on pose A et B deux matrices carrés,leur produit C vaut :

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, C = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

Ainsi, cette méthode nécessite 8 multiplications de matrices pour calculer les $C_{i,j}$. La force de l'algorithme de Strassen réside dans un ensemble de sept nouvelles matrices M_i qui vont servir à exprimer les $C_{i,j}$ avec seulement 7 multiplications au lieu de 8. L'algorithme peut donc se diviser en 4 étapes assez similaires à l'inversion :

1. On divise la matrice A et B en 4 sous-matrices chacune de taille n/2
2. On calcul les 7 matrices récursivement
3. On calcul les sous-matrices en utilisant les matrices Mi
4. On assemble les 4 sous-matrices qui forme le produit de A et B.

6.2 Description du code

Son implémentation se fait au niveau de la fonction `StrassenMult`. La fonction étant récursive, si n vaut 1, on retourne le produit des deux seuls coefficients de A et B

```
if(n <= 1) // if n = 1, then return the product of the only element in A and B
        res[0][0] = (long) (A[0][0] * B[0][0]) % p;
```

Ensuite, dans le else, on applique les 4 étapes décrites plus haut en se servant des fonctions d'addition et de soustraction puis on retourne le résultat.Voici un exemple pour chaque étape de l'algorithme.

```
//Divide the matrix A and B into 4 sub-matrices
matrix a11 = SubMatrix(A, n, 1);
// Algorithm steps
matrix m1 = StrassenMult(AddMatrix(a11,a22,n/2,p),AddMatrix(b11,b22,n/2,p),n/2);
// build the sub-matrices Cij
matrix c11 = AddMatrix(SubstractMatrix(AddMatrix(m1,m4,n/2,p),m5,n/2,p),m7,n/2);
// Assemble the complete matrix
res = assembly(c11,c12,c21,c22,n);
```

6.3 Résultat

Comme méthode de vérification, j'ai trouvé ce site permettant d'effectuer le produit de 2 matrices. On calcul ensuite le modulo de chaque coefficient voir si cela correspond au résultat et c'est le cas.

