SCIENCES SORBONNE UNIVERSITÉ

Projet MODEL: Implémentation et Analyse

FFT, Algorithmes de multiplication et Arithmétique des Nombres Complexes

Samy Horchani (n° étudiant : 28706765)
Département d'informatique - Sorbonne Université FSI
M1 CCA - Décembre 2023

Projet réalisé dans le cadre de l'UE :
Numerical and Symbolic Algorithms Modeling (MODEL, MU4IN901)

Contents

1 Introduction

Ce projet réalisé dans le cadre de l'UE de MODEL, se concentre principalement sur l'implémentation et l'analyse des algorithmes de Transformée de Fourier Rapide (FFT) et mets en lumière leurs importance dans des applications pratiques telles que la multiplications de polynômes. De plus, ce projet vise également à approfondir des concepts mathématiques et algorithmiques sous-jacent à l'arithmétique des nombres complexes en développant une bibliothèque permettant de travailler avec des nombres complexes.

Le rapport suivant détaille l'implémentation des algorithmes mentionnés, en commençant par une description de l'organisation et de la structure du code. Il présente ensuite une comparaison entre l'algorithme naïf et l'approche basée sur la FFT pour la multiplication de polynômes, soulignant l'efficacité accrue apportée par la FFT. Des benchmarks sont fournis pour illustrer les performances des implémentations, offrant une analyse critique de leurs efficacités respectives.

2 Organisation du code

Ce projet a été structuré de manière à favoriser la clarté, la modularité et la facilité de maintenance du code. Voici une description détaillé de cette structure :

2.1 Répertoire src

Le répertoire "src" contient tous les fichiers source ".c" ui implémentent les fonctions principales du projet. Chaque fichier source se concentre sur une partie spécifique de l'implémentatation, garantissant ainsi, une séparation claire des différentes composantes du projet.

- **main.c**: Point d'entrée du programme. Il gère l'initialisation et coordonne les appels aux différentes fonctions implémentées.
- **complex.c**: Implémentation des opérations arithmétiques sur les nombres complexes ainsi que certaines fonctions utiles sur les nombres complexes (conversion d'un entier ou d'un tableau d'entier en complex, affichage d'un complexe au format $ai + b$, ...) .
- **fft.c**: Contient l'algorithme de la transformée de Fourier rapide (FFT) et son inverse.
- **multiplyPolynomials.c**: Contient l'algorithme de multiplication naïve ainsi que l'algorithme de multiplication basé sur la FFT.

2.2 Répertoire include

Dans le répertoire "include", on retrouve les fichiers d'en-tête ".h". Ces fichier contiennent les déclarations des fonctions, structures et macros utilisées dans ce projet. Chaque fichier .h porte le même noms que son fichier .c associé. Chaque prototype de fonction est accompagné par un commentaire rappelant l'utilité de celle-ci.

2.3 Répertoire build

Le répertoire "build" est destiné aux fichiers objets ".o" générés lors de la compilation et sont utilisés pour construire les executables finaux. Le fait de placer ces fichiers dans un répertoire distinct aide à maintenir l'environnement de développement propre et organisé.

2.4 Repertoire benchmark

Ce dossier contient les scripts, les fonctions ainsi que les données utilisées dans la réalisations des benchmarks.

- **benchmarks.c**: Ce fichier contient les fonctions et le code permettant de mesurer et relever les données nécessaires à l'analyse des performances des fonctions implémentées.

- **data_FFT.txt, data_NAIVE.txt, ...** : données résultants des tests de performances.

2.5 Fichier Makefile

Le projet inclut un fichier "Makefile" qui automatise le processus de compilation. Ce fichier définit des règles pour la construction du projet, y compris la compilation des fichiers source, la génération des fichiers objets, et la création de l'exécutable final. Il s'assure également de la présence du dossier build nécessaire à la compilation des fichiers objets. L'utilisation du Makefile permet une compilation cohérente et simplifie les étapes nécessaire pour construire le projet, quel que soit l'environnement de développement.

