SORBONNE UNIVERSITÉ

IMPLEMENTATION PROJECT : MODEL

18.12.2024

CIDERE Taryck 21400635

CROUZET Oriane 21414555

CHEN Shiyao 28707756

Table des matières

1. Introduction

Ce projet a pour objectif d'implémenter différents algorithmes de traitement de matrices en langage C, incluant la multiplication matricielle, la décomposition LU ainsi que l'inversion, en utilisant différentes méthodes telles que la méthode naïve et l'algorithme de Strassen. Les performances de ces algorithmes seront évaluées et comparées sur des matrices de tailles variées, avec des coefficients à virgule flottante dans l'intervalle [-1, 1].

Dans les sections suivantes, nous présenterons les algorithmes que nous avons implémentés, ainsi que les tests effectués et la comparaison de leur complexité.

Ce projet est l'occasion de mieux comprendre les concepts abordés en classe ainsi que d'approfondir notre compréhension pratique de ces différents algorithmes.

2. Description d'implémentation

2.1. Multiplication Naïve

La fonction ***naive_matrix_multiplication*** calcule le produit matriciel C = A * B, où :

- A est une matrice de dimensions *n * m*.
- B est une matrice de dimensions *m * q*.
- C est le résultat de dimensions *n * q*.

La fonction suit une approche naïve de la multiplication de deux matrices, utilisant 3 boucles imbriquées pour effectuer les calculs.

Fonctionnement

1. La fonction vérifie que le nombre de colonnes de A correspond bien au nombre de colonnes de B.
2. On alloue ensuite une nouvelle matrice pour réceptionner le résultat de la multiplication.
3. On calcule chaque élément en parcourant les lignes de A et les colonnes de B.
4. On retourne la matrice C.

Complexité

On obtient une complexité de **O(n * m * q)**, où :

- *n* est le nombre de lignes de A.
- *m* est le nombre de colonnes de A (resp. le nombre de lignes de B).
- *q* est le nombre de colonnes de B.

2.2. Multiplication par Strassen

La deuxième fonction qui calcule le produit de deux matrices implémente l'algorithme de Strassen. Ainsi, ***strassen_product*** calcule aussi le produit matriciel C = A * B . Cet algorithme divise récursivement les matrices en sous-matrices plus petites (la taille est à chaque fois divisée par 2), ce qui permet d'obtenir une meilleure complexité. En effet, grâce à cette implémentation, on n'effectue que **7 multiplications** au lieu de 8, ce qui n'est pas négligeable quand on cherche à réduire la complexité d'un produit matriciel.

Fonctionnement

1. On ajuste les dimensions des matrices d'entrées si elles ne sont pas des puissances de 2. Pour les redimensionner, on ajoute des 0 jusqu'à obtenir une matrice de taille 2^k.
2. Lorsque les matrices ont une taille 1 * 1, le produit est directement calculé.
3. Les matrices A et B sont découpées en 4 sous-matrices égales : *A11*, *A12*, *A21*, *A22* (resp. pour B).
4. On calcule ensuite les 7 produits de Strassen :
    - *M1* = (*A11* + *A22*) * (*B11* + *B22*)
    - *M2* = (*A21* + *A22*) * *B11*
    - *M3* = *A11* * (*B12* - *B22*)
    - *M4* = *A22* * (*B21* - *B11*)
    - *M5* = (*A11* + *A12*) * *B22*
    - *M6* = (*A21* - *A11*) * (*B11* + *B12*)
    - *M7* = (*A12* - *A22*) * (*B21* + *B22*)
5. Les résultats des produits intermédiaires sont combinés pour reconstituer les sous-matrices de C (*C11*, *C12*, *C21*, *C22*), pour finalement former la matrice résultante.
6. Si la matrice a subi un redimensionnement au début, on effectue le processus inverse pour retrouver la taille initiale de la matrice.
7. Pour finir, on libère les matrices temporaires.