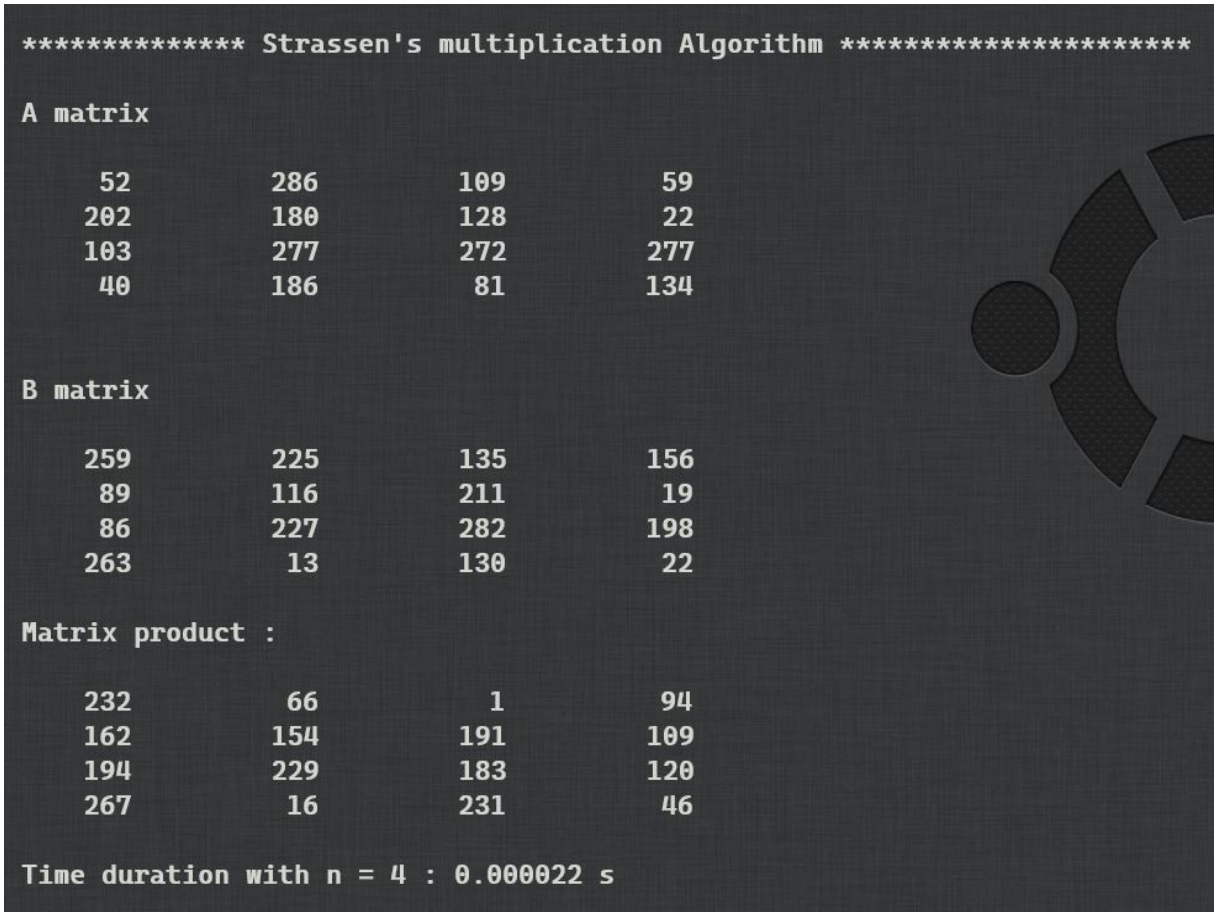


FIGURE 5 – Affichage multiplication Strassen

7 Mesure de performances

Pour finir, on souhaite maintenant étudier les performances de tout les algorithmes décrits dans ce rapport. Pour cela, comme expliqué en introduction, un dossier **Benchmarks** a été crée qui contiendra les tailles des matrices **n** ainsi que le temps d'exécution pour chaque **n** et cela pour chaque algorithme sous forme de fichiers texte. On se servira de ces données pour tracer sous forme de plot nos résultats qui graphiquement sont plus facile à comprendre. Les performances ont été faites sur un laptop avec un processeur Intel-Core i5-7300HQ à 3.5Ghz

7.1 Comparaisons théoriques

Commençons dans un premier temps par étudier les performances théoriques de l'algorithme de Strassen ainsi que de l'inversion naïve, c'est-à-dire leur complexité sur le papier. Lequel des deux est le plus performant ? En se basant sur le papier *matrix inversion is no harder than matrix multiplication*, on constate que l'on peut multiplier une matrice nxn avec un temps $mul(n) = \Omega(n^2)$ où $mul(n)$ satisfait $mul(n+k) = mul(n)\ \forall k \in [0, n]$ et $mul(n/2) \leq C.mul(n)$ avec C une constante < 1.

Ainsi, l'inverse naïve d'une matrice non-singulière se fait en un temps $O(mul(n))$. Si l'on pose A,B deux matrices carrés, le nombre d'opérations arithmétiques nécessaire à calculer le produit est $2n^3 - n^2 = O(n^3)$ ($n^3\otimes$ et $n^3 - n^2\oplus$). Donc pour n = 2, la multiplication naïve nécessite 8 produits de matrices or comme vu plus haut, l'algorithme de multiplication de Strassen arrive à réduire le nombre de produit à 7 d'où sa compléxité un peu meilleure en $O(n^{log_2 7}) \approx O(n^{2.807})$. Dès lors, l'inversion de Strassen combiné à la multiplication de Strassen a la même complexité. On peut donc faire un tableau récapitulatif comme celui-ci :