2.6 Les exécutables

- **main**: L'exécution de cet exécutable prend la forme d'un menu, il permet de pouvoir tester les différentes fonctions développées en calculant la FFT, son inverse mais en réalisant également la multiplication de polynômes avec les deux approches.

```
$ ./main
```

- **benchmark**: Il permet de lancer les tests et de récupérer les données afin de pouvoir tracer les différents graphiques.

```
$ ./benchmark
```

2.7 Fichier README.md

Le fichier README.md joue un rôle crucial dans la documentation du projet. Il fournit des instructions de base sur la façon de compiler et d'exécuter le code. Ce fichier est rédigé en format Markdown, ce qui permet une lecture aisée tant dans les éditeurs de texte que sur les plateformes de gestion de code source comme GitHub. Le README inclut des informations sur les dépendances du projet, les étapes de compilation, et des exemples d'utilisation du programme.

2.8 Fichier commande.gnuplot

Le fichier commande.gnuplot contient les commandes permettant de construire les différents graphiques avec gnuplot. après avoir généré les données. Il suffit d'executer la commande ci-dessous.

```
$ gnuplot commande.gnuplot
```

3 Implémentation

Voici le pseudo-code des principales fonctions que j'ai implémenté :

3.1 Pseudo code Multiplication naïve

L'algorithme de multiplication naïve présenté se base sur l'approche classique du produit de deux polynômes. Le pseudo-code décrit une implémention directe de cette méthode, sans optimisations supplémentaires. **Entrée** : deux tableaux d'entier $poly1$ et $poly2$ représentant les coefficients des polynômes, deux entiers $size\_p1$ et $size\_p2$ étant leur taille respective et un pointeur vers un entier $size\_res$ qui contiendra la taille du polynôme résultant.
Sortie : Un tableau d'entier : polynôme résultant ainsi que la taille de celui-ci dans l'entier $size\_res$.

Tout d'abord, la taille du tableau résultant est déterminée par la somme des degrés des polynômes multipliés, puis, chaque coefficient du polynôme résultant est obtenu en sommant les produits des coefficients des polynômes d'entrée, décalés en fonction de leurs indices.

Algorithm 1: Multiplication naïve

```
Function naiveMultiplyPolynomials (poly1, size_p1, poly2, size_p2, size_res)
size_res ← size_p1 + size_p2 − 1 ;
Allocation d'un tableau result de taille size_res initialisé avec des zéros;
for i = 0 → size_p1 − 1 do
    for j = 0 → size_p2 − 1 do
        result[i + j] ← result[i + j] + poly1[i] ∗ poly2[j] ;
    end
end
return result;
```

3.2 Pseudo code FFT

Le pseudo-code fourni décrit les deux algorithmes liés à la mise en œuvre de la Transformée de Fourier rapide (FFT) : une version récursive de la FFT et la fonction principale de la FFT qui prépare les données et appelle la version récursive. L'algorithme FFT récursif est une implémentation classique de la FFT. Il décompose le problème en sous-problèmes plus petits en divisant le vecteur d'entrée en deux parties à chaque étape récursive : une pour les éléments pairs et une pour les éléments impairs. Cette méthode est basée sur le fait que la FFT d'un vecteur peut être reconstruite à partir de la FFT de ses moitiés pairs et impairs.

3.2.1 Fonction FFT recursive

Algorithm 2: FFT recursive

```
Function FFT_rec(v, n, res, omega, step)
if n = 1 then
    res[0] ← v[0];
    return;
end
Calcul de omega ∗ omega;
Appel de la FFT_rec sur la première moitié de V avec omega^2;
Appel de la FFT_rec sur la seconde moitié de V avec omega^2;
omega_i ← 1;
for i = 0 → n/2 − 1 do
    p ← res[i];
    q ← [i];
    res[i] ← p + q;
    res[i + n/2] ← p − q;
    omega_i ← omega ∗ omega_i;
end
```

Entrée : un tableau de nombres complexes v, n : taille de v (entier), res : contenant le tableau résultat de taille n, $omega$: un double qui correspond à l'omega calculé dans l'algorithme 3 : FFT et $step$: un entier aidant à sélectionner les éléments corrects pour chaque appel récursif.
Sortie : $void \rightarrow$ calcul de la FFT dans le tableau res.