Complexité

L'algorithme de Strassen réduit le nombre d'opérations multiplicatives à $\mathbf{O(n^{2.81})}$, au lieu de $\mathbf{O(n^3)}$ pour l'algorithme naïf. Cependant, il augmente légèrement les opérations additives.

2.3. Décomposition LU

La fonction ***LU_factorization*** implémente une méthode de factorisation LU, ce qui nous permet de décomposer A en deux matrices triangulaires :

- L, une matrice triangulaire inférieure contenant les coefficients multiplicatifs utilisés pour éliminer les termes (sous la diagonale de 1).
- U, une matrice triangulaire supérieure, représentant la matrice transformée après les différentes éliminations.

Fonctionnement

1. Il faut tout d'abord vérifier que les matrices sont carrées et de même taille, sinon la fonction s'arrête en affichant un message d'erreur. Bien qu' aucune vérification ne soit effectuée à ce sujet, la matrice L passée en argument doit être une matrice identité pour que la fonction renvoie un résultat correct. On suppose également que la matrice U peut être sujette à la décomposition LU qui n'est pas toujours possible.
2. On procède aux différentes éliminations de Gauss pour chaque colonne : on parcourt les lignes pour sélectionner le pivot courant (U[i][i]) et on procède à l'élimination pour mettre à 0 tous les éléments de la colonne courante, en utilisant la fonction ***elimination***.

La fonction ***elimination*** utilise la fonction ***findEliminationMultiple***, qui calcule le coefficient multiplicateur utilisé pour éliminer un élément sous le pivot et effectue une soustraction sur toute la ligne de l'élément sujet à élimination afin de le mettre à 0.

3. Une fois les éliminations effectuées, on met à jour L et U. L conserve les coefficients utilisés pour chaque étape d'élimination, tandis que U devient une matrice triangulaire supérieure après toutes les éliminations.

Complexité

La complexité globale de la factorisation LU est de l'ordre de **O(n^3)**. En effet, la boucle extérieure de la fonction s'exécute **n - 1** fois. La boucle interne s'exécute **n - (i + 1)** à chaque itération de la boucle extérieure. Et enfin, chaque itération de la boucle interne de la fonction génère un appelle à ***elimination*** qui est en **O(n)**.

2.4. Inversion par LU

La fonction ***matrix_inversion*** permet d'inverser une matrice A carrée en utilisant la décomposition LU.

Fonctionnement

1. On crée deux matrices L et U, puis on copie A dans U pour effectuer les transformations.
2. On utilise la fonction ***LU_decomposition*** pour transformer A en facteur L et U.
3. Pour chaque colonne *j*, on résout
    - **L** * ***y*** = ***e***, où *e* est un vecteur unitaire.
    - **U** * ***x*** = ***y***, où *x* est la colonne j de l'inverse.
      Chaque colonne obtenue correspond à une colonne de la matrice inverse de A (invA).
4. Pour finir, on libère les matrices et les vecteurs temporaires.

Complexité

La complexité globale de l'inverse d'une matrice par la décomposition LU est en **O(n^3)**, dominée asymptotiquement par le calcul de la décomposition LU.

2.5. Inversion par Strassen

Deux versions de l'inversion par Strassen ont été implémentées :

- Une version qui utilise l'algorithme naïf de multiplication pour les multiplications internes.
- Une version qui utilise l'algorithme de Strassen pour optimiser les multiplications.

Les fonctions ***strassen_inversion_naive_product*** et ***strassen_inversion_strassen_product*** effectuent une inversion matricielle basée sur la décomposition récursive. L'idée principale est de découper A en 4 blocs différents *a*, *b*, *c* et *d*, de taille $\frac{n}{2} * \frac{n}{2}$. L'inverse A^{-1} est calculée à l'aide des formules suivantes :

- $e = a^{-1}$
- $Z = d - c*e*b$
- $t = Z^{-1}$
- $y = - e*b*t$
- $z = - t*c*e$
- $x = e + e*b*t*c*e$