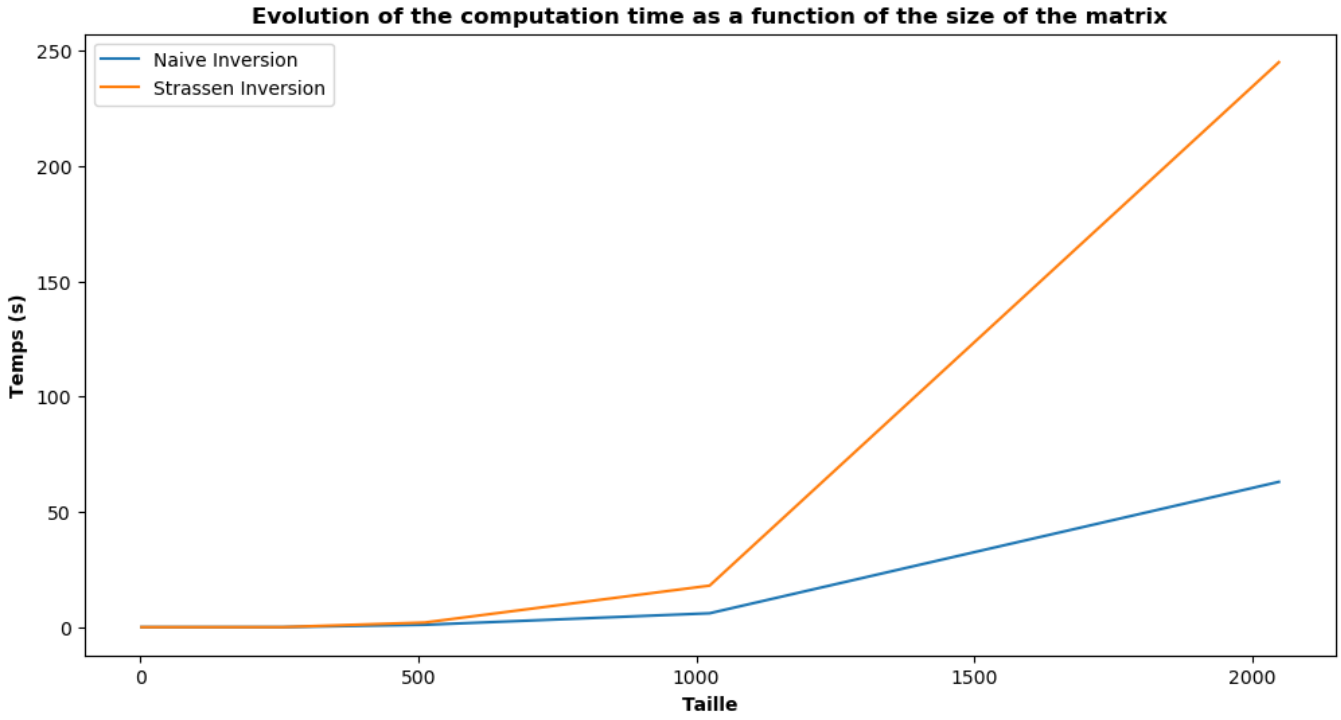
Opérations	**Algorithmes**	**Complexité**
Multiplications de matrices	Multiplication naïve	$O(n^3)$
	Multiplication de Strassen	$O(n^{2.807})$
Inversions de matrices	Élimination de Gauss-Jordan	$O(n^3)$
	Inversion de Strassen	$O(n^{2.807})$

Théoriquement, la multiplication de Strassen est plus efficace que la multiplication naïve et il en est de même pour l'inversion lorsque l'inversion de Strassen est combiné à la multiplication de Strassen. Regardons ce que cela donne dans la pratique !