- `Base de la Récursion` : La récursion s'arrête lorsque le vecteur d'entrée est réduit à un seul élément, auquel cas la FFT de ce vecteur est lui-même.
- `Décomposition` : Pour les vecteurs plus longs, l'algorithme calcule d'abord la FFT des moitiés pairs et impairs séparément.
- `Combinaison` : Une fois que la FFT des moitiés ont été calculées, l'algorithme combine ces résultats pour produire la FFT du vecteur original.

3.2.2 Fonction FFT

Algorithm 3: FFT

```
1  Function FFT(v, size_v, size_res)
2  size_res ← power_of_2(size_res);
3  Allocation d'un tableau de nombre complexe new_v de taille size_res;
4  Copie des éléments de v dans new_v;
5  Initialisation à 0 des éléments de new_v restant;
6  Allocation d'un tableau résultat result de taille size_res;
7  omega ← e^(2πi / size_res);
8  Appel de FFT_rec(new_v, size_res, result, omega, 1);
9  Libération de la mémoire allouée à new_v;
10 return result;
```

Entrée : un tableau de nombres complexes v, $size\_v$: taille de v (entier) ainsi que $size\_res$, un pointeur vers un entier.
Sortie : Un tableau de de nombres complexes $result$ contenant le résultat de la FFT de v ainsi que la taille de celui-ci dans $size\_res$.

L'algorithme FFT est la fonction principale qui prépare les données et orchestre le processus de calcul de la FFT en utilisant des fonctions récursive.

- `Préparation des Données` : La fonction calcule la taille nécessaire pour le vecteur de sortie et prépare un nouveau vecteur de cette taille, en s'assurant qu'il est une puissance de deux pour répondre aux exigences de la FFT. Cette opération est réalisé grâce à la fonction $power_of_2$ qui renvoie l'entier donnée en argument si celui-ci est bien une puissance de 2 ou renvoie la prochaine puissance de 2.
- `Initialisation et Allocation` :Le nouveau vecteur est initialisé et rempli avec les valeurs du vecteur d'entrée. Les espaces restants sont remplis avec des zéros si la taille du vecteur d'entrée n'est pas une puissance de deux.
- `Calcul de la FFT` : L'algorithme principal de la FFT est appelé avec le nouveau vecteur préparé.

Attention : L'entier size_res contiendra à la fin de l'exécution de l'algorithme de FFT, la taille du tableau contenant la FFT. Il sert au départ à indiquer la n-ième racine de l'unité voulu (ligne 7) et il est donc important celui-ci soit un nombre égale à la taille du vecteur v (size_v) ou dans le cas de la multiplication basée sur la FFT, à un nombre >= au degré des 2 polynômes multiplié afin de ne pas provoquer de segfault.

3.3 Pseudo code FFT inverse

Algorithm 4: FFT Inverse

```
1 Function invFFT(v, size_v, size_res)
2 Conjuguer tous les éléments de v;
3 res ← FFT(v, size_v, size_res) ;
4 for i = 0 → size_res do
5 |   Conjuguer res[i];
6 |   result[i] ← result[i]/size_v;
7 |   Conjuguer v[i] pour retrouver les éléments originaux de v;
8 end
9 return result
```

Entrée : L'algorithme prend les mêmes entrées que l'algorithme de FFT (Algorithme 3) : un tableau de nombres complexes v, $size\_v$: taille de v (entier) ainsi que $size\_res$, un pointeur vers un entier.
Sortie : Un tableau de de nombres complexes $result$ contenant le résultat de la FFT inverse de v ainsi que la taille de celui-ci dans $size\_res$.
Attentions : Les mêmes conditions s'applique au pointeur $size\_res$ que pour l'algorithme 3 FFT.

