POLYTECH SORBONNE

SCIENCES SORBONNE UNIVERSITÉ

FLAG : Rapport de projet

$$AA^{-1} = I,$$

UE : Fondement de l'algorithmique algébrique

Étudiants :
Tahar AMAIRI
Hamza RAIS

Enseignant :
Vincent NEIGER

26 avril 2022

Table des matières

1 Introduction

Ce projet vise à implémenter et à comparer différents algorithmes d'inversion mais aussi de multiplication de matrices. On travaillera dans le corps fini $\mathbb{Z}/p\mathbb{Z}$ avec p un nombre **premier** ne dépassant pas les **30 bits**. Ainsi, toute opération arithmétique se fera **modulo p**.

L'implémentation se fera en `C` et pour des raisons de simplicité nous travaillerons uniquement avec des matrices **carrées** de taille 2^n. En effet, un des algorithmes implémentés ne fonctionne qu'avec des matrices carrées de taille 2^n (produit matriciel via l"algorithme de Strassen).

Concernant l'inversibilité, on partira du principe suivant : toute matrice générée aléatoirement dans le corps $\mathbb{Z}/p\mathbb{Z}$ a une très grande probabilité d'être inversible. L'implémentation ne vérifie donc pas à l'entrée si les matrices sont inversibles mais il est possible de vérifier si c'est le cas avec les fonctions implémentées (dont on discutera ultérieurement). Il est donc vivement recommandé de choisir un nombre premier **p assez élevé** afin d'obtenir (avec grande probabilité) des matrices inversibles.

Finalement, voici les algorithmes implémentés dont on va comparer les performances :

- Inversion via la décomposition LU
  - Décomposition LU
  - Résolution de système linéaire avec la décomposition LU
- Inversion via l'algorithme de Strassen
  - Produit matriciel naïf
  - Produit matriciel via l'algorithme de Strassen

2 Implémentation

Dans cette section nous allons couvrir l'implémentation des différents algorithmes en `C`. Pour plus d'informations concernant les paramètres des fonctions, il est conseillé de jeter un coup d'oeil aux commentaires au format **Doxygen** fournis avec le code dans les fichiers headers.

2.1 Pré-requis

La première étape est de représenter une matrice en C. Pour cela, nous allons utiliser une `struct` (voir le header `Matrix.h`) :

```
typedef struct Matrix
{
    int n; //the size
    int** values; //its values mod p
} Matrix;
```

Cette structure possède deux attributs : `n` qui représente la taille de la matrice et le tableau 2D `values` stockant les valeurs de la matrice **modulo p**. Avec cette structure on aura aussi besoin de plusieurs fonctions permettant sa manipulation :

— des fonctions permettant d'allouer de la mémoire afin de créer cette structure :

```
//create a new matrix struct with uninitialized value
Matrix* newMatrix(int n);
//create a new identity matrix struct
Matrix* newIdentity(int n);
//create a new matrix struct with random value mod p
Matrix* newMatrixModP(int n, int p);
```

— des fonctions pour effectuer des opérations d'addition et de soustraction mod p :

```
//sum two matrices (return A+B mod p)
Matrix* addMatrix(const Matrix* A, const Matrix* B, int p);
//sub two matrices (return A-B mod p)
Matrix* subMatrix(const Matrix* A, const Matrix* B, int p);
/* sub two matrices and save the result in one of them i.e :
invert = false : A = A - B mod p or invert = true : B = A - B mod p */
void subMatrixInPlace(Matrix* A, Matrix* B, int p, bool invert);
```

— une fonction test afin de déterminer si l'inverse obtenu est correct :

```
/* check if the matrix A_ is the inverse of A mod p, i.e :
A*A_ == I mod p*/
bool assertMatrixInversion(const Matrix* A, const Matrix* A_, int p);
```

— et finalement une fonction pour afficher la matrice et pour free la mémoire :

```
//free the allocated memory used by matrix
void freeMatrix(Matrix* matrix);
//print a matrix
void printMatrix(const Matrix* matrix);
```

Maintenant que nous avons notre structure `Matrix` et les fonctions nécessaires pour la manipuler, il faut transcrire toutes les opérations arithmétiques dans le corps $\mathbb{Z}/p\mathbb{Z}$ (voir le header `OperationModP.h`) :

— l'addition :

```
//compute x + y mod p
int add(int x, int y, int p);
```

— la soustraction :

```
//compute x - y mod p
int sub(int x, int y, int p);
```

— l'inverse modulaire en utilisant l'algorithme d'Euclide étendu [3] :

```
//compute 1/x mod p (exit(1) if x == 0 or the inverse does not exist)
int inv(int x, int p);
```

L'addition et la soustraction sont effectuées sans l'opérateur `%`, connu pour être très lourd en calcul. Concernant la multiplication elle s'effectue comme ceci :

```
((long) x * y) % p
```

Le cast en `long` permet d'éviter un overflow lorsqu'on utilise un nombre premier très grand.