7.2 Comparaisons pratiques

7.2.1 Inverse naïve VS Inverse Strassen

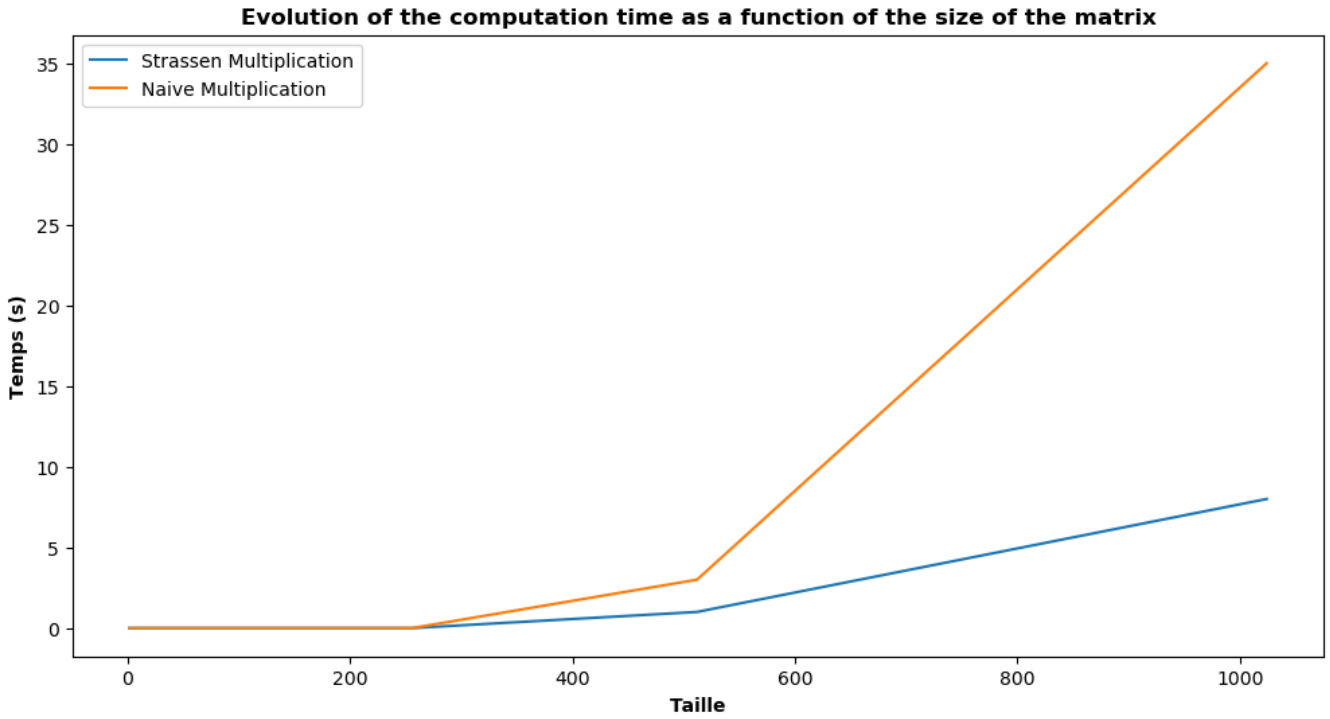
On cherche à comparer l'inversion naïve utilisant la décomposition LU contre l'inversion de Strassen. Pour cela j'ai fais tourné les benchmarks jusqu'à $n = 2^{11}$, taille suffisamment grande pour avoir des résultats pertinents mais pas trop grande pour faire planter mon pc.



Graphiquement, on voit que l'inversion naïve s'en sort beaucoup mieux que l'inversion de Strassen et plus la taille de la matrice augmente, plus l'écart se creuse entre les deux. J'ai trouvé cela étonnant au départ mais après avoir comparé avec des collègues et demander à notre professeur, cela est plutôt normal car cet algorithme de Strassen fonctionne avec la multiplication naïve.

Or on sait que l'inversion de Strassen sans un produit de matrice efficace n'est pas intéressant au niveau des performances. En rajoutant à cela les multiples allocations mémoire lié à la récursivité, cela justifie les courbes obtenues.
Voyons ce que cela donne avec le produit de matrice version Strassen.

7.2.2 Multiplication naïve VS multiplication Strassen



En comparant ici la multiplication naïve avec la multiplication de Strassen, on retrouve bien ce qui était attendu théoriquement à savoir Strassen meilleure que le produit naïve lorsque n devient grand. Au niveau du code, la multiplication de Strassen s'applique lorsque $n \geq 32$, car le produit naïve reste meilleure pour les petites matrices. Cette taille limite a été trouvé en essayant les puissances de 2 entre 16 et 128. Finalement, l'importance du produit de matrice en moins dans la multiplication de Strassen (7) se voit d'autant plus que n est grand.

```
if(n <= 32)// limit below wich we apply naive product, 32 = crossover point
{
    matrix res = matmul(A,B,n,p);
    return res;
}
... Strassen multiplication
```

Combinons maintenant la multiplication de Strassen avec l'inversion de Strassen (tout deux récursifs) et voyons ce que l'on obtient. On garde toujours le seuil pour la multiplication de Strassen.

