UNIVERSITÉ PIERRE ET MARIE CURIE

MASTER D'INFORMATIQUE SPÉCIALITÉ SFPN

RAPPORT DE STAGE DE FIN D'ÉTUDES

Implémentation logicielle de l'algorithme de composition modulaire de Kedlaya-Umans

---

Auteur :
Jocelyn RYCKEGHEM

Encadré par :
Jean-Pierre FLORI
Jérôme PLÛT

14 mars – 13 septembre 2016

Table des matières

Ce stage porte sur l'étude de l'algorithme de composition modulaire de Kedlaya et Umans [KU11] et son implémentation efficace en C. Le problème de composition modulaire consiste à calculer la composition de deux polynômes modulo un troisième polynôme. L'algorithme de Kedlaya et Umans est le premier à proposer une complexité quasi-linéaire en le degré des polynômes, pour une caractéristique quelconque. Pourtant, en pratique c'est l'algorithme de complexité sous-quadratique de Brent et Kung [BK78] qui est généralement implémenté. Aucune étude ne montre l'aspect quasi-linéaire de l'algorithme de Kedlaya et Umans. L'objectif principal de ce stage est donc d'essayer de faire apparaître cet aspect quasi-linéaire.

Présentation de l'organisme d'accueil

L'Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI) est un service rattaché au secrétariat général de la défense et de la sécurité nationale (SGDSN), autorité chargée d'assister le Premier ministre dans le domaine de la défense et de la sécurité nationale. Sa mission est de défendre les systèmes d'information français et de préserver la souveraineté nationale. Elle a aussi pour mission de porter conseil et soutien aux administrations et aux opérateurs d'importance vitale. Elle doit sensibiliser aussi bien les entreprises que les particuliers au niveau de la cybersécurité et des cybermenaces.
Créée par le décret n° 2009-834 du 7 juillet 2009, elle est aujourd'hui constituée d'un effectif de plus de 500 personnes, et a un budget de 80 millions d'euros.

Je travaille au sein de la sous-direction expertise, division scientifique et technique, au laboratoire de cryptographie. Je suis encadré par Jean-Pierre Flori et Jérôme Plût qui sont membres du laboratoire de cryptographie.

1 Introduction

Notations
Dans ce rapport, on pose R un anneau commutatif. On note $R[X]$ l'anneau des polynômes à coefficients dans R. $\mathbb{F}_p$ désigne le corps fini $\mathbb{Z}/p\mathbb{Z}$, et $\mathbb{F}_q$ désigne une extension de degré e de $\mathbb{F}_p$. La complexité sera principalement exprimée en nombre d'opérations dans l'anneau des coefficients. On pose $M(n)$ une borne supérieure de la complexité de la multiplication de deux polynômes de degré n, et on définit $\widetilde{O}(n)$ comme $O(n)$ multiplié par des facteurs logarithmiques en n. Avec l'algorithme de Schönhage et Strassen [VZGG99], $M(n) = O(n\log(n)\log(\log(n)))$. La table 1 rappelle les complexités d'algorithmes utiles pour la suite.

Algorithme	Entrée	Sortie	Complexité
Multiplication	$P(X), Q(X)$ de degré $\leq n$	$P(X) \times Q(X)$	$M(n) = \widetilde{O}(n)$
Réduction modulaire	$P(X), Q(X)$ de degré $\leq n$	$P(X) \mod Q(X)$	$O(M(n))$
Évaluation multipoint	$P(X)$ de degré $\leq n$, α_i pour $1 \leq i \leq n$	$P(\alpha_i)$ pour $1 \leq i \leq n$	$O(M(n)\log(n))$
Interpolation	α_i, β_i pour $0 \leq i \leq n$	$P(X)$ de degré $\leq n$	$O(M(n)\log(n))$
Produit matriciel	$A, B \in M_n(R)$	$A \times B$	$O(n^\omega), 2 \leq \omega \leq 3$

TABLE 1 – Complexité des algorithmes élémentaires

On peut maintenant définir formellement le problème de composition modulaire.

Problème 1 (*Problème de composition modulaire)*
Étant donnés $f(X)$, $g(X)$ et $h(X) \in R[X]$, chacun de degré au plus $n-1$, et h unitaire, le problème de composition modulaire consiste à calculer $f(g(X)) \mod h(X)$.

