Master d'Informatique
Modèles de calcul

Multiplication de polynômes

Rapport du projet

Authors
Virginie CHEN
Lasha GOGRITCHIANI
Philippe TAN

7 janvier 2024

Table des matières

1 Introduction

1.1 Mise en contexte

Dans le cadre de l'unité d'enseignement MODEL, nous avons eu l'occasion d'étudier l'algorithme de transformation de Fourier rapide. Un algorithme permettant notamment de pouvoir multiplier deux polynômes de manière bien plus efficace. Il s'agira ici d'implémenter un algorithme de multiplication utilisant la technique naïve, en implémenter un autre reposant sur l'algorithme FFT et comparer leur efficacité.

1.2 Implémentation

Le projet a été réalisé en **C**. Toutefois, nous avons décidé d'utiliser **Python** pour la partie analyse puisqu'il existe un vaste écosystème de bibliothèques disponibles et simple d'utilisation permettant la génération de courbes à partir des données mesurées sur notre programme.

Nous avons organisé notre environnement de travail de la manière suivante :

- **README.md** : contient des informations sur la manière d'utiliser le projet.
- **makefile** : contient toutes les instructions de compilation de notre programme.
- **script.py** : script permettant de générer les courbes pour l'analyse des performances de nos fonctions.
- **include/** : dossier contenant tous les fichiers d'en-tête (format .h).
- **src/** : dossier contenant tous les fichiers code source (format .c).
- **bin/** : dossier contenant tous les fichiers binaires issus de la compilation (format .o).

Il existe plusieurs représentations des polynômes. Dans ce projet il a été choisi de représenter un polynôme de degré d sous forme de tableau de ses coefficients. La taille du tableau est donc $d+1$. Le coefficient en indice i du tableau correspond au coefficient associé au terme de degré i. Par exemple, prenons un polynôme de degré 4 : $3x^4 - 10x^2 + 2x + 1$.

Inidice	0	1	2	3	4
Coefficients	1	2	-10	0	3

2 Algorithme FFT

2.1 Les nombres complexes

L'algorithme FFT reposant sur l'évaluation d'un polynôme en une n-ième racine primitive de l'unité, il fallait pouvoir manipuler celles-ci. Il est donc essentiel d'avoir les outils pour manipuler les nombres complexes. En effet, une racine primitive est un nombre complexe qui s'exprime sous forme exponentielle en : $e^{i\frac{2k\pi}{n}}$ pour un certain k.

2.2 Diviser pour régner

Sans grande originalité, pour implémenter notre algorithme FFT nous avons utilisé la méthode "Divide and conquer". Le principe est de diviser notre tableau en entrée en deux plus petits tableaux (une partie comportant les coefficients pairs, l'autre les impairs). On répète ceci récursivement jusqu'à se retrouver dans le cas de base, un tableau à un seul coefficient. Vient ensuite la partie recombinaison de ces sous-tableaux en évaluant ceux-ci aux racines primitives de l'unité pour chaque étape de division. Finalement, on obtient notre algorithme de FFT récursif. Cet algorithme de base part du postulat que la taille du polynôme est une puissance de 2. En effet, le cas où la taille n'est pas une puissance de 2 est à traiter à part.

2.3 Taille des polynômes

Une pratique assez courante pour considérer les polynômes à taille quelconque est d'élargir le tableau de coefficients en y rajoutant des zéros. Ainsi il faut tout d'abord déterminer la puissance de 2 supérieure la plus proche. Ensuite, on élargit le tableau des coefficients du polynôme jusqu'à avoir une taille correspondant à cette puissance de 2. On remplit les nouvelles cases du tableau avec des 0 et on applique notre algorithme FFT récursif définit précédemment sur ce nouveau tableau.