- `Conjugaison de v et calcul de la FFT` : L'algorithme principal de la FFT est appelé avec le nouveau vecteur conjugué.
- `Traitement du résultat`: Chaque élément du résultat de l'appel à la fonction FFT est alors conjugué et divisé par la taille du vecteur v.

3.4 Pseudo code Multiplication basé sur la FFT

Algorithm 5: Multiplication basé sur la FFT

```
1  Function fftMultiplyPolynomials (poly1, size_p1, poly2, size_p2, size_res)
2  size_res ← size_p1 + size_p2 − 1 ;
3  Conversion du tableau d'entier P en tableau de nombres complexes P_complex;
4  Conversion du tableau d'entier Q en tableau de nombres complexes Q_complex;
5  FFT_P ← FFT(P_complex, size_P, size_res);
6  FFT_Q ← FFT(Q_complex, size_Q, size_res);
7  Allocation d'un tableau FFT_res de taille size_res;
8  for i = 0 → size_res − 1 do
9  |  FFT_res[i] ← FFT_P[i] * FFT_Q[i]
10 end
11 invFFT_res ← invFFT(FFT_res, size_res, size_res);
12 Allocation d'un tableau d'entier result de taille size_res initialisé avec des
     zéros;
13 for i = 0 → size_res − 1 do
14 |  result[i] ← round(invFFT_res[i].real)
15 end
16 libération de la mémoire utilisée par P_complex, Q_complex, FFT_P, FFT_Q,
     FFT_res et invFFT_res;
17 return result;
```

Entrée : Il prend les mêmes paramètres que l'algorithme 1 - naïf : deux tableaux d'entier $poly1$ et $poly2$ représentant les coefficients des polynômes, deux entiers $size\_p1$ et $size\_p2$ étant leur taille respective et un pointeur vers un entier $size\_res$ qui contiendra la taille du polynôme résultant.
Sortie : Un tableau d'entier : polynôme résultant ainsi que la taille de celui-ci dans l'entier $size\_res$.
Le pseudo-code présenté décrit l'implémentation de la multiplication de deux polynômes en utilisant la méthodes basée sur la Transformée de Fourier rapide telle que décrite dans le polycopié de cours :

- `Calcul de la FFT` : appel de la FFT sur les polynômes $poly1$ et $poly2$ avec ω une n-ième racine de l'unité avec `n` > `degré de poly1` + `degré de poly2`
- `Multiplication des FFT` : Chaque coefficient des FFT des deux polynômes est multiplié afin d'obtenir la FFT de `R = poly1` * `poly2`
- `Calcul de la FFT inverse de R`

4 Benchmarks

4.1 Environnement de tests et méthodologie

- Type de Processeur : Apple M1 (ARM 64 bit)
- Mémoire RAM : 8 Go
- Système d'exploitation : macOS 14.2.1
- Compilateur C : clang 15.0.0
- Débogueur : Valgrind et Callgrind/Xcode Instruments : time profiler

Il est important de noter que les performances peuvent varier en fonction de l'environnement matériel et logiciel.

Les benchmarks ont été réalisés en mesurant le temps d'exécution des fonctions pour différentes tailles d'entrée allant de matrice de taille 1 à des vecteurs de taille 10000 afin de couvrir une gamme étendue de cas d'utilisation. Les mesures ont été répétées plusieurs fois (5) pour chaque taille pour assurer la précision des résultats et une c'est la moyenne de temps obtenu chaque entrée qui a été utilisée dans les graphiques suivant.

4.2 Complexité et performance théoriques

La partie suivante est consacrée à une analyse approfondie de la complexité algorithmique et des performances théoriques des méthodes de multiplication de polynômes étudiées. Nous examinerons les caractéristiques intrinsèques de chaque algorithme, en mettant en lumière les implications théoriques de leur utilisation dans différents contextes de taille d'entrée, et en présentant une série de graphiques générés pour illustrer ces concepts.