2.2 Inversion via la décomposition LU

Afin d'inverser une matrice via la décomposition LU, il est tout d'abord nécessaire de pouvoir décomposer la matrice à inverser en un produit de deux matrices triangulaires l'une supérieure et l'autre inférieure (évident) mais aussi de pouvoir résoudre un système linéaire via cette décomposition. Par ailleurs, toutes les fonctions implémentant ces méthodes se trouvent dans le header `LU.h`.

2.2.1 Décomposition LU

Il existe de nombreuses versions pour la décomposition LU dont la version la plus connue utilisant le pivot de Gauss. Dans notre cas, nous avons implémenté la décomposition **Doolittle** [2] qui n'utilise pas le pivot de Gauss. Cette version est très simple à coder et permet d'obtenir une matrice $\mathbf{L}$ avec une diagonale fixée à 1 [1].

1. A l'inverse, pour obtenir une diagonale fixée à 1 concernant la matrice $\mathbf{U}$, on utilise la décomposition de **Crout**

Nous n'allons pas discuter de l'algorithme Doolittle car il existe une pléthore de liens l'expliquant mais nous nous sommes basé sur lien suivant [2]. La décomposition LU, via la méthode de Doolittle, est donc implémentée par la fonction suivante :

```
//compute the LU decomposition of A using Doolittle's method mod p
void LU(const Matrix* A, Matrix* L, Matrix* U, int p);
```

Cette fonction prend en entrée les matrices A, L et U : la matrice A est en `const` car elle n'est jamais modifiée par la fonction, cependant, le résultat est stocké dans les matrices L et U en paramètre. Ainsi, celles-ci doivent être déjà allouées en mémoire (à l'aide de la fonction `newMatrix()`) avant l'appel de la fonction. Bien plus, on rappelle que toute opération effectuée est modulo p d'où le nombre premier p en paramètre. Finalement, afin de vérifier le bon fonctionnement de l'implémentation, il existe une fonction test permettant de vérifier si le produit LU obtenu forme bien A :

```
//check if A == LU mod p
bool assertLU(const Matrix* A, const Matrix* L, const Matrix* U, int p);
```

2.2.2 Résolution de système linéaire

La décomposition LU a de nombreuses applications dont la résolution de système linéaire. En effet, en décomposant A en LU on peut prendre avantage de la forme triangulaire de ces matrices. De cette manière, le problème linéaire $Ax = b$ devient :

$$\begin{cases} Ly = b \\ Ux = y \end{cases} \tag{1}$$

Étant donné que ce système fait intervenir deux matrices triangulaires, on peut donc facilement le résoudre à l'aide de la **forward / backward substitution**. Ces deux méthodes sont implémentées par les fonctions suivantes :

```
//forward substitution
int* forwardSub(const Matrix* L, const int* b, int p);
//backward substitution
int* backwardSub(const Matrix* U, const int* b, int p);
```

Afin d'obtenir la solution du système linéaire, on utilise la fonction suivante permettant de résoudre (1) à l'aide des méthodes forward et backward :

```
//solve the linear system LUx = b mod p using forward/backward substitutions
int* linearSolveLU(const Matrix* L, const Matrix* U, const int* b, int p);
```

Cette fonction prend en paramètre les matrices L & U formant A mais aussi le vecteur [2] b du système linéaire pour renvoyer le vecteur solution x. Afin de vérifier si le vecteur x obtenu est bien solution, on utilise la fonction ci-dessous :

```
//check if Ax == b mod p
bool assertLinearSolveLU(const Matrix* A, const int* x, const int* b, int p);
```

2. tout comme pour la struct `Matrix`, il existe dans le header `Matrix.h`, des fonctions pour manipuler les vecteurs : pour allouer de la mémoire, pour afficher etc...