7.2.3 Inversion Strassen avec la multiplication Strassen

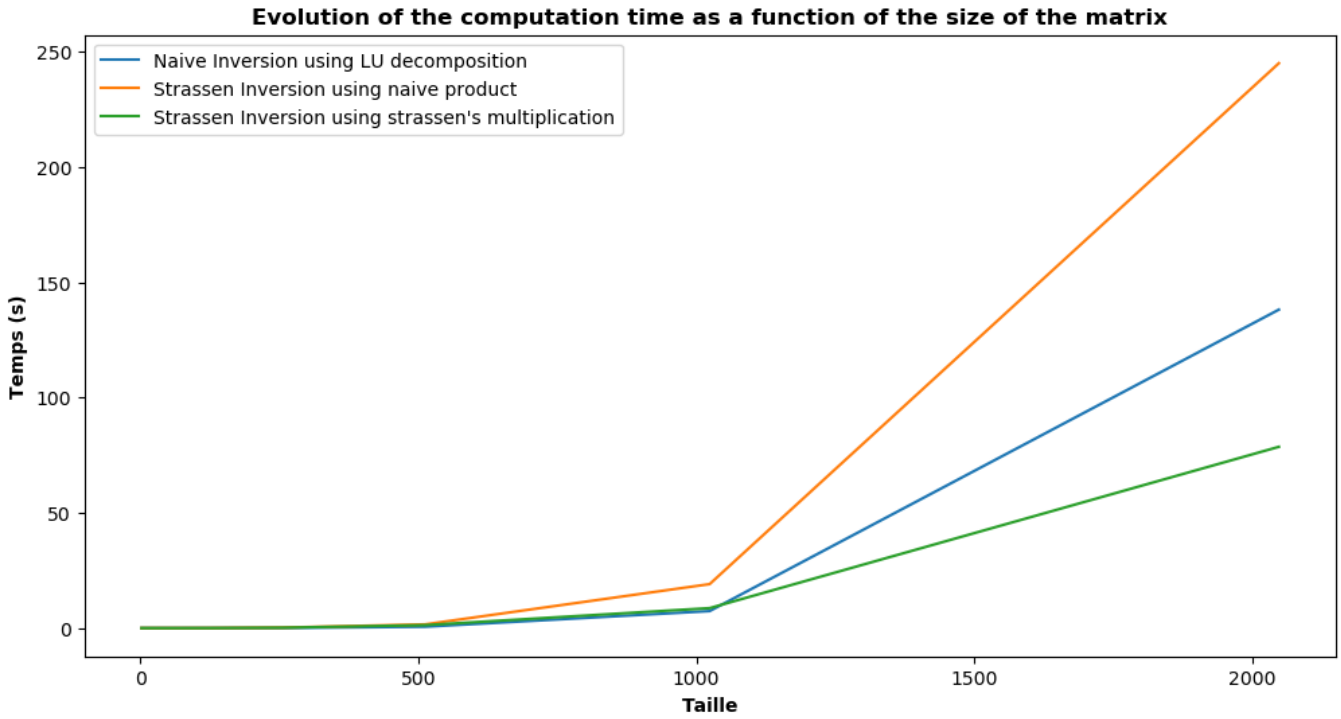


FIGURE 6 – Comparaisons des algorithmes

Sur ce plot sont superposés les résultats de **LU** et des deux variantes de l'inversion de Strassen. Comme on peut le voir, on retrouve ce qui était espéré à savoir l'inversion de Strassen avec la multiplication de Strassen qui devient beaucoup plus intéressante qu'avec le produit naïve. On remarque aussi que pour $n \leq 1024$ l'inversion naïve arrive à tenir tête à l'inversion de Strassen bien qu'après cette taille l'écart devient évident. On peut étudier le choix de la taille limite ϵ à partir de laquelle on effectue le produit naïve (*crossover point*) qui diffère selon les machines. En tâtonnant entre 16 et 128, je me suis aperçu que le crossover point ϵ optimal était $\epsilon = 32$.

