Implantations efficaces de calculs sur les polynômes à une variable : FFT

a

7 Avril 2022

Table des matières

Introduction

Dans le cadre de l'UE LU2IN013, nous avons réalisé un projet sur l'optimisation de calculs sur les polynômes à une variable. Le but final de ce projet est la multiplication de deux polynômes sur $\mathbb{Z}/p\mathbb{Z}$ le plus efficacement possible.
Pour ce faire, nous nous intéressons à plusieurs type d'algorithmes pour la multiplication, en particulier : l'algorithme naïf, de Karatsuba et de multiplication par la FFT.
Nous avons tout d'abord commencé avec Python mais nous avions besoin d'un langage bas niveau pour plus de rapidité d'où notre choix du langage C.

1 Algorithme Naïf et de Karatsuba

1.1 Implémentations

Pour commencer, nous avons réalisé un algorithme simple (l'algorithme naïf) de multiplication de deux polynômes P et Q de degrés n. Cette algorithme consiste à développer le produit terme à terme, comme on le ferait à la main, c'est-à-dire qu'on écrit :

$$R(X) = PQ(X) = \sum_{i=0}^{n} \sum_{j=0}^{n} p_i q_j X^{i+j}$$

où $p_0, \ldots, p_n$ *et* $q_0, \ldots, q_n$ sont les coefficients respectifs de P et Q. Ici, à chaque tour de boucle, on rajoute aussi un modulo p sur les opérations de produits et de sommes pour que les coefficients restent dans $\mathbb{Z}/p\mathbb{Z}$.
De par sa simplicité, cette algorithme nous permet de vérifier les résultats de nos futurs algorithme plus performants mais plus complexes à implémenter.
Par la suite, grâce aux différentes sources trouvées, nous avons implémenté l'algorithme de Karatsuba. Nous avons appliqué l'algorithme se trouvant sur la source [2] tout en mettant les modulos au bon endroit.

1.2 Comparaison - Naïf/Karatsuba

L'algorithme naïf est en $O(n^2)$ et celui de Karatsuba en $O(n^{log_2(3)}) \approx O(n^{1.58})$. Voici les temps d'exécution (en seconde) de ces deux algorithme :

Degré	Naïf	Karatsuba
2^{14}	0.211790	0.047673
2^{15}	0.826513	0.136512
2^{16}	3.274061	0.393863
2^{17}	13.145910	1.276518
2^{18}	59.188884	3.325646

Nous voyons bien que l'algorithme de Karatsuba est bien plus performant que le naïf. Par ailleurs, nous n'avons pas vraiment cherché à optimiser l'algorithme de Karatsuba car il existe l'algorithme de multiplication par la FFT qui est encore plus rapide.

2 Fast Fourier Transform (FFT)

Comme dit dans la précédente partie, nous avons cherché à implémenter un troisième algorithme encore plus performant que celui de Karatsuba pour des polynômes de degrés supérieurs à quelques centaines (selon les implémentations). Le nom de cet algorithme est la FFT (Fast Fourier Transform). L'algorithme se base sur le théorème suivant :

Théorème 1. *Soient $a_1, \ldots, a_n$ et $b_1, \ldots, b_n$ des nombres réels (avec les a_i deux à deux distincts). Alors il existe un unique polynôme P de degré $< n$ tel que pour tout i dans $\{1, ..., n\}$, $P(a_i) = b_i$.*

En effet, l'idée est d'évaluer les polynômes P et Q de degrés n en des points spécifiques, qui sont des puissances successives d'une racine principale n-ième de l'unité ω du corps $\mathbb{Z}/p\mathbb{Z}$. Puis, il faut faire le produit de ces évaluations et retrouver l'unique polynôme $R = PQ$ à partir des valeurs du produit. Pour appliquer cet algorithme nous nous sommes servis de la source [1].

Algorithme de multiplication par FFT

Entrée. P et Q deux polynômes, n une puissance de 2 tel que $n > deg(PQ)$, et ω une racine principale n-ième de l'unité.
Sortie. $R = PQ$
Algorithme.