Dans $\mathbb{F}_q[X]$, l'algorithme de composition modulaire de Kedlaya-Umans a une complexité $C(n,q)$ égale à $\widetilde{O}(n\log(q))$ opérations élémentaires. Cette nouvelle complexité est quasi-linéaire en la taille de l'entrée, et a un impact sur les complexités d'algorithmes existants :

- La factorisation de polynôme univarié [KS98] : $\widetilde{O}(C(n,q)n^{0.5} + n\log^2(q))$ opérations élémentaires. Les algorithmes de factorisation sont habituellement quadratiques en le degré, cette nouvelle complexité permet de descendre à une complexité $\widetilde{O}(n^{1.5})$ pour q fixé. L'algorithme utilisé est probabiliste.
- Le test d'irréductibilité de polynôme univarié [Rab80] : $\widetilde{O}(n\log^2(q) + C(n,q)\log^2(n))$ opérations élémentaires [VZGG99]. On obtient un algorithme (déterministe) quasi-linéaire en le degré.

L'algorithme de factorisation est celui qui a l'impact le plus important en cryptographie : il permet notamment d'accélérer des déchiffrements (comme par exemple ceux des cryptosystèmes de type HFE (Hidden Field Equation)). Il a également un impact important en mathématiques, il sert par exemple à calculer des groupes de Galois sur $\mathbb{Q}$.

Application au principe de transposition

Si un problème peut se modéliser comme un produit matrice vecteur $A \times v = w$, alors on définit son problème transposé comme ${}^tA \times w = v$. Le principe de transposition [BLS03], ou principe de Tellegen, consiste à dire que si on connaît un algorithme de complexité linéaire pour résoudre un problème basé sur le calcul d'une application linéaire, alors on connaît un algorithme de même complexité asymptotique pour résoudre le problème transposé.

Par exemple, évaluer un polynôme P de degré n en une valeur a se modélise comme $\begin{pmatrix}1 & a & \dots & a^n\end{pmatrix} \times \begin{pmatrix}p_0 & p_1 & \dots & p_n\end{pmatrix}^t = x_0$. Son problème transposé est de multiplier le vecteur des puissances de a par un scalaire x_0, car il se modélise comme $\begin{pmatrix}1 & a & \dots & a^n\end{pmatrix}^t \times x_0 = \begin{pmatrix}p_0 & p_1 & \dots & p_n\end{pmatrix}^t$.

Le problème transposé au problème de composition modulaire est le problème de projection modulaire de puissances (*modular power projection*).

Problème 2 (*Problème de projection modulaire de puissances)*
Étant donnés g et h dans $R[X]$ de degré au plus $n-1$, et α un élément de R, le problème de projection modulaire de puissances consiste à calculer $(g(X)^i \mod h(X))(\alpha)$ pour i allant de 0 à $n-1$.

L'algorithme de composition modulaire de Kedlaya et Umans ne se base pas que sur le calcul d'une application linéaire. De ce fait, le principe de transposition ne permet pas d'obtenir directement un algorithme quasi-linéaire pour la projection modulaire des puissances : on ne

peut pas appliquer le principe de transposition sur les autres parties de l'algorithme. Cependant, il est quand même possible de trouver des algorithmes quasi-linéaires pour ces parties [KU11], permettant d'avoir une complexité finale $P(n,q) = \widetilde{O}(n\log(q))$.

Une des applications de l'algorithme de projection modulaire de puissances est le calcul du polynôme minimal d'un élément de $\mathbb{F}_q[X]$ [Sho99], qui a pour complexité $\widetilde{O}(n\log(q) + C(n,q) + P(n,q))$ opérations élémentaires. L'algorithme devient donc quasi-linéaire en le degré.

2 Rétrospection

Tout d'abord, il faut remarquer une différence fondamentale entre des algorithmes de type multiplication modulaire avec ceux de type composition modulaire. Contrairement à la multiplication modulaire, séparer la composition de la réduction modulaire, c'est-à-dire calculer $f(g(X))$, puis le réduire modulo $h(X)$, ne permet pas d'atteindre la meilleure complexité possible. En effet, le polynôme $f(g(X))$ est de degré n^2 dans le pire des cas, la complexité ne peut donc être meilleure que $O(n^2)$, alors que l'on peut espérer atteindre $O(n)$ puisque l'on a une entrée de taille $O(n)$. Il faut donc un algorithme prenant en compte la réduction modulaire lors de la composition.