Une amélioration possible de l'inversion de Strassen serait de modifier le cas de base de la récursion. Initialement, le cas de base se fait pour des matrice 1x1 avec l'inverse modulaire ce qui d'un point de vue des performances est assez pénalisant. Comme l'inversion naïve est efficace pour des matrices de petites tailles, on peut la mettre en cas de base en fixant une taille limite comme dans l'algorithme de multiplication.

```
if(n <= 32) // crossover point
    matrix res;
    res = InverseMatrix(I,L,U,P,Z,n,p); // Naive inverse
    return res;
```

Taille n	Inverse modulaire	Inversion naïve
2	7 μs	2 μs
4	6 μs	3 μs
8	18 μs	7 μs
16	108 μs	30 μs
32	497 μs	153 μs

TABLE 1 – Temps obtenus avec l'inversion de Strassen selon le cas de base choisis

On remarque que pour $n \leq 32$ (taille limite), on a un gain de temps non négligeable d'environ 30 %. Une autre amélioration purement informatique serait de modifier les options d'optimisations du compilateur au niveau du makefile. Comme vous l'aurez remarqué, la compilation se fait avec le flag `-O3`. Ce niveau d'optimisation permet des optimisations qui nécessitent une analyse et des ressources importantes au moment de la compilation, et modifie l'heuristique des optimisations par rapport à `-O2`. En revanche, `-O0` désactive toutes les optimisations. Comme on peut le