2.2.3 Inversion

Maintenant que nous pouvons obtenir la décomposition LU de A et résoudre un système linéaire avec, il devient très trivial d'inverser A, en effet :

$$LU = A \iff LU \cdot A^{-1} = A \cdot A^{-1} \iff LU \cdot A^{-1} = I$$

Ainsi, pour trouver A^{-1} il suffit de résoudre de multiples systèmes linéaires avec comme vecteur b les colonnes de la matrice d'identité. Pour chaque système linéaire résolu on obtiendra une ligne de la matrice inverse de A. L'implémentation de cette inversion se fait via la fonction `LUInversion()` :

```
//compute the inverse of A using the LU decomposition and by revolving linear system mod p
Matrix* LUInversion(const Matrix* A, const Matrix* L, const Matrix* U, int p, bool assert);
```

Cette fonction renvoie l'inverse de la matrice A et nécessite la décomposition LU en amont. Enfin, le paramètre `assert` permet d'effectuer une vérification à chaque résolution de système linéaire en utilisant la fonction `assertLinearSolveLU()`. Si l'une des vérifications échouent, la fonction s'arrête en retournant un pointeur `NULL` (bien plus, elle avertit l'utilisateur avec un `print`).

2.3 Inversion via l'algorithme de Strassen

L'algorithme d'inversion de Strassen nécessite l'implémentation du produit matriciel. Par conséquent, nous discuterons de deux implémentations différentes pour cette opération : le produit matriciel standard et celui de Strassen. Par ailleurs, toutes ces fonctions se trouvent dans le header `Strassen.h`.

2.3.1 Produit matriciel

Le premier produit matriciel implémenté est le standard via cette fonction :

```
//compute the naive product between two matrices mod p
Matrix* naiveMultiplication(const Matrix* A, const Matrix* B, int p);
```

Comme dit auparavant, cette fonction implémente le produit matriciel naïf de complexité $\mathcal{O}(n^3)$ et renvoie le résultat sous forme d'une nouvelle matrice (on rappelle tout de même que celui-ci se fait toujours modulo p).

La seconde implémentation concerne l'algorithme de Strassen reposant sur le principe de **"Divide and conquer"** : l'idée est de décomposer récursivement un problème en deux ou plusieurs sous-problèmes de même type, jusqu'à ce que ceux-ci deviennent suffisamment simples pour être résolus directement. Les solutions aux sous-problèmes sont ensuite combinées pour donner une solution au problème initial. Dans l'algorithme de Strassen cela s'apparente par la décomposition des matrices A et B en sous matrices de taille $\frac{n}{2}$ et d'effectuer les produits matriciels récursivement sur celles-ci. A la fin, on assemble les résultats obtenus dans une matrice C de taille n. Pour implémenter cet algorithme nous allons nous reposer sur le PDF suivant : [1] (page 41).

La phase de décomposition (étape 2 de l'algorithme du PDF) en sous matrices de taille $\frac{n}{2}$ est effectué par la fonction suivante :

```
//decompose the matrix M into 4 equally sized matrices : A,B,C and D
void decomposeMatrix(const Matrix* M, Matrix* A, Matrix* B, Matrix* C, Matrix* D);
```

Cette fonction prend en paramètre la matrice M à décomposer et les sous matrices associées : A,B,C et D. Il est donc nécessaire d'allouer la mémoire à celles-ci en amont car elle va distribuer les valeurs de M dans ces sous-matrices. La seconde fonction dont aura besoin pour l'implémentation est celle permettant de regrouper les sous-matrices en une seule (étape 5 l'algorithme du PDF) :

```
//merge the matrices A,B,C and D into one matrix M
void mergeMatrix(Matrix* M, const Matrix* A, const Matrix* B, const Matrix* C,
const Matrix* D);
```

Enfin, l'implémentation finale est effectuée par la fonction suivante :

```
//compute the product between two matrices using the Strassen algorithm
Matrix* StrassenMultiplication(const Matrix* A, const Matrix* B, int p);
```

2.3.2 Inversion

L'inversion de matrice via l'algorithme de Strassen repose, tout comme le produit matriciel, sur le principe de "Divide and conquer". Par conséquent, il est nécessaire d'implémenter l'étape 4 (du PDF page 48) d'assemblage avec la fonction suivante :

```
//fourth step of the Strassen inversion algorithm
void recoverMatrixInverse(Matrix* M, const Matrix* A, const Matrix* B, const Matrix* C,
const Matrix* D, const Matrix* E, int p);
```

L'étape 4 de l'inversion consiste à recomposer la matrice M en effectuant des opérations modulo p sur les sous matrices suivantes :