1. *Précalcul.* Calculer les puissances $\omega^0, \omega^1, \omega^2, \ldots, \omega^{n-1}$.
2. *Évaluation (DFT).* Calculer les valeurs :
   $Eval(P) = (P(\omega^0), \ldots, P(\omega^{n-1}))$; $Eval(Q) = (Q(\omega^0), \ldots, Q(\omega^{n-1}))$.
3. *Produit point à point.* $Eval(R) = (PQ(\omega^0), \ldots, PQ(\omega^{n-1}))$.
4. *Interpolation.* On retrouve R grâce à $Eval(R)$.

Nous pouvons voir que la complexité des étapes de précalcul et de produit point à point sont en $O(n)$. Notre problème est désormais de voir comment effectuer les étapes d'évaluation et d'interpolation rapidement.

2.1 Évalution d'un polynôme (DFT)

L'algorithme d'évaluation d'un polynôme est la deuxième étape de la multiplication de polynômes par FFT. Plus communément appelé DFT (Discrete Fourier Transform), il se base sur le principe de diviser pour régner. Soient $n = 2^k$ *et* $P(X) = p_nX^n + \ldots + p_1X + p_0$, on note P_0 et P_1 les polynômes composés des coefficients de rang respectivement pair et impair de P, c'est-à-dire :
$P_0(X) = p_nX^{n/2} + \ldots + p_2X + p_0$ *et* $P_1(X) = p_{n-1}X^{(n-2)/2} + \ldots + p_3X + p_1$.
On a alors que $P(X) = P_0(X^2) + XP_1(X^2)$.
La DFT se base sur cette propriété. En effet, on voit très vite qu'avec une telle décomposition, on peut faire un algorithme récursif où l'on divise le degré par deux à chaque occurrence, tout en mettant au carré les points où l'on veut évaluer le polynôme.

Algorithme DFT

Entrée. $P = p_0 + \ldots + p_{n-1}X^{n-1}$, n une puissance de 2 et le tableau $[1, \omega, \ldots, \omega^{n-1}]$ où ω est une racine n-ième principale de l'unité.
Sortie. $P(1), \ldots, P(\omega^{n-1})$.
Algorithme.

1. Si $n = 1$, renvoyer p_0.
2. Sinon, soit $k = n/2$. Calculer :

$$R_0(X) = \sum_{i=0}^{k-1}(p_i + p_{i+k})X^i$$

$$R_1(X) = \sum_{i=0}^{k-1}(p_i - p_{i+k})\omega^i X^i$$

3. Calculer récursivement $R_0(1), R_0(\omega^2), \ldots, R_0((\omega^2)^{k-1})$ et $R_1(1), R_1(\omega^2), \ldots, R_1((\omega^2)^{k-1})$.
4. Renvoyer $R_0(1), R_1(1), R_0(\omega^2), R_1(\omega^2), \ldots, R_0((\omega^2)^{k-1}), R_1((\omega^2)^{k-1})$.

Nous voyons que dans la deuxième étape de l'algorithme, nous devons faire une boucle allant de 0 à $\frac{n}{2} - 1$ pour les polynômes R_0 et R_1. Cette opération s'effectue en $O(n)$. De plus, le degré étant divisé par deux à chaque tour de récursion, on voit que la complexité de la DFT est en $O(n * log_2(n))$.

Par ailleurs, on sait montrer que pour l'étape d'interpolation, il suffit essentiellement de réutiliser l'algorithme de DFT mais en appelant la fonction avec les coefficients $Eval(R) = Eval(PQ)$, et, avec l'inverse des racines principales n-ième de l'unité, c'est-à-dire, $\omega^0, \omega^{-1}, \omega^{-2}, \ldots, \omega^{-(n-1)}$. On en déduit donc que la complexité de la FFT est $O(n * log_2(n))$.