Pour l'algorithme de multiplication naïf, la complexité temporelle a été modélisée comme étant proportionnelle au carré du degré du polynôme, reflétant la nature quadratique de l'approche. En ce qui concerne l'algorithme FFT, la complexité a été estimée sur la base de l'opération la plus coûteuse, à savoir la transformée de Fourier rapide, dont la performance théorique est de l'ordre de $n \log n$, où n représente la prochaine puissance de deux supérieure au degré du polynôme.

4.2.1 Algorithme de multiplication naïf

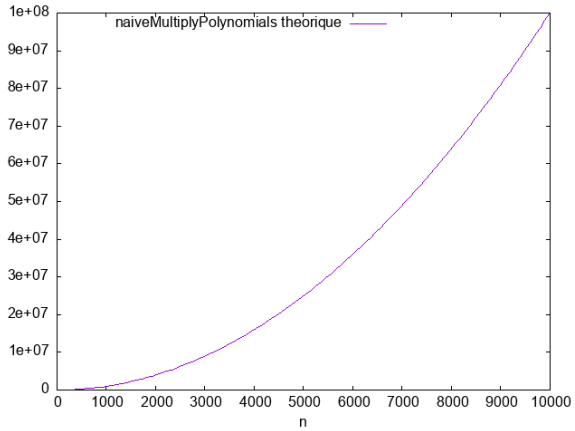

Figure 1: Complexité théorique l'algorithme de multiplication naïve ($n \log n$) en fonction de la taille de l'entrée `n * n`

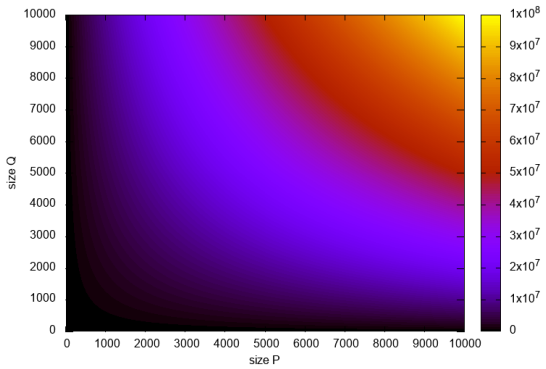

Figure 2: Graphique 2D de la complexité théorique de l'algorithme de multiplication naïve en fonction de la taille des polynômes P et Q.

L'algorithme de multiplication naïf est basé sur la méthode classique d'appariement terme à terme des coefficients des polynômes. La complexité de cet algorithme est quadratique $O(n^2)$, ce qui signifie que le temps de calcul augmente de manière proportionnelle au carré de la taille de l'entrée. Les résultats théoriques, représentés graphiquement, montrent une croissance exponentielle du temps nécessaire pour les grandes tailles de polynômes, soulignant ainsi l'inefficacité de cette approche pour des calculs de grande envergure.

4.2.2 Algorithme de multiplication utilisant la FFT

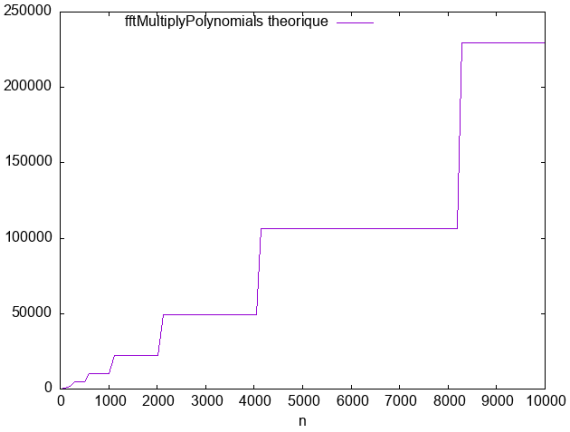

Figure 3: Complexité théorique l'algorithme de multiplication avec FFT $(n \log n)$ en fonction de la taille de l'entrée $\texttt{n} * \texttt{n}$