2.1 Méthode naïve

Un algorithme simple consiste à calculer $g(X)^i$ pour i allant de 2 à $n-1$, en décomposant $g(X)^i$ comme $g(X)^{i-1} \times g(X)$, puis de remplacer les puissances de X de f par ces valeurs.

Le calcul des $g(X)^i$ se fait avec $n-2$ multiplications polynomiales modulaires, ce qui donne une complexité égale à $O(nM(n)) = \widetilde{O}(n^2)$ opérations dans R.
La substitution demande $n-1$ additions et multiplications entre polynôme et coefficient, ce qui se fait avec une complexité égale à $O(n^2)$ opérations dans R.
On obtient donc un algorithme quasi-quadratique.

On peut également utiliser la méthode de Horner [Ost54], qui permet d'éviter le calcul des puissances de $g(X)$. On économise $n-2$ multiplications entre polynôme et coefficient, mais la classe de complexité ne change pas.

2.2 Méthode de Brent & Kung

Publié en 1978, l'algorithme de Brent et Kung [BK78] est le premier algorithme de composition modulaire de polynômes à proposer une complexité sous-quadratique. Il repose sur une réduction du problème de composition modulaire au problème du produit matriciel. Cette réduction se fait à l'aide de la méthode « *Baby Step-Giant Step* », et permet d'obtenir une complexité bornée par $O(n^{\frac{\omega+1}{2}})$, où ω est l'exposant dans la complexité de la multiplication matricielle.

L'idée est d'appliquer une règle de Horner, où au lieu de multiplier par X chaque coefficient individuel, on multiplie par X^s chaque bloc de s coefficients.

Par exemple, soit $f = 9 + X + 2X^2 + 3X^3 + 4X^4 + 5X^5 + 6X^6 + 7X^7 + 8X^8$, la règle de Horner classique consisterait à écrire :

$$f = (((((((8 \times X + 7) \times X + 6) \times X + 5) \times X + 4) \times X + 3) \times X + 2) \times X + 1) \times X + 0.$$

Si on l'applique avec un pas de $X^{\lfloor\sqrt{8}\rfloor} = X^2$, on obtient :

$$((([8] \times X^2 + [7X + 6]) \times X^2 + [5X + 4]) \times X^2 + [3X + 2]) \times X^2 + [X + 9].$$

L'algorithme 1 préférera séparer la constante du reste du polynôme, ce qui donne :

$$(([8X^2 + 7X] \times X^2 + [6X^2 + 5X]) \times X^2 + [4X^2 + 3X]) \times X^2 + [2X^2 + X] + 9.$$

Algorithme 1 Composition modulaire de Brent et Kung

Entrée : $f(X)$, $g(X)$ et $h(X)$ trois polynômes univariés de degré au plus n.
Sortie : $f(g(X)) \mod h(X)$.

0. Calculer $s := \lfloor\sqrt{n}\rfloor$ et $t := \lceil\frac{n}{s}\rceil$.

1. Calculer $g_i := g(X)^i \mod h(X)$ pour i allant de 1 à s. On calcule chaque g_i comme étant égal à $g(X) \times g_{i-1}$. Ce sont les pas de bébé.
   Représenter ensuite ces données comme une matrice G de taille $s \times n$ où la ligne i correspond aux coefficients de g_i.

2. Représenter les coefficients de f comme une matrice F de taille $t \times s$ où la ligne i correspond aux coefficients de $X^{1+j\times s}$ à $X^{s+j\times s}$, j allant de 0 à $s-1$. Le coefficient constant est laissé de côté pour l'instant.

3. Calculer le produit matriciel $F \times G$. On pose p_k comme étant le polynôme formé par les coefficients de la $k^{ème}$ ligne du résultat.

4. Initialiser la variable res à 0, et pour k descendant de t à 1, calculer $res := (res \times g_s \mod h(X)) + p_k$. Ce sont les pas de géant. Ajouter le coefficient constant de f au résultat final pour obtenir $f(g(X)) \mod h(X)$.

Analyse de complexité

L'étape 1 demande s multiplications modulaires de polynôme.

L'étape 2 est négligeable dans cet algorithme.