Fonctionnement

1. Si la taille de A n'est pas une puissance de 2, on l'étend pour obtenir une matrice de taille 2^k à l'aide de la fonction ***pad_matrix***.
2. On crée une matrice inverse de A.
3. On divise A en 4 blocs différents *a*, *b*, *c* et *d*, de taille $\frac{n}{2} * \frac{n}{2}$.
4. On calcul *e*, *Z*, *t*, *y*, *z* et *x* comme décrit précédemment, *e* et *t* étant calculés récursivement.
5. Les matrices *x*, *y*, *z* et *t* sont combinées dans une seule matrice *res*.
6. On redimensionne la matrice à sa taille d'origine avec ***unpad_matrix***.
7. On libère les matrices temporaires pour éviter les fuites de mémoire avant de retourner *res*.

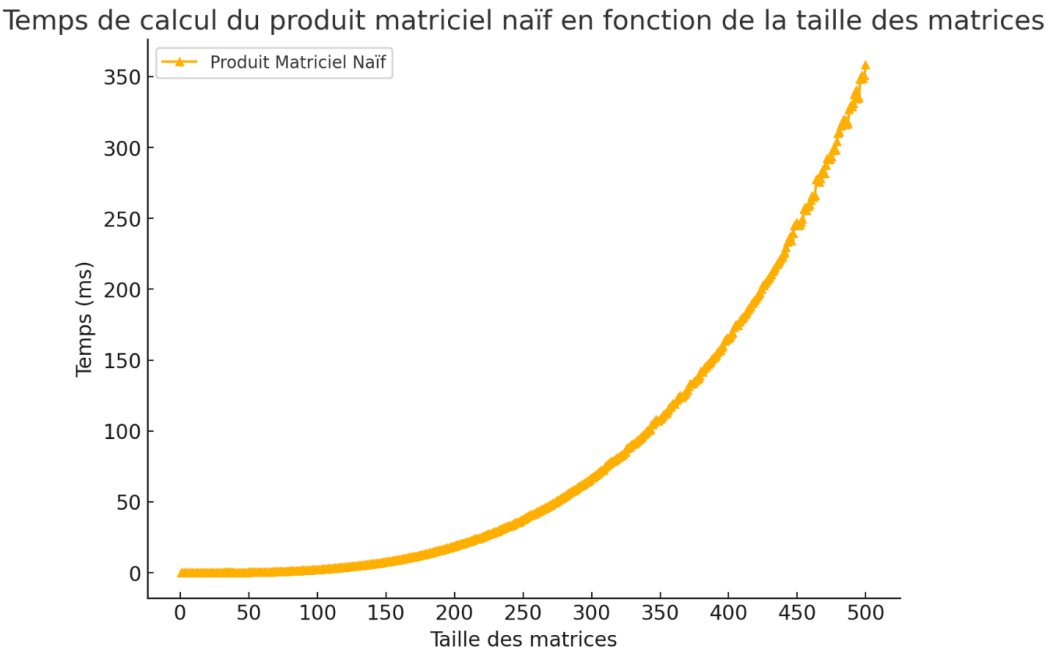
Complexité

La complexité de l'inversion par Strassen est :

- **$O(n^{2.81})$** avec la multiplication par Strassen.
- **$O(n^3)$** avec la multiplication naïve.