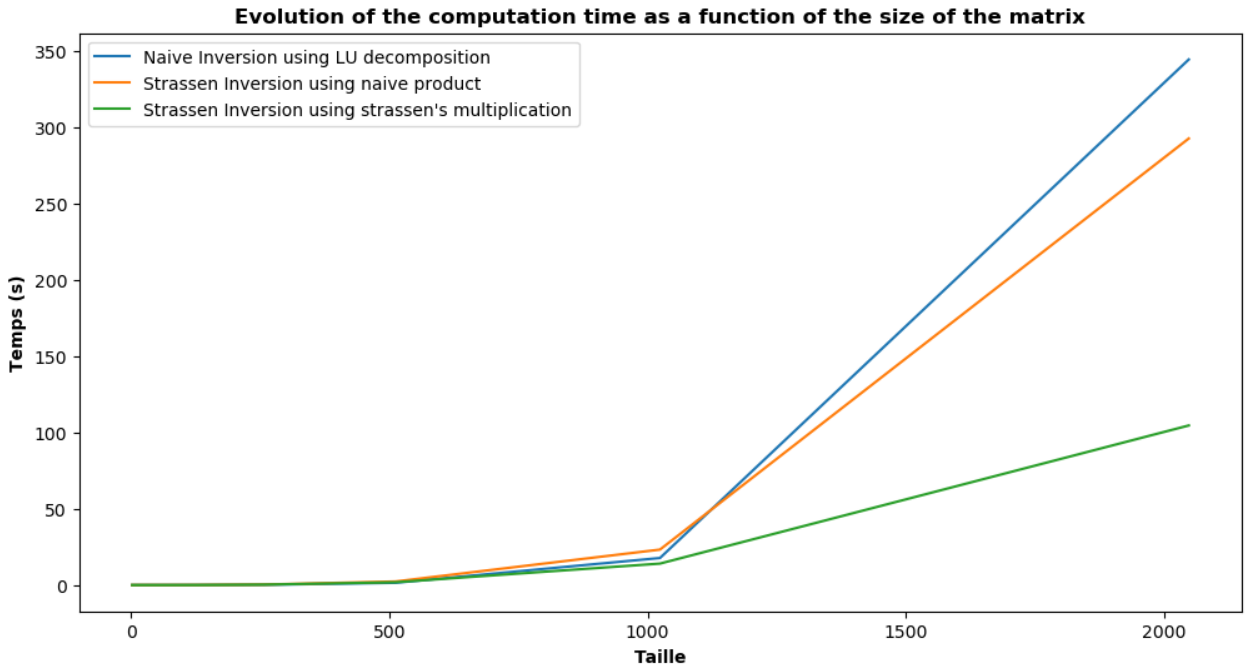


FIGURE 7 – Comparaisons des algorithmes sans optimisation du compilateur

voir sur ce plot, sans aucune optimisation, le temps d'exécution est nettement plus élevé. Pour n = 2048, les trois sont au dessus de 100s. De plus, si la courbe verte reste la plus rapide, on remarque que l'inversion de Strassen avec le produit naïve est devenu plus performant que l'inversion naïve. En effet, l'utilisation de l'option `-O0` entraîne une compilation et un temps de construction plus rapides, mais produit un code plus lent que les autres niveaux d'optimisation. La taille du code et l'utilisation de la pile sont nettement plus élevées comparé contrairement à `-O3` qui demande au compilateur d'optimiser les performances du code généré et de ne pas tenir compte de la taille du code généré. Comme Strassen est récursif et libère la mémoire à chaque fois, on comprend pourquoi pour $n = 2048$, Strassen avec le produit naïve devient meilleur que LU. Il peut y avoir d'autres raisons que j'ignore.

7.3 Comparaison finale et récapitulatif

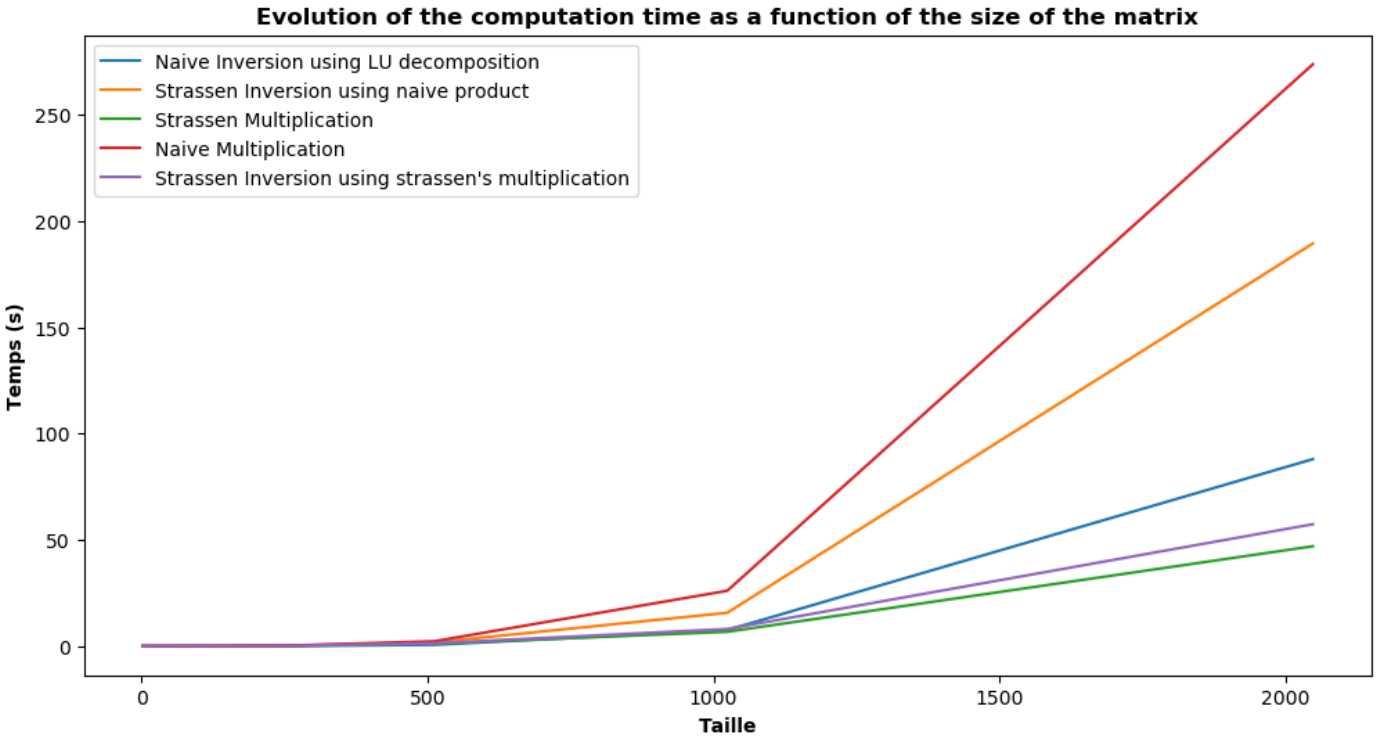


FIGURE 8 – Comparaisons des algorithmes jusqu'à n = 2048

Enfin, après ces analyses, on a regrouper tous les algorithmes vus précédemment sur un seul et même plot que vous pouvez voir ici. Du plus rapide au moins rapide, on retrouve :