L'étape 3 demande $O(n^{\frac{\omega+1}{2}})$ opérations dans R. Cette borne vient de la considération qu'un produit matriciel de $t \times s$ par $s \times n$ peut être vu comme n/t produits de matrices de $t \times s$ par $s \times t$, ce qui est dans notre cas quasiment un produit de matrices carrées. Une complexité plus précise serait de considérer directement la complexité de ce produit matriciel. En posant ω_2 l'exposant dans la complexité d'un produit de matrice $n \times n$ par $n \times n^2$, on a donc un algorithme de complexité bornée par $O(n^{\frac{\omega_2}{2}})$, avec $\omega_2 \leq \omega + 1$.

L'étape 4 demande t multiplications modulaires de polynôme et additions.

On a au final $O(\sqrt{n}M(n)) = \widetilde{O}(n^{1.5})$ opérations dans R pour les multiplications modulaires. Pour le produit matriciel, en pratique $\omega \leq 2.3727$ [Wil12] (avec une constante énorme dans le $O(n^\omega)$), et $\omega_2 \leq 3.334$ [HP98]. L'étape 3 domine donc l'exécution de l'algorithme de Brent et Kung, ce qui implique une complexité finale bornée par $\widetilde{O}(n^{1.67})$.

Un produit de matrices carrées a la complexité de $2n^2$ entrées, sa complexité est donc minorée par $\Omega(n^2)$, ce qui implique $w \geq 2$. On en déduit que l'algorithme de Brent et Kung est minoré par $\Omega(n^{1.5})$. Or la composition modulaire a $O(n)$ entrées, un algorithme quasi-optimal aurait donc une complexité quasi-linéaire en n.

Exemple 1.
Soient $f(X) = 9 + X + 2X^2 + 3X^3 + 4X^4 + 5X^5 + 6X^6 + 7X^7$, $g(X) = X$, $h(X) = X^8 - X$.
Cet exemple a la particularité que $f(g(X)) \mod h(X) = f(g(X)) = f(X)$.
On a ici $n = 8, s = 2$ et $t = 4$.
On calcule $g_1 = X, g_2 = g_1 \times g = X^2$.

$$\text{On a alors } G = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \text{ et } F = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 0 \end{pmatrix}, \text{ donc}$$

$$F \times G = \begin{pmatrix} 0 & 1 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 5 & 6 & 0 & 0 & 0 & 0 & 0 \\ 0 & 7 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} X + 2X^2 \\ 3X + 4X^2 \\ 5X + 6X^2 \\ 7X \end{pmatrix}.$$

Le calcul du résultat est effectué de la manière suivante :
$(([7X] \times X^2 + [6X^2 + 5X]) \times X^2 + [4X^2 + 3X]) \times X^2 + [2X^2 + X] + 9$.
On vérifie que le résultat est bien égal au polynôme f de départ.

3 Algorithme de Kedlaya et Umans

3.1 Composition modulaire rapide

Pour résoudre le problème de composition modulaire (algorithme 2), l'idée clé est l'utilisation d'une réduction au problème d'évaluation multipoint multivariée ci-dessous au moyen de transformations entre polynômes univariés et multivariés définies en 3.1.2.

Problème 3 (*Problème de l'Évaluation Multipoint Multivariée (EMM)*)
Étant donnés $f(X_0, ..., X_{m-1}) \in R[X_0, ..., X_{m-1}]$ de degré au plus $d-1$ en chaque variable, et $\alpha_0, ..., \alpha_{N-1} \in R^m$ des points d'évaluations, le problème consiste à calculer $f(\alpha_i)$ pour i allant de 0 à $N-1$.

3.1.1 Substitution de Kronecker

Définition 1. La substitution de Kronecker est l'application de paramètre $M \in \mathbb{N}$, qui à $P(X) \in \mathbb{Z}[X]$, associe l'entier $P(M)$. Elle permet de représenter un polynôme comme un entier en base M.

Exemple 2. Soit $P_1(X) = 3X^2 + 2X + 1$, on a $P_1(10) = 321$.
Soit $P_2(X) = 32X + 1$, on constate que l'application n'est pas injective : $P_2(10) = 321 = P_1(10)$.

Proposition 1. La substitution de Kronecker est injective si et seulement si M est strictement supérieur au plus grand coefficient de P. L'application réciproque est alors donnée par la lecture des chiffres de la représentation en base M de l'évaluation.