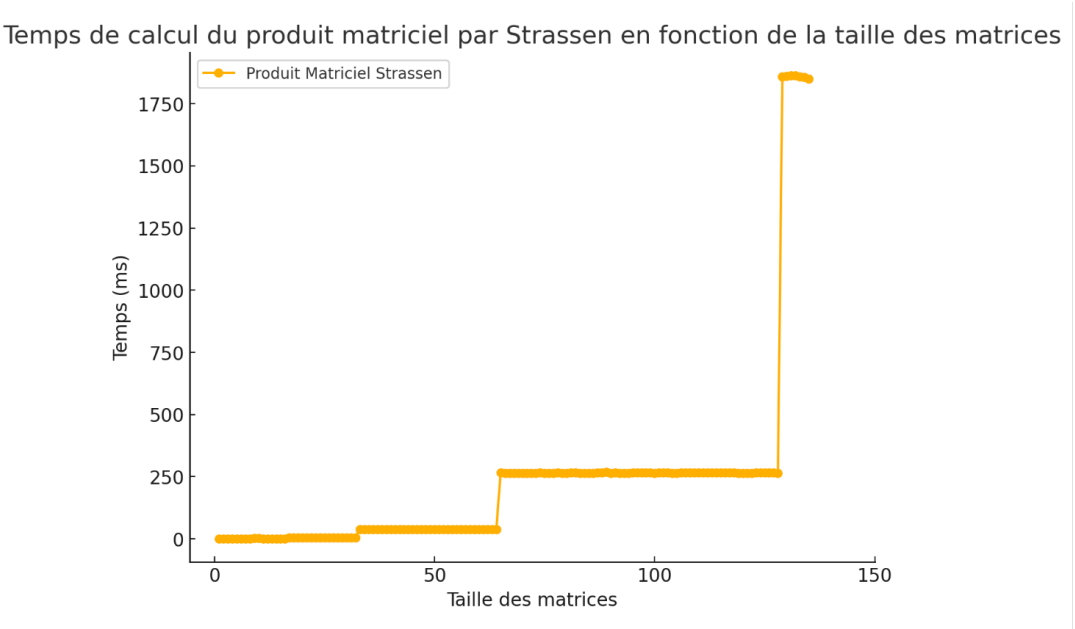
3. Résultats des Benchmarks

3.1. Multiplication Naïve



On observe que la courbe a une allure exponentielle, ce qui est cohérent avec une complexité cubique **O(n^3).** Les temps de calcul nous indiquent que l'algorithme naïf est efficace sur des matrices de petite taille, mais qu'il devient rapidement coûteux dès lors que les matrices ont des tailles plus importantes. Les temps de calcul sont ainsi en accord avec la complexité attendue.