3 Algorithme de multiplication naïf

L'objectif principal de ce projet est de démontrer l'efficacité de l'algorithme de multiplication de polynômes qui utilise la transformation de Fourier rapide. À cette fin, un algorithme de multiplication naïf a également été implémenté pour servir de référence comparative. Cet algorithme naïf consiste à multiplier chaque terme du premier polynôme avec chaque terme du second, puis à additionner les termes de même degré. De premier abord cela ne paraît pas très long, c'est ce que l'on fait habituellement pour multiplier deux polynômes entre eux. Cependant sur des très grandes instances, il s'avère que cela est bien trop coûteux et qu'il y a un moyen d'aller bien plus rapidement. Effectuons une analyse rapide de la complexité. On considère deux polynômes $P1$ et $P2$ tous deux de taille n (donc de degré $n-1$).

- On multiplie le coefficient de $P1$ avec les n autres de $P2$: n opérations
- On répète l'opération pour les n coefficients de $P1$: n opérations
- On additionne les coefficients de même degré : moins de n opérations

Ce qui nous donne une complexité théorique de $\mathbf{O(n^2)}$.

4 Algorithme de multiplication FFT

La multiplication par FFT repose sur la représentation de polynômes en termes de points (abscisse, image). Il prend en entrée deux polynômes et calcule leur FFT respective. On transforme ainsi nos deux tableaux de coefficients en tableau contenant les évaluations de ces polynômes en plusieurs points (les racines primitives n-ième de l'unité). On multiplie les cases de chaque FFT entre elles deux à deux. On obtient donc le produit des deux polynômes.

Pour récupérer le polynôme en représentation de coefficients, on applique l'inverse FFT qui revient à faire une interpolation. L'inverse FFT n'est autre qu'une FFT évaluée en le conjugué de w, la racine primitive, pour laquelle les résultats seront divisés par n. Il a fallu donc créer la fonction iFFT (inverse FFT) qui est donc complètement similaire au code utilisé pour la FFT.

5 Comparaison de l'efficacité

Dans le but d'approfondir notre analyse, nous avons choisi de tracer des courbes représentant le temps d'exécution moyen de chaque algorithme en fonction de la taille de l'instance. Les données ont été extraites du code en C sous forme de tableau, converti ensuite en fichier. Ce dernier a été lu à l'aide de la bibliothèque pandas de Python, et les courbes ont été tracées avec matplotlib.pyplot. Nous avons également fait usage des bibliothèques numpy et scipy.stats pour des raisons qui seront explicitées ultérieurement.

Pour évaluer et comparer l'efficacité des algorithmes naïf et de multiplication FFT [1], plusieurs étapes ont été suivies.

5.1 Détermination du N_{max}

La première étape consiste à déterminer une certaine valeur de N_{max}. Cette valeur correspond à la taille d'instance pour laquelle l'exécution de l'algorithme excède une seconde. On prendra donc le minimum entre l'algorithme naïf et l'algorithme FFT. Autrement dit :

$$N_{max} = min(N_{max_{naïf}}, N_{max_{fft}})$$

Ce N_{max} nous sera utile pour le traçage de courbes. En effet, pour éviter une erreur statistique trop importante, nous faisons la moyenne sur plusieurs instances pour chaque point. Ainsi, cette manière de déterminer les points de la courbe est bien plus longue. Il faut donc choisir un pire cas de temps d'exécution pour les algorithmes d'environ 1 seconde. Ceux-ci étant exécutés une dizaine de fois, le temps qu'il faut pour obtenir nos données devient rapidement long.

N_{max} dépendant de nos ordinateurs personnels, on trouvera dans le tableau 1 les valeurs trouvées respectivement par chacun.

Naïf		
Nom	Taille	Temps
Lasha	23000	1.03
Virginie	26000	1.01
Philippe	24000	1.01

(a) Résultats pour l'algorithme naïf

FFT		
Nom	Taille	Temps
Lasha	65000	0.18
Virginie	65000	0.14
Philippe	65000	0.19

(b) Résultats pour l'algorithme FFT

TABLE 1. Valeurs de $N_{\max}$ trouvées

Au vu des résultats de l'algorithme naïf en tableau 1, nous avons choisi $N_{max} = 25000$.