Exemple 3. On peut écrire 321 comme $3 \times 10^2 + 2 \times 10^1 + 1 \times 10^0$, on en déduit le polynôme $3X^2 + 2X + 1 = P_1(X)$.

Cette application est principalement connue de par son utilisation dans l'algorithme de multiplication polynomiale par substitution de Kronecker [Har09]. L'idée est de transformer les polynômes en entier en base $M = 2^k$ afin de les multiplier dans $\mathbb{Z}$, puis de lire directement les bits des coefficients du polynôme résultat sur le produit obtenu précédemment.

3.1.2 Substitution de Kronecker inverse

On peut définir l'application de substitution de Kronecker inverse. Plutôt que de passer de polynôme univarié à entier, l'application inverse permet de passer de polynôme univarié à multivarié.

Soient a et d deux entiers, on définit $a = \sum_{j=0}^{m-1} a_j d^j$ avec $0 \leq a_j < d$, ce qui correspond à la décomposition de a en base d.

Définition 2. L'application de substitution de Kronecker inverse est l'application $\psi_{d,m}$ qui permet de passer d'un polynôme en une variable de degré $n-1$, à un polynôme en m variables de degré $d-1$ par variable, avec $d \geq 2$ et $m \geq 2$. Elle est définie ci-dessous :

$$\begin{array}{rccc} \psi_{d,m} & : & R[X] & \to & R[X_0, ..., X_{m-1}] \\ & & X^a & \mapsto & X_0^{a_0} X_1^{a_1} ... X_{m-1}^{a_{m-1}} \end{array}$$

Cela revient à chercher $f \in R[X_0, ..., X_{m-1}]$ tel que $f(X, X^d, X^{d^2}, ..., X^{d^{m-1}}) = f^*(X)$.

L'idée de cette transformation est de diminuer le degré, pour en contrepartie augmenter le nombre de variables. Il existe une définition plus générale [KU11, définition 2.3] permettant de

partir d'un polynôme multivarié plutôt qu'univarié, et d'augmenter son nombre de variable tout en diminuant le degré en chaque variable.

Exemple 4. Soit $f(X) = X + 2X^2 + 3X^3 + 4X^4 + 8X^8$, on a $n = 9$. On choisit $d = 3$, ainsi $m = 2$ est optimal. Ici $X_0 = X$ et $X_1 = X^3$. On obtient alors $\psi_{3,2}(f) = X_0 + 2X_0^2 + 3X_1 + 4X_0X_1 + 8X_0^2X_1^2$.

Proposition 2. Si $n \leq d^m$, alors la substitution de Kronecker inverse $\psi_{d,m}$ est inversible. L'application réciproque $\psi_{d,m}^{-1}$ s'obtient en calculant $f^*(X) = f(X, X^d, X^{d^2}, ..., X^{d^{m-1}})$.

Exemple 5. Soient $g(X_0, X_1) = X_0 + 2X_0^2 + 3X_1 + 4X_0X_1 + 8X_0^2X_1^2$, $d = 3$ et $m = 2$. On calcule $g(X, X^d) = X + 2X^2 + 3X^3 + 4X^4 + 8X^8$.

Remarque. Une fois d choisi, on pose $m = \lceil \log_d(n) \rceil$. Cela permet, pour un d fixé, de choisir le m tel que d^m est le plus proche possible de n. Prendre un m plus grand reviendra à avoir des variables inutiles.

3.1.3 Algorithme de réduction de la composition modulaire à l'EMM

Algorithme 2 Composition modulaire rapide

Entrée : $f(X)$, $g(X)$ et $h(X)$ les polynômes définis dans le problème de composition modulaire (cf. problème 1), $d \geq 2$ un paramètre entier.
Sortie : $f(g(X)) \mod h(X)$.

1. Calculer $m := \lceil \log_d n \rceil$. Calculer ensuite $f^{\#} := \psi_{d,m}(f)$.

2. Calculer $g_i := g(X)^{d^i} \mod h(X)$ pour i allant de 0 à $m - 1$, en utilisant des mises à la puissance d successives.

3. Soit $N := d^m md$, choisir N éléments de R que l'on appelera $\beta_0, ..., \beta_{N-1}$, dont les différences deux à deux sont inversibles.
   Évaluer tous les g_i en les β_j, en utilisant m fois l'algorithme d'évaluation multipoint univariée rapide.