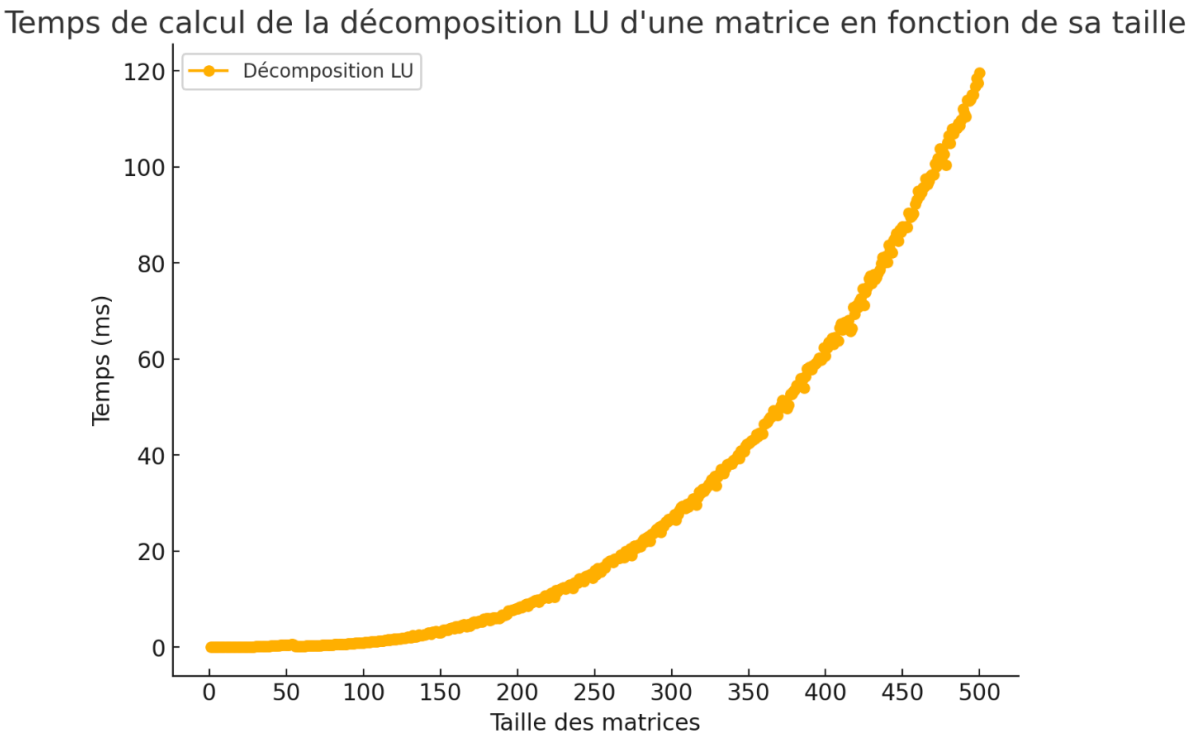
3.2. Multiplication de Strassen



Contrairement à la courbe du produit naïf, celle-ci présente des paliers soudains. Cela s'explique par les redimensionnements à la puissance de 2 la plus proche effectués dans l'algorithme de Strassen. L'algorithme est donc moins efficace pour des matrices dont la taille n'est pas une puissance de 2. Il est aussi moins efficace que l'algorithme naïf pour des tailles de matrices modérées.

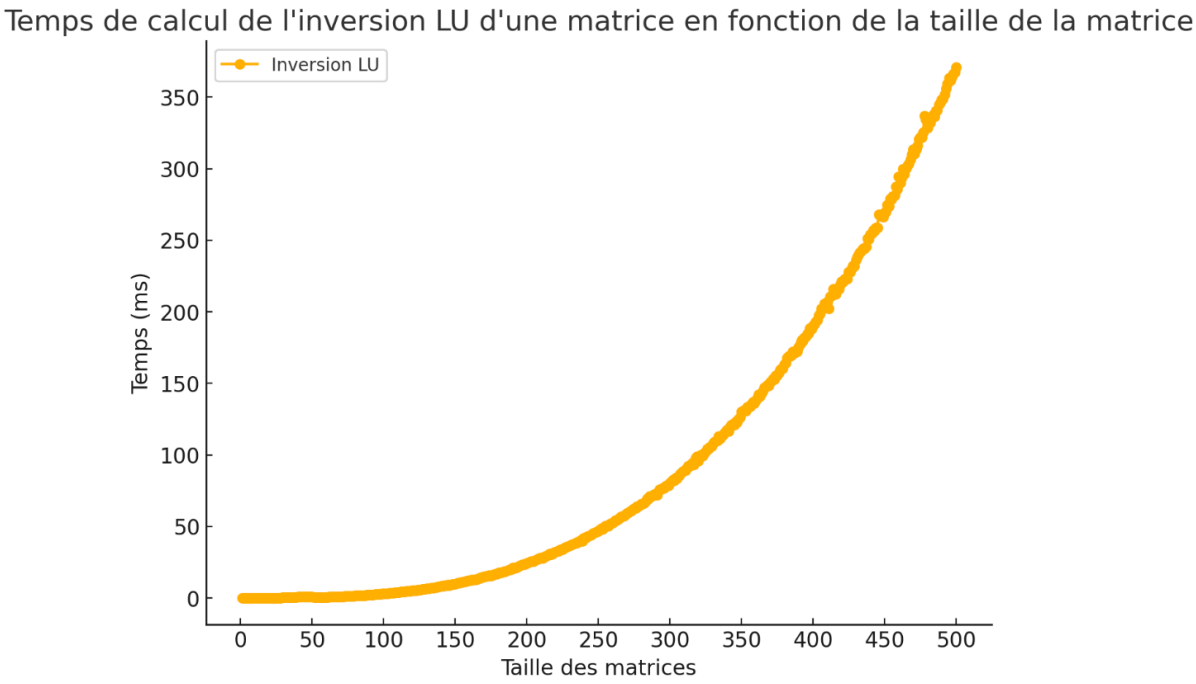
Suite à des problèmes de gestion mémoire, nous n'avons pas pu appeler la méthode ***strassen_product*** sur des matrices de plus grandes tailles, ce qui a compromis la comparaison complète des deux méthodes.