- matrice A (matrice E dans le PDF)
- matrice B (matrice EBTCE dans le PDF)
- matrice C (matrice EBT dans le PDF)
- matrice D (matrice TCE dans le PDF)
- matrice E (matrice T dans le PDF)

Enfin, l'entièreté de l'algorithme récursif d'inversion est implémenté dans la fonction :

```
//compute the inverse of M using the Strassen algorithm mod p
Matrix* StrassenInversion(const Matrix* M, int p, bool ifStrassenProduct);
```

Le booléen `ifStrassenProduct` permet de commuter entre l'utilisation du produit matriciel standard ou bien celui de Strassen. Par ailleurs, notre implémentation utilise un seuil (`productThreshold`) où lorsque la taille d'une matrice est inférieure à celui-ci, la fonction `StrassenInversion()` utilise le produit standard. En effet, pour des petites matrices la multiplication de Strassen n'est pas du tout efficace et il vaut mieux privilégier le produit naïf. Dans notre implémentation nous avons fixé ce seuil à 64 (cette valeur se base sur celle suggérée par le PDF [1] page 42).

3 Fonctionnement

3.1 Compilation

Le code est fourni avec un **makefile** facilitant la compilation. Celui-ci utilise le compilateur `GCC` et nécessite la version **C99** du langage C. Il existe aussi une règle `clean` avec le makefile. Finalement, l'exécutable généré sera nommé `project.out`.

3.2 Options

L'exécutable accepte différentes options :

- **--prime** p : permet de choisir le nombre premier p du corps $\mathbb{Z}/p\mathbb{Z}$ (par défaut $p = 65537$)
- **--size** n : permet de choisir la taille des matrices. Attention, la taille doit être une puissance de deux (par défaut $n = 4$)
- **--demo** d : si $d \neq 0$, cette option permet d'exécuter une démo utilisant toutes les méthodes implémentées (par défaut $d = 1$)
- **--test** t : si $t \neq 0$, cette option permet d'effectuer une mesure du temps d'exécution de chacune des méthodes et d'exporter les résultats au format CSV dans le dossier `benchmark` (par défaut $t = 0$)
- **--repeat** r : permet de fixer le nombre de répétition effectuée pour chaque taille n lors des tests afin d'obtenir une moyenne (par défaut $r = 3$)
- **--limit** l : permet de fixer la taille maximale des matrices lors des tests tel que $n = \left[2, 4, ..., 2^l\right]$ (par défaut $l = 10$)

Il est important de noter que le programme vérifie la cohérence des paramètres fournis en option (par exemple un nombre p non premier, une taille n négative etc...). Dans ce cas, il enverra un message d'erreur et terminera son exécution. Concernant l'exportation des résultats des tests en graphique, un script `plot.py` est mis à la disposition de l'utilisateur. Néanmoins, celui-ci requiert l'installation du module **matplotlib**.

Finalement, voici un exemple de commande d'exécution :

```
./project.out --size 4 --demo 0 --test 1 --limit 5 --repeat 10
```

Ici, la configuration exécuté par le programme est la suivante :

- $p = 65537$: car on ne l'a pas spécifié et c'est donc la valeur par défaut qui est prise
- $n = 4$: la taille des matrices
- $d = 0$: pas d'exécution de la démo
- $t = 1$: exécution des benchmarks
- $l = 5$: lors des tests la taille des matrices va varier de 2 à $2^l = 2^5 = 32$
- $r = 10$: chaque test sera répété 10 fois

4 Performance

Dans cette section nous allons présenter et discuter des performances de chacune des méthodes implémentées. Tous les tests ont été effectués sous **Windows 10 avec WSL2 (Ubuntu)** en utilisant un **Ryzen 5 1600 clocké à 3,8 Ghz**. Concernant les options d'exécution :