1. Pour plus de légèreté dans la lecture, nous ferons référence à l'algorithme de multiplication se basant sur la FFT simplement comme "l'algorithme FFT".

Il est important de noter que, lors des tests avec l'algorithme FFT, la taille des polynômes a été limitée à 65 000 en raison d'une "Segmentation Fault" survenue durant l'exécution. Cette erreur est probablement liée à la manipulation d'un tableau de 65000 [2] cases. Cependant, puisque le $N_{\max}$ recherché est clairement inférieur, avoisinant les 20000, il n'est pas nécessaire d'aller au-delà de cette taille pour les besoins de ce projet. Ainsi, nous ne chercherons pas à résoudre ce problème de dépassement de mémoire et nous nous concentrerons plutôt sur l'analyse de la complexité qui va suivre.

5.2 Nombres de points et moyenne

Ensuite, nous choisissons le nombre de points selon la précision qu'on souhaite avoir de notre courbe. En l'occurence, il a été choisi de diviser N_{max} par 25, correspondant à 25 points. Un pas qui est donc de 1000.

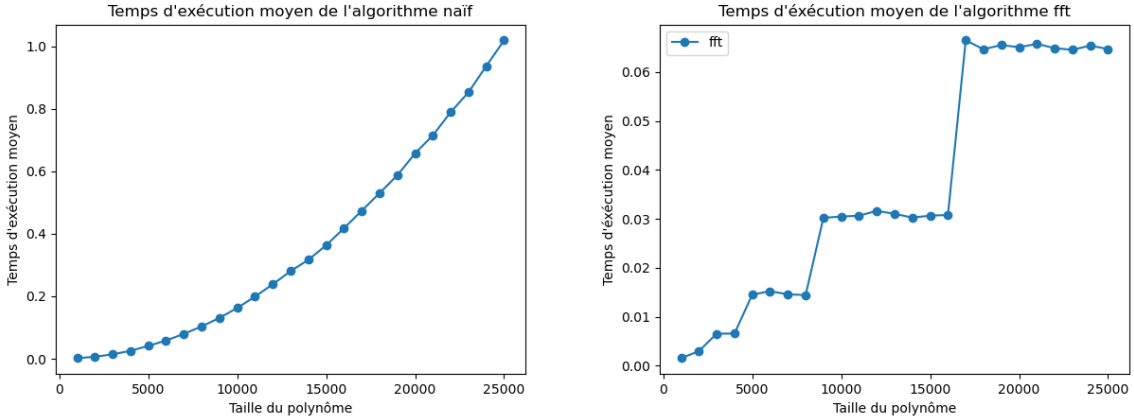
Le nombre d'instances sur lequel nous faisons la moyenne a été fixé à 15. Il a été choisi de sorte à ce que la courbe soit précise tout en ayant un temps d'exécution raisonnable.

5.3 Analyse des courbes obtenues

Finalement, en considérant toutes les données suivantes :

- $\mathbf{N_{max}} = 25000$
- **Nombres de points** : 25
- **Nombres d'instances considérées pour la moyenne** : 15

On obtient les courbes de la figure 1.



(a) Algorithme naïf (b) Algorithme FFT

FIGURE 1. Courbes du temps d'exécution moyen en fonction de la taille du polynôme

On remarque immédiatement que sur la figure 1b, la courbe a une allure assez curieuse. En effet, on observe des anomalies pour certaines tailles de polynômes où le temps d'exécution moyen augmente soudainement pour ensuite reprendre une augmentation plus stable. Ces sauts sont la conséquence de notre algorithme FFT traitant les tailles de polynômes