4. Résoudre le problème d'Évaluation Multipoint Multivariée de $f^{\#}$ en les α_j.
   Pour avoir une complexité intéressante, il faudra utiliser l'algorithme d'évaluation multipoint multivariée rapide qui sera décrit par la suite (cf. section 3.2).

5. Interpoler les N points obtenus à l'étape 4 pour obtenir $f^{\#}(g_0(X) \mod h(X), ..., g_{m-1}(X) \mod h(X))$. Il faut utiliser l'algorithme d'interpolation (univariée) rapide [Pot14].

6. Réduire ce polynôme par $h(X)$ pour obtenir $f(g(X)) \mod h(x)$.

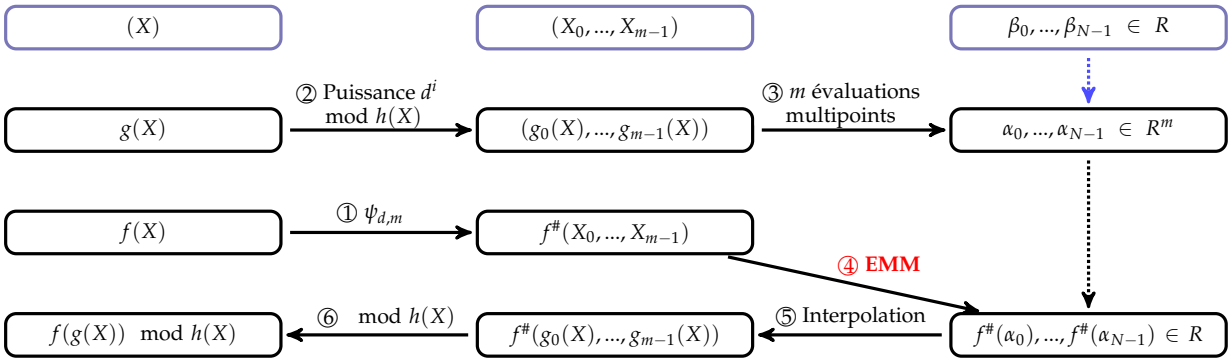

FIGURE 1 – Algorithme de composition modulaire rapide

Choix de N
Pour déterminer la valeur de N à choisir pour permettre l'interpolation, il faut connaître une majoration du degré du polynôme $f^\#(g_0(X) \mod h(X), ..., g_{m-1}(X) \mod h(X))$. Le degré de $f^\#$ est le degré du monôme de plus haut degré. Ce terme est celui possédant m variables avec chaque variable à la puissance $d-1$. Chaque variable va être substituée par un polynôme de degré au plus $n-1$, ce qui crée un polynôme de degré $(n-1)(d-1)m$. Il faut donc choisir $N \geq (n-1)(d-1)m+1$, que les auteurs de [KU11] ont décidé de majorer par $d^m md$.

3.1.4 Analyse de complexité

Théorème 1
L'algorithme 2 résout le problème de composition modulaire et sa complexité est bornée par $\widetilde{O}(T(d,m,N) + n^{1+\frac{2}{m}} m^2)$, $T(d,m,N)$ étant la complexité de l'algorithme d'EMM.

Démonstration. On analyse la complexité des différentes étapes de l'algorithme :

1. Le calcul de $f^\#$ a une complexité égale à $O(d^m \log(|R|))$ puisque la nouvelle structure est un polynôme à m variables, de degré au plus $d-1$ en chaque variable. Ce polynôme possède donc d^m termes à coefficients dans R.
2. Cette étape calcule m mises à la puissance d modulaires. Chaque mise à la puissance d modulaire demande $\log(d)$ phases de « *square and multiply* » modulaires. La réduction modulo $h(x)$ permettant de toujours avoir des polynômes de degré au plus $n-1$, le coût de la phase « *square and multiply* » a une complexité égale à $O(M(n))$. La complexité de cette étape vaut donc $O(m \log(d) M(n))$.
3. Le choix de N points peut se faire avec une complexité égale à $O(N \log(N))$. On a ensuite m évaluations multipoints univariées de polynômes de degré au plus $n-1$. Sachant que $n < N$, évaluer un polynôme en N points se fait avec une complexité égale à $O(M(N) \log(N))$ (cf. table 1). Cette étape a donc une complexité égale à $O(mM(N) \log(N))$.
4. On pose $T(d,m,N)$ la complexité de l'évaluation multipoint multivariée que nécessite cette étape. Cette complexité sera étudiée dans la suite de ce rapport.
5. L'interpolation en N points se fait avec une complexité $O(M(N) \log(N))$ (cf. table 1).