2.1.1 Utilisation des unsigned int (Uint) et choix de p

Comme nous travaillons sur $\mathbb{Z}/p\mathbb{Z}$, les coefficients des polynômes sont tous positifs. Cela nous permet de travailler avec des *unsigned int* (*Uint*) et ainsi de travailler avec des entiers plus grands (jusqu'à $2^{32} - 1$ au lieu de $2^{31} - 1$ avec les *int*). Notre nombre premier valant initialement $p = 2013265921$ ($2^{30} < p < 2^{31}$), les *Uint* nous permettaient de pouvoir stocker la somme de deux entiers a et b entre 2^{30} et $p - 1$, alors que ce n'était possible avec les *int* car $a + b \geq 2 \times 2^{30} = 2^{31}$.

Malheureusement lors de l'implémentation de la vectorisation avec AVX2 dans notre code, nous avons vu qu'AVX2 ne permettait que de stocker des *int*, c'est pourquoi nous avons décidé de prendre un nombre premier plus petit, $p = 754974721$ qui est entre 2^{29} et 2^{30}.

De plus, le choix de p doit être judicieux pour pouvoir trouver des racines principales n-ième de l'unité. En effet, ici $p = 754974721 = 1 + 2^{24} * 3^2 * 5$, une racine primitive $p - 1$-ième de l'unité est $r = 11$, c'est-à-dire que $11^{p-1} \% p = 1$. On dit que $ordre = p - 1$ est l'ordre de la racine. Ainsi, pour trouver une racine principale n-ième de l'unité pour

la FFT, nous avons la formule $r\_principale = r^{ordre/n} \% p$. On remarque aussi dans cette formule qu'il faut que $ordre$ soit divisible par n, sachant que n est une puissance de 2, c'est pourquoi il faut qu'il y ait une grosse puissance de 2 dans la décomposition de p pour multiplier des polynômes de degré conséquent : ici, $ordre = p - 1 = 2^{24} * 3^2 * 5$. On comprend donc qu'avec ce nombre premier muni de la racine $r = 11$, on multiplie au plus des polynômes de degrés 2^{23} (2^{24} étant le degré maximum du polynôme résultat de la multiplication).

2.1.2 Opérations modulo p

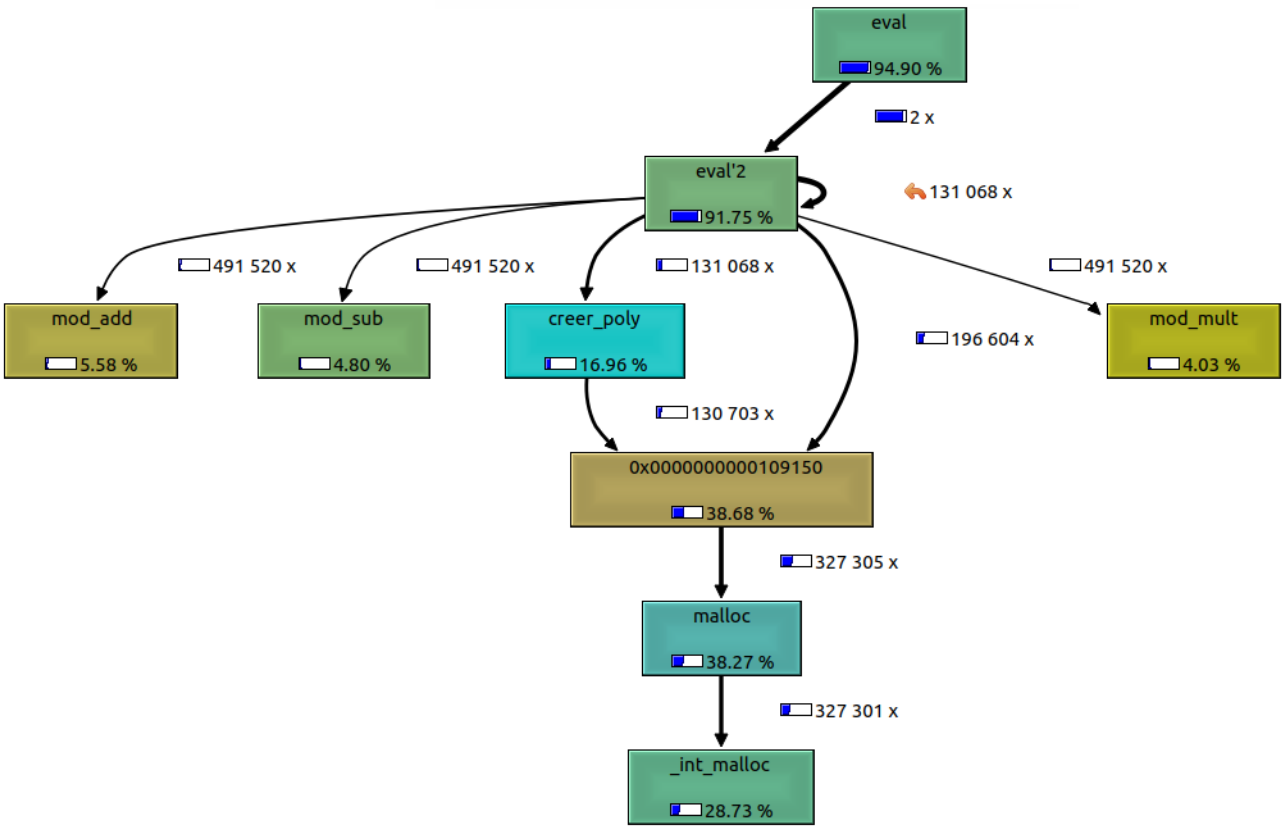
Nous avons décidé de faire des fonctions d'addition, de soustraction et de multiplication de deux nombres avec modulo. Comme l'instruction % prend environ 10 fois plus de temps que les instructions + et −, nous avons donc utilisé le fait que $(a + b) \% p = \begin{cases} a + b - p & si\ a + b \geq p \\ a + b & sinon \end{cases}$ pour faire l'addition plus rapidement.