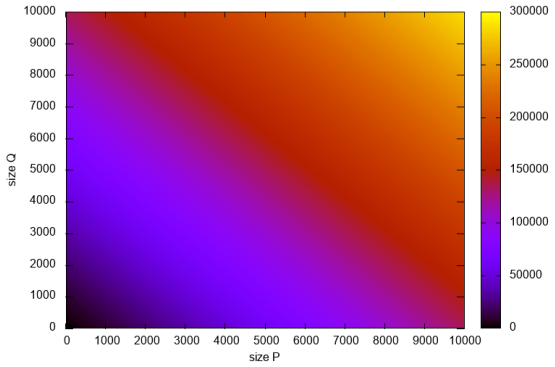

Figure 4: Graphique 2D de la complexité théorique de l'algorithme de multiplication basé sur la FFT en fonction de la taille des polynômes P et Q.

À l'opposé, l'algorithme de multiplication utilisant la transformée de Fourier rapide (FFT) doit-être beaucoup plus performant pour les grandes entrées. La FFT réduit la complexité de la multiplication de polynômes de n à $O(n \log n)$ en utilisant une approche "divide and conquer". Le graphique correspondant illustre une augmentation beaucoup moins rapide du temps de calcul avec la taille de l'entrée, ce qui fait de la FFT une méthode de choix pour des applications nécessitant des multiplications polynomiales rapides et efficaces.

4.3 Résultats et Analyse

Voici les données obtenus après mesures des temps des fonctions de multiplications en fonction de la taille des entrées.

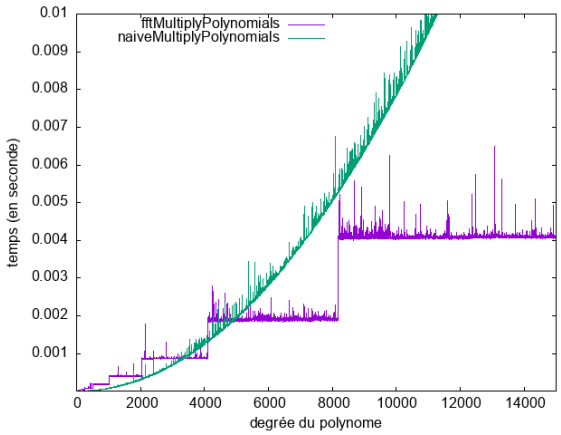

Figure 5: Évolution du temps en fonction de la taille de l'entrée

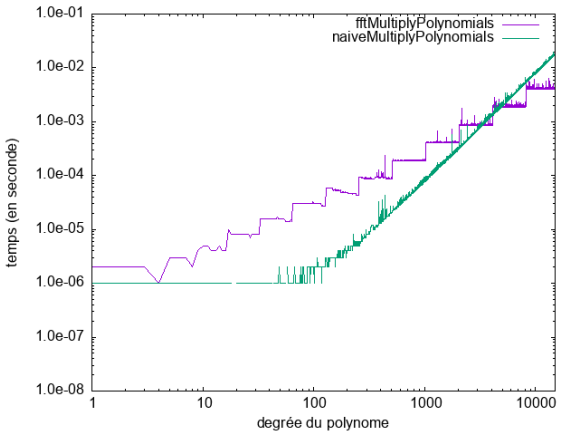

Figure 6: Utilisation d'une échelle logarithmique

Le premier graphique (Échelle Linéaire) montre l'évolution du temps de calcul en fonction du degré du polynôme. La courbe liée à l'implémentation FFT semble suivre une tendance linéaire-logarithmique, ce qui est attendu pour la FFT (section 4.2.2). Le temps d'exécution augmente de manière relativement douce avec la taille de l'entrée. De plus, l'allure en "escalier" est dû au fait que notre algorithme de multiplication étend la taille des vecteurs qui ne sont pas des puissances de 2 à la prochaine puissance de 2 supérieure. L'approche naïve, en revanche, montre une augmentation beaucoup plus rapide du temps de calcul à mesure que le degré du polynôme augmente, ce qui est cohérent avec sa complexité algorithmique quadratique $O(n^2)$. Les pics observés peuvent être le résultat de variations de l'environnement d'exécution.