2. Le tableau du polynôme résultant étant donc trop grand.

quelconques[3]. Pour considérer ces tailles de polynômes, nous agrandissons le tableau des coefficients du polynôme à la puissance de 2 supérieure. Comme explicité dans la section 2.3. Ainsi, pour toutes les tailles de polynômes quelconques entre deux puissances de 2, l'algorithme FFT est appliqué sur la puissance de 2 prochaine.
Prenons par exemple $2^{13} = 8192$ et $2^{14} = 16384$. A partir de la taille 8193 jusqu'à 16384 inclus, l'algorithme FFT est appliqué sur un tableau de taille 16384. Alors que pour toutes les tailles entre $2^{12} = 4096$ et 8192, l'algorithme FFT était appliqué sur des tableaux de taille 8192. Cette hypothèse sera vérifiée ultérieurement dans le rapport. Tout d'abord, analysons la courbe de l'algorithme naïf.

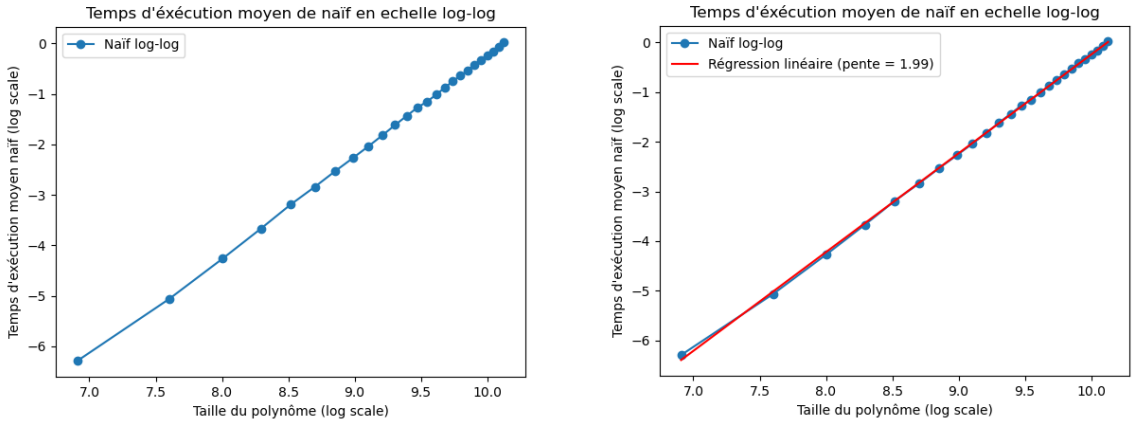
5.3.1 Courbe de l'algorithme naïf

La courbe obtenue semble être polynomiale. Pour vérifier si tel est le cas, et par la même occasion déterminer le degré de celui-ci, nous allons utiliser les propriétés du logarithme :

$$y = x^n \equiv \log(y) = \log(x^n) = n\log(x), \quad \text{où n est le degré du polynôme}$$

Autrement dit, si nous appliquons le logarithme sur les deux axes, abscisses et ordonnées, nous devrions trouver une droite qui a pour pente le degré du polynôme. Nous allons donc passer échelle logarithmique sur les deux axes pour obtenir la courbe en figure 2a. Cette dernière est une droite ce qui nous confirme qu'il s'agit bien d'un polynôme. Nous procéderons à une régression linéaire afin d'ajuster les points et déterminer le coefficient directeur. Le résultat est observé en figure 2b.

Ainsi, il est possible de conclure sur la nature de la courbe obtenue figure 1a. D'après nos calculs, ce serait donc un polynôme de degré 1.99. Un résultat tout à fait satisfaisant et cohérent avec notre complexité théorique qui était pour rappel de $O(n^2)$.



(a) Echelle logarithme (b) Échelle logarithme avec le calcul de pente

FIGURE 2. Courbes passées en échelle logarithme

5.3.2 Courbe de l'algorithme FFT

Commençons par vérifier l'hypothèse que nous avons établie précédemment. Pour cela, nous avons essayé de tracer la même courbe mais en ne considérant que les tailles de

3. Les tailles qui ne sont pas une puissance de 2.

polynômes puissance de 2. Ceci devrait en théorie, si notre hypothèse est correcte, effacer nos anomalies. On observe sur la figure 3 la courbe en question. Nous avons choisi d'aller jusqu'à 2^{15} pour atteindre le N_{max} [4] de la courbe originelle.