De même pour la soustraction on utilise que : $(a - b) \% p = \begin{cases} p - (b - a) & si\ b > a \\ a - b & sinon \end{cases}$.

Pour la multiplication nous avons quand même utilisé l'opération % tout en castant la multiplication en *long* car la multiplication de deux coefficients peut facilement dépasser le nombre maximum des $Uint$. Il existe cependant des algorithmes plus efficaces pour faire le modulo dans la multiplication comme la réduction de Barrett qu'on va utiliser dans la troisième partie.

2.1.3 Optimisation de la DFT

Nous avons d'abord fait une première version fonctionnelle mais pas du tout optimisée de la DFT (voir la fonction *eval_malloc*()) avec beaucoup de malloc inutiles. Par exemple, nous avons malloc, à chaque tour, les tableaux des polynômes R_0 et R_1 de l'algorithme. Nous avons alors décidé de faire un profilage de cette fonction et voici le résultat :



On voit que les malloc représentent environ 40% du temps d'exécution de la fonction ! Pour optimiser cette fonction, notre priorité était alors de réduire le nombre de malloc. Nous avons d'abord remarqué qu'on pouvait enlever le malloc pour le tableau *racines_bis*, en effet, ce tableau sert à stocker les valeurs des cases $0, 2, \ldots, n$ du tableau *racines*, donc on a pu se passer du tableau *racines_bis* en passant en paramètre un pas *pas_rac* qu'on multiplie par deux à chaque tour.
Pour le malloc des tableaux de R_0 et R_1 de tailles $\frac{n}{2}$, nous avons pu les enlever en remarquant qu'on pouvait stocker les coefficients de R_0 dans la première moitié du tableau de P (de taille n) et ceux de R_1 dans la deuxième moitié. Ceci étant grâce au fait qu'on ai plus besoin de P aux prochains tours de récursion. Comme prévu la nouvelle fonction optimisée (fonction *eval*()) est au moins 2 fois plus rapide que la première version :

Degré (+1)	*eval_malloc*()	*eval*()
2^{20}	0.192210	0.079640
2^{21}	0.640100	0.211665
2^{22}	1.267813	0.677525
2^{23}	3.193026	1.468338
2^{24}	7.822262	3.202197