Le second graphique (figure 6) utilise une échelle logarithmique pour l'axe des abscisses et des ordonnées. Elle est particulièrement utile pour visualiser des données sur plusieurs ordres de grandeur et pour mettre en évidence les différences dans les taux de croissance des algorithmes. Avec cette échelle, la courbe FFT semble presque linéaire, ce qui confirme la complexité en $O(n \log n)$, car l'échelle logarithmique transforme les fonctions polynomiales en lignes droites.

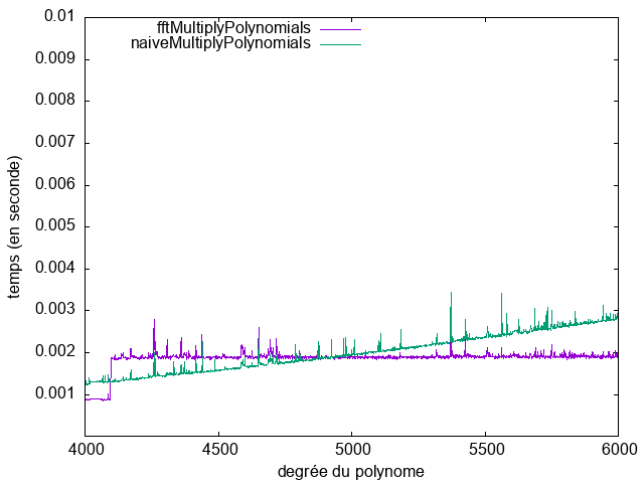

Figure 7: Comparaison détaillée des temps d'exécution

Dans la figure 7, nous observons une comparaison plus détaillée des temps d'exécution de la FFT par rapport à la multiplication naïve. Cette vue rapprochée révèle la taille de polynôme pour laquelle l'algorithme de multiplication basée sur la FFT devient plus rapide ($n \approx 4800$). Cette vue révèle des nuances de comportement comme les pics intermittents dans le temps de calcul pour la multiplication naïve qui suggèrent que des facteurs environnementaux, tels que la gestion de la mémoire et les opérations de cache, peuvent jouer un rôle non négligeable dans la performance mesurée. En contraste, la FFT montre une augmentation plus constante du temps de calcul, ce qui souligne son efficacité pour traiter des données de grande taille. Cependant, même la FFT n'est pas à l'abri de variations occasionnelles, possiblement dues à des opérations de rembourrage pour maintenir les entrées à des tailles optimales.

4.3.1 Avantages et inconvénients - Multiplication Naïve

	Multiplication Naïve	**Multiplication FFT**
Avantages	• Simplicité d'implémentation • Faible coût de mise en place • Pas d'overhead initial	• Efficace pour de grands ensembles de données • Optimisé pour les convolutions • Performances significativement meilleures pour les polynômes de degré élevé
Inconvénients	• Inefficace pour de grands ensembles de données • Plus lent que la FFT pour de grandes tailles	• Plus complexe à implémenter • Sensible aux erreurs d'arrondi, précision à prendre en compte (ici précision sur les double) • Overhead initial dû à la préparation des données et au calcul des racines de l'unité

Table 1: Comparaison des avantages et inconvénients de la multiplication naïve et avec FFT

5 Difficultés rencontrés et pistes d'améliorations

Lors de la phase de benchmarking, un constat inattendu a été fait : la fonction de multiplication avec Transformée de Fourier rapide (FFT) affichait des performances similaires, voire inférieures, à la multiplication naïve des polynômes. Cette observation allait à l'encontre des attentes théoriques, où la FFT devrait nettement surpasser en efficacité la multiplication directe pour des tailles de données significatives.