3.3. Décomposition LU



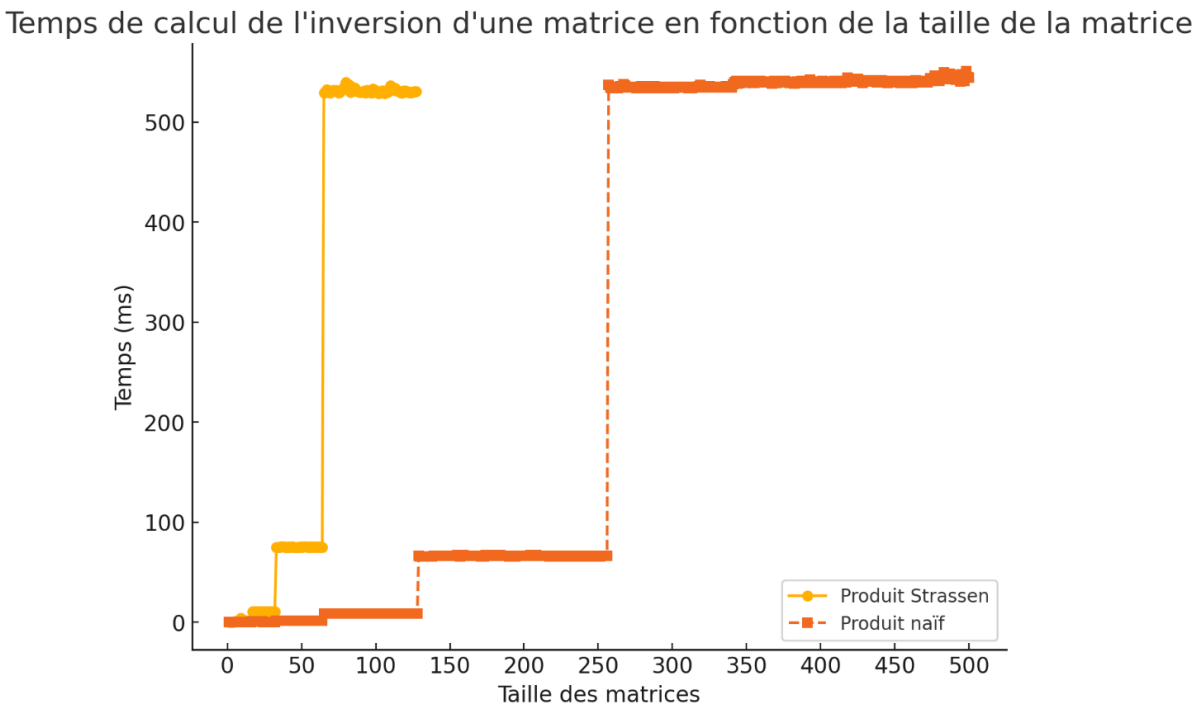
On observe que la courbe a une allure exponentielle, ce qui est cohérent avec une complexité cubique **O(n^3)**. La courbe est lisse et continue car la décomposition LU ne nécessite pas de redimensionner la matrice à une taille de puissance de 2. Les temps de calcul sont ainsi en accord avec la complexité attendue.

3.4. Inversion par LU



La courbe montre une croissance exponentielle caractéristique d'une complexité en **$O(n^3)$**, similaire à la décomposition LU. Cette croissance est normale et justifiée car la décomposition LU nécessite plusieurs opérations coûteuses (décomposition en deux matrices, résolution de deux systèmes triangulaires). On observe que l'inversion LU est plus coûteuse que la décomposition LU, en raison des étapes supplémentaires pour obtenir la matrice inverse.