— $p = 65537$
— $l = 11$
— $r = 10$

4.1 Produit

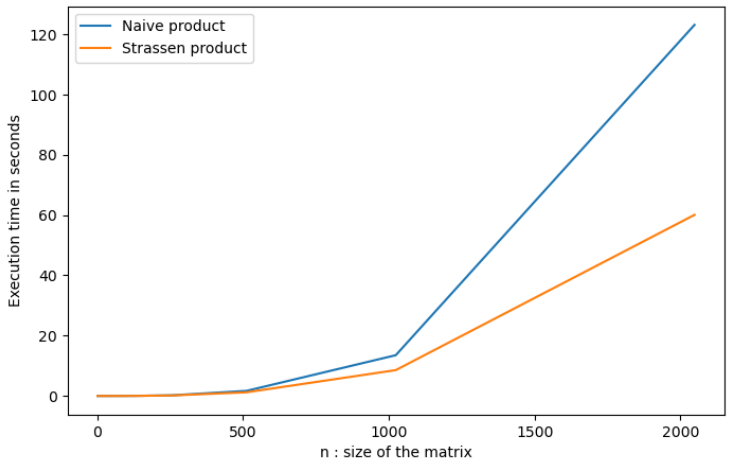

FIGURE 1 – Temps d'exécution pour le produit entre deux matrices

Sans surprise, le produit de Strassen est beaucoup plus rapide que le standard pour les grosses matrices (à partir de $n > 500$). A l'inverse, pour les petites matrices leur temps d'exécution est très proche, ce qui est normal car le produit de Strassen fait appel au produit naïf dans ce cas de figure.

La complexité asymptotique du produit standard et de celui de Strassen est respectivement de $\mathcal{O}(n^3)$ et $\mathcal{O}(n^{2.8})$. Comme nous travaillons avec des matrices d'une taille 2^n, passer d'une abscisse à une autre devrait théoriquement multiplier par un facteur de $2^3 = 8$ le temps d'exécution pour le produit naïf et par $2^{2.8} \approx 7$ pour le produit de Strassen. Et c'est ce que nous remarquons :

Rapport	**Naïf**	**Strassen**
$t_{n=2048}/t_{n=1024}$	9.11	7.01
$t_{n=1024}/t_{n=512}$	8.02	7.10
$t_{n=512}/t_{n=256}$	8.10	7.11
$t_{n=516}/t_{n=128}$	8.09	7.12

4.2 Inversion

Lors des tests sur l'inversion, nous avons remarqué que l'utilisation des flags d'optimisation du compilateur, i.e `-O3` et `-march=native`, influe énormément sur les résultats de l'inversion [3]. Par conséquent, nous présenterons les temps d'exécutions dans les deux cas de figure.

4.2.1 Sans flags d'optimisation

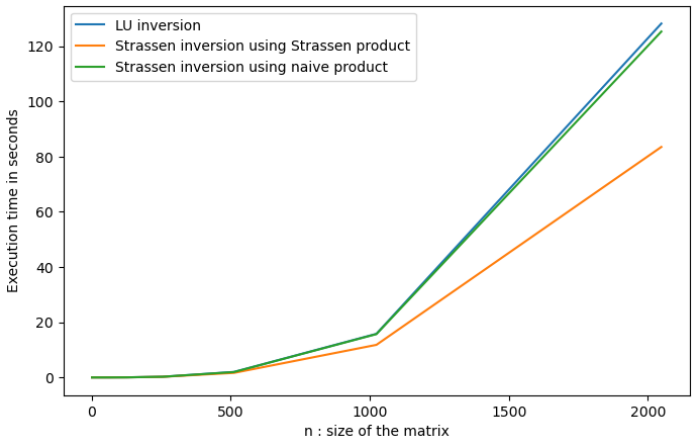

FIGURE 2 – Temps d'exécution pour inverser une matrice

Sans l'utilisation de flags d'optimisation lors de la compilation, on remarque que l'inversion de Strassen utilisant le produit matriciel naïf possède un temps d'exécution similaire à l'inversion par décomposition LU. L'implémentation complète de l'inversion de Strassen, i.e avec le produit de Strassen, fournit les meilleurs temps d'exécution.

Comme pour le produit matriciel, la différence entre les méthodes s'accentue avec les grandes matrices (surtout à partir $n > 1024$, où cette différence explose). Au vu de la forme des trois courbes, on peut **estimer** que les trois méthodes ont au moins une complexité asymptotique polynomiale $\mathcal{O}(n^k)$ avec un exposant $k \geq 2$.

En partant de ces deux observations, on peut facilement deviner que ces méthodes ont un exposant très proches. En effet, nous observons le même phénomène que pour le produit matriciel où nous avons une différence de 0.2 pour l'exposant : celle-ci mène à un temps d'exécution entre les méthodes très similaire pour les petites matrices et une énorme disparité pour les grandes matrices. Ce qui est tout à fait logique, car multiplier une petite valeur par un facteur 8 ne résulte pas de la même manière qu'avec une grande valeur.

Finalement, pour cette partie, il se peut qu'il y est quelques inconsistances concernant les résultats : parfois, l'inversion LU est plus rapide que l'inversion de Strassen utilisant le produit naïf. Cependant, la différence reste toujours minime comme sur le graphique d'au dessus.