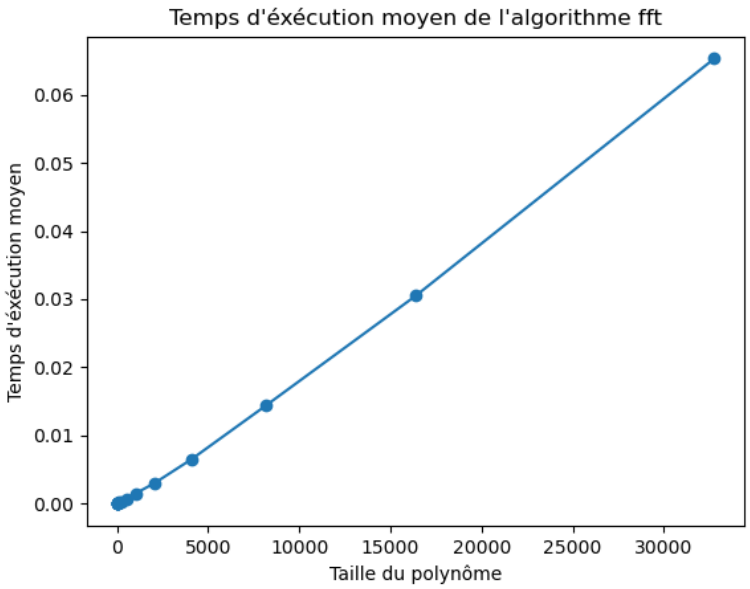


FIGURE 3. Courbe du temps moyen en fonction des tailles puissances de 2

Cette nouvelle courbe efface clairement les anomalies observées précédemment. Ainsi, notre hypothèse sur les points de stagnation semble être correcte. Cela resterait donc pertinent d'étudier l'évolution de cette courbe en guise de comparaison avec la courbe pour l'algorithme naïf. En effet, les points entre deux puissances de 2 restent à peu près aux même temps d'exécution. Leur fluctuation résidant dans le caractère aléatoire des polynômes étudiés. Par conséquent, le fait de négliger [5] ceux-ci n'entrave pas fortement à la justesse de l'analyse.