3.5. Inversion par Strassen



La courbe du produit de Strassen présente de grands sauts dans ses temps de calculs, que nous avons expliqués précédemment par le redimensionnement des matrices. La courbe du produit naïf présente également des sauts pour les mêmes raisons.

Pour les petites matrices, Strassen est légèrement plus lent que le naïf en raison des appels récursifs. Le produit de Strassen introduit un surcoût significatif pour des tailles qui ne sont pas des puissances de 2. Cependant, pour des matrices de plus grande taille, Strassen devient graduellement de plus en plus efficace comparée à la version naïve de cet algorithme.

4. Pistes d’amélioration

Comme vu précédemment avec les courbes, nous obtenons des temps de calcul plus efficaces pour le produit naïf par rapport au produit de Strassen. Comment expliquer cette différence, alors que l'algorithme de Strassen nous promet une complexité en **$O(n^{2.81})$** ?

Plusieurs facteurs peuvent expliquer cela :

- Strassen est un algorithme **diviser pour régner**, il sera donc graduellement plus efficace au fur et à mesure que la taille des données augmente. La récursivité introduit un coût important dû aux appels de fonction, aux phases de division ou encore celles de combinaisons et à la gestion de la mémoire. Les divisions de matrices et les combinaisons des résultats ajoutent beaucoup d’étapes supplémentaires, qui sont plus gourmandes que les opérations du produit naïf pour des matrices de petites tailles.
- Strassen nécessite 7 produits récursifs au lieu de 8 pour le produit naïf, mais il introduit 18 additions/soustractions pour la reconstruction de la matrice résultat. **Ainsi, lorsque les matrices sont petites, le coût de ces opérations peut l'emporter sur le gain de 7 produits.**
- Le produit naïf accède séquentiellement aux éléments des matrices ce qui optimise l’utilisation du cache CPU, tandis que le produit par Strassen divise les matrices, ce qui peut causer une **fragmentation mémoire** et donc une efficacité moindre du cache.

Strassen devient donc intéressant pour des matrices de tailles très importantes car le coût des multiplications domine alors le coût des additions.

Solutions possibles pour améliorer Strassen :

- Il pourrait être intéressant d’introduire un **seuil** pour déterminer quand utiliser le produit naïf et quand utiliser le produit de Strassen. Tant que la taille des matrices reste en-dessous dudit seuil, on utilise l’algorithme naïf. Une fois le seuil dépassé, on utilise alors l’algorithme de Strassen pour optimiser le temps d'exécution des calculs.
- Une autre option pourrait être d’implémenter une **version parallèle de Strassen**, pour paralléliser les 7 produits récursifs.

5. Conclusion

Ce projet nous a permis de **confirmer par nous-mêmes** les résultats théoriques présentés en cours. En particulier, nos expérimentations montrent que les méthodes **naïves** restent relativement efficaces pour les matrices de petite taille, tandis que les méthodes basées sur **Strassen** sont généralement moins performantes dans ces mêmes cas en raison de leurs surcoûts.

Cependant, à mesure que la taille des matrices augmente, les algorithmes utilisant **Strassen** deviennent progressivement plus efficaces et peuvent parfois surpasser les algorithmes naïfs. Bien qu'un tel phénomène n'ait pas pu être constaté pour la multiplication en raison des **limitations mémoire** de nos machines, il est raisonnable de penser que le même comportement apparaît pour cette opération sur des matrices de très grande taille.

Par ailleurs, l'algorithme de Strassen ouvre la voie à de **nouvelles approches d'optimisation**, telles que :

- la **parallélisation**, qui permettrait de diviser les calculs récursifs ;
- l'utilisation d'un **seuil adaptatif**, permettant de tirer parti du meilleur des deux approches : utiliser la méthode naïve pour les petites matrices et basculer vers Strassen pour les grandes tailles.

Ainsi, ce projet nous a permis de **mesurer les avantages** et les **limites pratiques** des deux types d'algorithmes tout en identifiant des pistes d'amélioration.