3. Ceci n'est pas vrai pour la multiplication de matrices, les flags d'optimisation conservent les résultats qu'on a obtenu

4.2.2 Avec flags d'optimisation

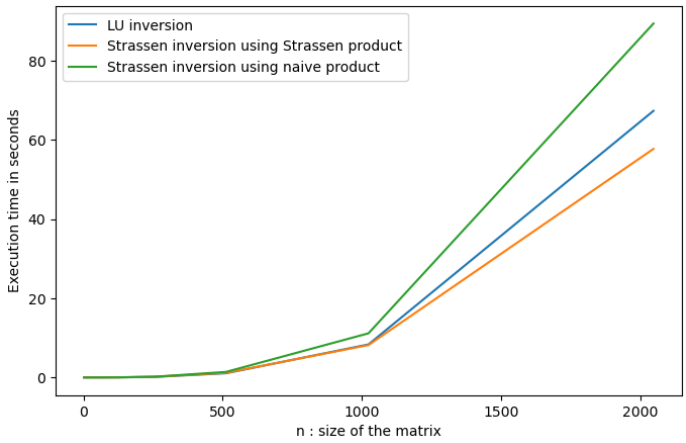

FIGURE 3 – Temps d'exécution pour inverser une matrice (flags)

En activant les flags d'optimisation du compilateur, on a une toute autre tendance : l'inversion par décomposition LU devient plus rapide que l'inversion naïve de Strassen. Cependant, elle reste toujours plus lente que l'inversion optimale de Strassen. Concernant l'allure des courbes, on retrouve les mêmes observations : croissance polynomiale, différence minime entre les méthodes pour les petites matrices etc...

En général, toutes les méthodes ont bénéficié des optimisations du compilateur mais la décomposition LU est celle qui a obtenu la meilleure amélioration :

LU	**Strassen naïf**	**Strassen optimal**
-47.5%	-35.0 %	-34.7 %

TABLE 1 – Réduction moyenne du temps d'exécution (inversion)

Produit naïf	**Produit de Strassen**
-27.3 %	-26.8 %

TABLE 2 – Réduction moyenne du temps d'exécution (produit)

Ce qui est tout à fait normal car l'algorithme de Strassen est récursif et nécessite des allocations mémoires récurrentes. Par conséquent, cela ne lui permet pas de prendre avantage de certaines optimisations comme par exemple l'utilisation du cache.

5 Conclusion

En conclusion, nous n'avons pas rencontré d'importantes difficultés lors de l'implémentation de ces différents algorithmes. Nous obtenons des résultats cohérents avec la théorie, i.e, de meilleures performances avec les algorithmes de Strassen. Cependant, il est intéressant de remarquer qu'au delà de la complexité asymptotique, il faut aussi prendre en considération la manière dont l'algorithme est implémenté. En effet, comme nous avons pu le voir avec les flags d'optimisation, notre implémentation de la décomposition LU bénéficie beaucoup plus des optimisations du compilateur que l'inversion de Strassen qui est une fonction récursive nécessitant des allocations mémoires à chaque appel récursif. Cette observation permet de souligner à quel point les spécifications de l'hardware (niveau de cache, vitesse de la RAM, vitesse d'horloge du processeur, sets d'instructions etc...) influent sur les performances et qu'il faut donc prendre en considération ce point lorsqu'on programme.

Nous effectuons énormément d'allocation mémoire pour stocker des résultats intermédiaires dans nos fonctions de Strassen et on pense qu'il est possible d'améliorer ceci en utilisant des matrices temporaires de stockage. Une implémentation intéressante est discuté dans ce papier de recherche. Il implémente un second algorithme de multiplication, Winograd & Waksman, adapté pour les petites matrices. Finalement, un autre moyen d'amélioration est la parallélisation.

6 Bibliographie

[1] Alin Bostan. *Dense Linear Algebra from Gauss to Strassen.* URL : https://specfun.inria.fr/bostan/mpri/DenseLinAlg.pdf.

[2] *Doolittle Decomposition of a Matrix.* URL : https://www.engr.colostate.edu/~thompson/hPage/CourseMat/Tutorials/CompMethods/doolittle.pdf.

[3] *Extended Euclidean algorithm.* URL : https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm.

[4] *LU decomposition and more.* URL : https://en.wikipedia.org/wiki/LU_decomposition.