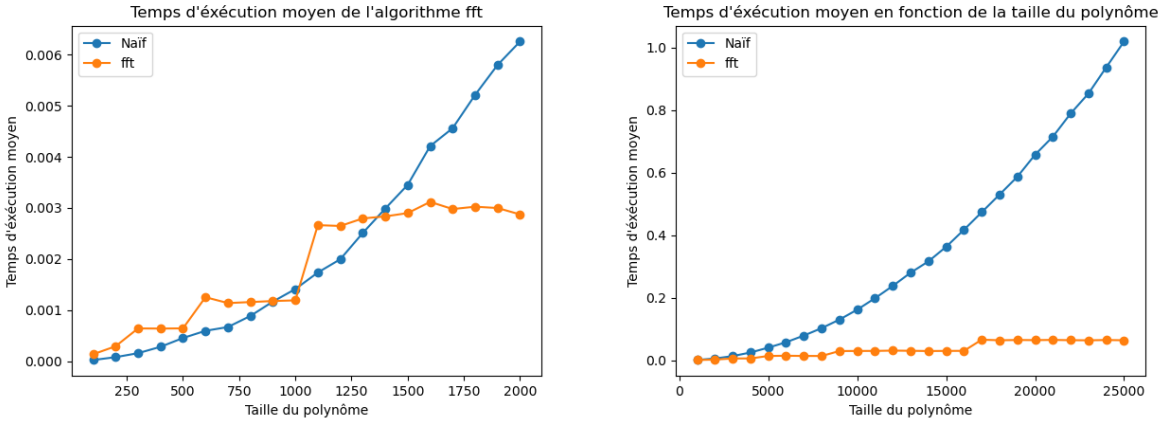
5.3.3 Comparaison

Sur la figure 1 il n'est pas difficile de constater que plus on augmente la taille du polynôme, plus l'algorithme de multiplication naïf devient lent comparé à fft. Pour mettre en évidence de la meilleure façon cet écart de croissance, nous avons jugé utile d'observer le comportement du temps d'exécution de chacun en les traçant sur un même graphe. On observe ce graphe en figure 4b. Ainsi, la croissance de fft comparée à celle de naïf est loin d'être polynômiale. Elle semble être même plus efficace qu'une complexité linéaire. En conséquence, multiplier en utilisant la fft est largement plus efficace.

Néanmoins, nous avons remarqués que cela ne semble pas toujours être le cas. En effet, lorsque nous considérons un polynôme de taille inférieur à en moyenne $N_{critic} = 930$, l'algorithme naïf réussit à multiplier les polynômes plus rapidement. Cette valeur est déterminée grâce à la fonction **find_critical_size_mean()** qui cherche la taille de polynôme pour laquelle fft est pour la **première fois** plus rapide que naïf. Cependant,

4. Pour rappel, $N_{max} = 25000$.
5. ne pas les avoir tracés.

comme nos polynômes sont aléatoires, cette valeur peut varier dû aux instances considérées. On choisira donc de faire une moyenne sur 50 exécutions. Pourtant la taille de polynôme max choisie pour tracer nos courbes sera de 2000. En effet, la réelle valeur pour laquelle fft devient définitivement plus rapide est aux alentours de 1400. Nous avons choisi, afin de pouvoir observer clairement ce changement au niveau des courbes, d'aller un peu plus loin que N_{critic}. On observera ce phénomène sur la figure 4a.



(a) $N_{max} = 2000$, 20 points. (b) $N_{max} = 25000$, 25 points.

FIGURE 4. Graphes de comparaisons

On peut donc conclure cette partie sur la comparaison de l'efficacité de nos deux algorithmes de la manière suivante :
Pour multiplier deux polynômes à coefficients entiers ayant chacun un degré similaire, utiliser la FFT est intéressant si le degré est supérieur à environ 1500. En-dessous, il est généralement plus rapide d'utiliser l'algorithme de multiplication naïf.

5.4 Pour aller plus loin

Pour aller plus loin que ce qui est demandé dans ce projet, il aurait été intéressant d'analyser la courbe obtenue pour fft et de trouver sa réelle nature. Comme ce qui a été fait pour la courbe de l'algorithme naïf. On aurait pu ainsi comparer la complexité théorique attendue et celle obtenue expérimentalement. Cependant, la fft ayant une courbe assez spéciale, il nous a été difficile de comprendre comment aborder cette démarche et de réussir à analyser une courbe pour laquelle on s'attend à avoir du $O(n \log(n) \log(\log(n))$.

En ce qui concerne les coefficients entiers utilisés, nous avons choisi de les représenter, ainsi que la taille du tableau les contenant, en tant qu'entiers signés sur 32 bits. Cela étant dit, si nous devons multiplier deux polynômes dont le résultat dépasse 32 bits [6], il sera nécessaire de représenter la taille du polynôme résultant avec un type **uint64_t**, c'est-à-dire un entier non signé sur 64 bits. De même, pour gérer des coefficients de plus grande taille, nous pouvons les représenter comme des entiers signés sur 64 bits (**int64_t**). Les types d'entiers nécessaires à cela sont disponibles dans la bibliothèque **stdint.h**. Il est important de noter que, dans le cas d'une représentation sur 64 bits, il ne faut pas oublier de modifier la fonction qui trouve la puissance de 2 la plus proche, car celle-ci manipule des bits pour y parvenir.

6. Segmentation Fault dans la section 5.1