5.1 Mise à la puissance et formule de Moivre

Grâce à l'emploi d'outils de profilage tels que Callgrind puis Xcode Instruments : time profiler, il est apparu que le goulot d'étranglement résidait dans la mise en puissance utilisée au sein de la FFT par le biais de ma fonction $complex\_power(n)$. L'approche initiale, bien que directe, s'avérait coûteuse en temps de calcul, réitérant inutilement les mêmes opérations à chaque appel de la fonction. La recherche d'une solution plus optimisée m'a conduit à la formule d'Euler de Moivre, qui stipule que :

$$(cos(\theta) + i * sin(\theta))^n = cos(n\theta) + i * sin(n\theta) \tag{1}$$

Cette optimisation permet de calculer la puissance d'un nombre complexe en $O(log(n))$ entraînant une amélioration du temps d'exécution mais, il s'est finalement avéré plus efficace de ne pas utiliser la fonction précédente mais de simplement calculer le i-ème oméga à l'intérieur de la FFT car l'appel de cette fonction ralentissait la multiplication polynomiale.

5.2 Recalcule du Omega

Une autre amélioration a été de ne pas recalculer les omégas à chaques appels de la fonctions FFT_res, ces derniers sont désormais calculés une seule fois et passés en paramètres aux fonctions requises. Cette

optimisation a permis de réduire considérablement le temps de calcul global de la FFT, la rendant plus efficace que l'algorithme naïf sur des entrées très grandes comme en témoignent les benchmarks.

5.3 Pistes d'améliorations

Malgré les améliorations significatives apportées par l'intégration de la formule de Moivre et l'optimisation du recalcul des omégas, le potentiel d'amélioration de notre algorithme de multiplication de polynômes basé sur la FFT reste important. Les pistes suivantes pourraient être envisagées pour accroître encore davantage l'efficacité et la performance de notre implémentation :

- `Parallélisation des Calculs` : La nature des opérations de la FFT se prête bien à la parallélisation en distribuant les multiplication éléments par éléments.
- `Utilisation de Bibliothèques Optimisées` : L'emploi de bibliothèque mathématiques hautement optimisé pour les opérations de FFT et de représentation des nombres complexes pourrait offrir des gains de performance significatifs.

De plus, un profilage régulier du code avec des outils avancés peut aider à identifier les goulots d'étranglement inattendus.

6 Conclusion

Le travail présenté dans ce rapport a mis en lumière les caractéristiques distinctives et les performances de deux approches fondamentales pour la multiplication de polynômes : l'algorithme de multiplication naïve et l'algorithme basé sur la Transformée de Fourier rapide (FFT). Nos expérimentations ont clairement démontré l'efficacité supérieure de la FFT dans le traitement de polynômes de grande taille, validant ainsi la complexité théorique en $O(n \log n)$ par rapport à la complexité quadratique en $O(n^2)$ de la méthode naïve.

À travers l'analyse des résultats obtenus, il a été possible de confirmer que, malgré un surcoût initial associé à la FFT, les avantages en termes de temps d'exécution deviennent significatifs à mesure que la taille des polynômes augmente. Ce constat souligne l'importance d'une sélection judicieuse de l'algorithme en fonction de la taille des données à traiter.

Les optimisations mises en œuvre ont apporté des améliorations notables. Cependant, le chemin vers une efficacité optimale est semé d'obstacles, comme l'ont révélé les pics de performance et les anomalies observées lors des tests.

En conclusion, ce projet a non seulement renforcé notre compréhension des principes algorithmiques sous-jacents à la multiplication de polynômes et de l'algorithme de Transformée de Fourier Rapide mais a également permis d'approfondir ma compréhension de l'arithmétique des nombres complexes. Il est essentiel de continuer à explorer des méthodes d'optimisation et de parallélisation pour améliorer davantage les performances et l'efficacité des calculs.