- ★ La multiplication de Strassen
- ★ L'inversion de Strassen utilisant le produit de Strassen
- ★ L'inversion naïve utilisant la décomposition LU
- ★ L'inversion de Strassen utilisant le produit naïve
- ★ La multiplication naïve

Ces résultats sont plutôt logique au regard des performances de chacun. La multiplication naïve qui parcourt chaque ligne/colonne de la matrice et effectue $2048^3 \otimes$ est beaucoup plus lente que la multiplication de Strassen qui effectue moins de produits de matrice et est récursive. L'inversion de Strassen avec la multiplication de Strassen en $O(n^{2.807})$ meilleure que l'inversion naïve en $O(n^3)$, qui elle même est meilleure que l'inversion de Strassen avec le produit naïve en raison du produit naïve qui rend l'inversion de Strassen peu performante, du cas de base de la récursion à 1x1 pénalisant ainsi que les multiples allocations mémoires.

8 Conclusion

En conclusion de ce projet, nous avons étudié dans un premier temps les problèmes fondamentaux d'algèbres linéaires, comme la décomposition LU d'une matrice, la résolution du système linéaire $Ax = B$ ou encore retrouver l'inverse de A à partir de LU. Puis dans un second temps, nous avons pu nous intéressé à l'algorithme d'inversion de Strassen, d'abord une première version avec un produit de matrice naïf où on a pu mettre en évidence les résultats obtenus puis une seconde version amélioré avec l'algorithme de multiplication de Strassen permettant de rendre les performances optimaux d'où les graphes obtenus.

Nous avons pu étudier les performances avec les benchmarks des différents algorithmes pour constater une éventuelle convergence vers les attendus théoriques et le cas échéant expliquer la ou les raisons des résultats. On a vu que plusieurs paramètres pouvait influer sur nos résultats comme la valeur de la taille limite (*crossover point*) ou encore l'utilisation ou non d'options d'optimisations du compilateur.

Enfin, d'un point de vu personnel, ce projet fut une excellente occasion de concilier l'aspect théorique et pratique de certains algorithmes. En plus de renforcer nos compétences de programmation, leurs implémentations permet de développer une réflexion personnelle sur nos travaux et de susciter éventuellement notre curiosité sur les résultats pratiques.

Références

1. *Dense Linear Algebra from Gauss to Strassen*, Alin Bostan, September 30, 2021
https://specfun.inria.fr/bostan/mpri/DenseLinAlg.pdf

2. *Naïve matrix multiplication versus Strassen algorithm in multi-thread environment*, F. Belić , D Severdija, Z Hocenski, 2011
https://hrcak.srce.hr/file/106984

3. *Generalized matrix inversion is not harder than matrix multiplication*, Marko D. Petkovi´c, Predrag S. Stanimirovi´c
http://tesla.pmf.ni.ac.rs/people/DeXteR/old/Papers/RapidLU.pdf

4. *Triangular Factorization and Inversion by Fast Matrix Multiplication*, James R. Bunch and John E. Hopcroft, 1974
https://www.ams.org/journals/mcom/1974-28-125/S0025-5718-1974-0331751-8/S0025-5718-1974-0331751-8.pdf

5. *Computational complexity of mathematical operations*
https://en.wikipedia.org/wiki/Computational_complexity_of_mathematical_operations#Matrix_algebra

6. *Determining the Effects of Cross-over Point on the Running Time of Strassen Matrix Multiplication Algorithm*, January 2015
https://www.researchgate.net/publication/281548335_Determining_the_Effects_of_Cross-over_Point_on_the_Running_Time_of_Strassen_Matrix_Multiplication_Algorithm

7. *Arm Compiler User Guide*
https://developer.arm.com/documentation/100748/0612/using-common-compiler-options/selecting-optimization-options#:~:text=%2DO3%20instructs%20the%20compiler%20to,debug%20experience%20compared%20to%20%2DO2%20.