3 Implémentation de la vectorisation dans la FFT avec AVX2

Pour que notre FFT soit encore plus rapide, nous avons eu recours à la vectorisation avec AVX2. AVX2 nous permet de faire un certain nombre d'opérations (addition, soustraction...) beaucoup plus rapidement qu'avec les opérateurs normaux de C. En effet, AVX2 permet de créer un type de variable ($\_\_m256i$) pouvant stocker 8 entiers de 32 bits (à l'image d'un tableau), ce qui convient parfaitement à la structure des polynômes. AVX2 pouvant réaliser une opération élémentaire sur deux $\_\_m256i$ aussi vite qu'une opération élémentaire sur deux *int* en C, il est donc possible d'aller jusqu'à 8 fois plus vite avec AVX2. Voici un exemple de l'addition de deux $\_\_m256i$ en AVX2 :

```
__m256i a = _mm256_set_epi32(1, 2, 3, 4, 5, 6, 7, 8);
__m256i b = _mm256_set_epi32(1, 0, 1, 0, 1, 0, 1, 0);
__m256i x = _mm256_add_epi32(a, b);
// on a x == [2, 2, 4, 4, 6, 6, 8, 8]
```

Enfin, voici un lien qui recense les "intrinsics" pour AVX2 (les autres types SIMD/AVX/AVX512... sont aussi disponibles) : (METTRE LIEN INTEL)

3.1 Opération modulo p avec AVX2

Le problème avec AVX2 est qu'il n'y a pas d'opération modulo, de plus, comme on fait des opérations directement sur 8 entiers en même temps, il est difficile de faire le modulo comme dans la partie 2.1.2. Par exemple, pour l'addition supposons que :
$p = 7$, $\overrightarrow{a} = [6, 2, 0, 0, 0, 0, 0, 0]$, $\overrightarrow{b} = [4, 1, 0, 0, 0, 0, 0, 0]$ et $\overrightarrow{x} = \overrightarrow{a} + \overrightarrow{b}$.
On a notamment $\overrightarrow{x}[0] = 10$ et $\overrightarrow{x}[1] = 3$, on devrait donc retirer p pour faire le modulo sur $\overrightarrow{x}[0]$ mais alors on aurait $\overrightarrow{x}[1] = 3 - 7 = -4$, ce qui n'est pas le résultat attendu.

Pour faire l'addition et la soustraction, il faut savoir qu'AVX2 nous permet de prendre les minimums "positif" sur chaque cases de deux $\_\_m256i$ (tous les nombres négatifs vont être converti en $Uint$), par exemple :
$min\_pos([1, 0],\ [0, -10]) = min([1, 0],\ [0,$ UINT_MAX$-10]) = [0, 0]$ et non $[0, -10]$.

Pour l'addition, après avoir eu $\overrightarrow{x} = \overrightarrow{a} + \overrightarrow{b}$, nous allons faire l'opération $min\_pos(\overrightarrow{x},\ \overrightarrow{x} - \overrightarrow{p})$ où $\overrightarrow{p}$ est un $\_\_m256i$ ne contenant que des p. Ceci permet bien de faire le modulo car si $\overrightarrow{a}[i] + \overrightarrow{b}[i] \geq p$, alors $min\_pos()$ retourne $\overrightarrow{x}[i] - p$. Sinon on a $\overrightarrow{a}[i] + \overrightarrow{b}[i] < p$, donc $\overrightarrow{x}[i] - p < 0$, et après conversion en $Uint$, $\overrightarrow{x}[i] - p$ sera plus grand que $\overrightarrow{x}$[i] et donc l'opération retournera $\overrightarrow{x}$[i]. Ceci correspond bien aux cas de la partie 2.1.2.
C'est la même logique pour la soustraction $\overrightarrow{x} = \overrightarrow{a} - \overrightarrow{b}$, on prend $min\_pos(\overrightarrow{x},\ \overrightarrow{x} + \overrightarrow{p})$.

Pour la multiplication, nous avons utilisé l'algorithme de réduction de Barrett, nous nous sommes inspirés de la source [3] qui montre comment faire cet algorithme avec vectorisation sur des entiers 16 bits.

3.2 Comparaison de temps des opérations sans et avec AVX2

Dans cette partie, nous allons comparer les temps d'exécution pour les opérations d'addition, de soustraction et de multiplication modulo p sans AVX et avec AVX. Pour cela, nous avons fait les opérations sur deux tableaux de taille $AVOIR$.

Opération	Sans AVX2	Avec AVX2
Addition	XX	XX
Soustraction	XX	XX
Multiplication	XX	XX

3.3 Implémentation dans la DFT

L'implémentation de l'addition et soustraction vectorisée dans la DFT est plutôt directe, on fait une boucle dont l'indice augmente de 8 en 8 en passant les tableaux de taille 8 correspondants dans les fonctions de vectorisation.
De plus, on remarque que dans la DFT, on doit faire une addition et une soustraction avec les mêmes coefficients ($p_i + p_{i+k}$ et $p_i - p_{i+k}$). On remarque aussi que dans les fonctions $vect\_mod\_add()$ et $vect\_mod\_sub()$, nous devons charger les tableaux en $\_\_m256i$ à chaque fois (avec la fonction $\_mm256\_loadu\_si256()$). Pour optimiser, nous avons donc fait la fonction $vect\_mod\_add\_sub\_eval()$ qui permet de faire l'addition et la soustraction en ne chargeant qu'une fois les deux tableaux.

Désormais, pour la multiplication $(p_i - p_{i+k}) * \omega^i$, le principal problème que nous avons rencontré est le pas pas_rac pour le tableau $racines$. En effet, on voit que dans la fonction sans AVX2 : $eval()$, dans la première boucle for, nous devons accéder à la case $i * pas\_rac$. Donc en travaillant sur des tableaux de taille 8 avec AVX2, il faudrait que nous accédions aux cases $(i+0) * pas\_rac, (i+1) * pas\_rac, \ldots, (i+7) * pas\_rac$. Malheureusement notre fonction $\_mm256\_loadu\_si256()$ ne nous permet que de charger 8 cases à la suite (les cases $i{+}0, i{+}1, \ldots, i{+}7$). Cependant, nous avons trouvé la fonction $\_mm256\_i32gather\_epi32()$ qui prend en paramètre deux tableaux $\_\_m256i$: tab et $indices$ où $indices$ contient les indices des éléments qu'on veut avoir dans tab, la fonction prend aussi un entier $scale$ qui représente la taille en octet des éléments dans tab. Ce qui fait qu'ici, tab correspond au tableau $racines$, $indices$ au tableau $[(i{+}0){*}pas\_rac,\ (i{+}1){*}pas\_rac,\ \ldots,\ (i{+}7){*}pas\_rac]$ et $scale$ à 4 (taille d'un int).

Pour charger les cases voulues du tableau $racines$, il nous suffit alors de créer $indices$. Ce qui se fait aisément en chargeant les tableaux :
$\overrightarrow{i} = [i,\ \ldots,\ i],\ \overrightarrow{u} = [0, 1, 2, 3, 4, 5, 6, 7], \overrightarrow{pas} = [pas\_rac,\ \ldots,\ pas\_rac]$.
Il en vient alors que $indices = (\overrightarrow{i} + \overrightarrow{u}) * \overrightarrow{pas}$, ces opérations se faisant sans problème avec AVX2.

3.4 Tests de temps

Degré (+1)	Normal	AVX2
2^{16}	XX	XX
2^{17}	XX	XX
2^{18}	XX	XX
2^{19}	XX	XX
2^{20}	XX	XX
2^{21}	XX	XX
2^{22}	XX	XX
2^{23}	XX	XX
2^{24}	XX	XX

Références

[1] Bostan A., Chyzak F., Giusti M., Lebreton R., Lecerf G., Salvy B., and Schost E. Algorithmes efficaces en calcul formel. Technical report, Université Paris Diderot and Écoles Normales Supérieures de Cachan et de Paris and École Polytechnique, décembre 2018.

[2] Werner B. and Schost E. Informatique tronc commun td2 récursivité 2 : l'algorithme de karatsuba. Technical report, École Polytechnique, novembre 1999.

[3] Van Der Hoeven J., Lecerf G., and Quintin G. Modular simd arithmetic in mathemagix. Technical report, CNRS and École Polytechnique and Laboratoire LIX Universite de Limoges and Laboratoire XLIM, août 2016.