6. Réduire un polynôme de degré au plus N se fait avec une complexité égale à $O(M(N))$ (cf. table 1).

Pour la complexité finale, l'étape 4 a pour complexité $T(d, m, N)$, et l'étape 3 domine sur toutes les autres étapes. En remarquant que $d^m \leq nd$, et donc que $N \leq nmd^2$, on peut réécrire la complexité de l'étape 3 comme $\widetilde{O}(nm^2d^2)$. Cette majoration de d^m a permi de réécrire la complexité en fonction de n, faisant apparaître un aspect quasi-linéaire en n. On obtient au final une complexité égale à $\widetilde{O}(T(d, m, N) + nm^2d^2) = \widetilde{O}(T(d, m, N) + n^{1+\frac{2}{m}}m^2)$.

□

Pour que l'algorithme de Kedlaya et Umans soit quasi-linéaire, il faut maintenant un algorithme d'EMM au plus quasi-linéaire en d^m et N. Il faut noter que dans l'étude de complexité du problème d'EMM, la relation $N = d^m md$ n'existe pas, ce qui explique que les paramètres soient traités de manière indépendante.

Remarque. Les problèmes de composition modulaire et d'évaluation multipoint multivariée sont équivalents. La preuve vient de l'algorithme de réduction de l'EMM à la composition modulaire [Uma08].

3.2 Évaluation multipoint multivariée rapide

3.2.1 En petite caractéristique

On s'intéresse ici à l'EMM de paramètres d, m, N dans un corps $\mathbb{F}_q$ de caractéristique p petite. La méthode employée ici [Uma08] utilise la sustitution de Kronecker inverse (cf. 3.1.2) pour se ramener à l'évaluation multipoint univariée. Pour cela, soient c un entier et $h = p^c$, il faudra assurer de pouvoir trouver un élément $\eta \in R$ qui respecte deux propriétés :

(i) η est d'ordre $h - 1$.

(ii) $\eta^i - \eta^j$ est inversible pour $i \not\equiv j \mod c$.

Si R est un corps, la seconde condition est nécessairement assurée.

Deux cas sont possibles :

— Soit un élément d'ordre $h - 1$ existe dans $\mathbb{F}_q$, et dans ce cas il suffit de l'utiliser ;

— Soit un tel élément n'existe pas, et il faut alors construire une extension de $\mathbb{F}_q$ pour générer ce nouvel élément. L'algorithme 3 permet de trouver un élément d'ordre $h - 1$.

Algorithme 3 Recherche d'un élément d'ordre $h - 1$ dans $\mathbb{F}_q^\times$

1. Trouver un polynôme irréductible $P(W)$ de degré c dans $\mathbb{F}_p$.
2. Trouver une racine primitive η de $\mathbb{F}_p[W]/P(W)$.
3. η est d'ordre $h - 1$ dans $R = \mathbb{F}_q[W]/P(W)$.

Pour réduire le problème à une évaluation multipoint univariée, il faut d'abord définir quelques applications utiles :

- l'endomorphisme de Frobenius σ qui à x associe x^h.

- l'application d'interpolation ϕ qui à $\alpha \in \mathbb{F}_q^m$ associe le polynôme g_α tel que : $g_\alpha(\eta^i) = \sigma^{-i}(\alpha_i)$ pour i allant de 0 à $m-1$.

- l'application π qui à $g(Z) \in R[Z]/(Z^{h-1} - \eta)$ associe $g(1)$.

Ces applications vérifient le lemme suivant [KU11, lemme 6.2].

Lemme 1. Pour tout $\alpha \in \mathbb{F}_q^m$, $\pi(f^*(\phi(\alpha))) = f(\alpha)$.

À partir de cette relation, on peut obtenir un algorithme pour l'EMM.

Algorithme 4 Évaluation multipoint multivariée rapide en petite caractéristique

---

Entrée : $f \in \mathbb{F}_q[X_0, ..., X_{m-1}]$ de degré au plus $d-1$ pour chaque variable et $\alpha_0, ..., \alpha_{N-1} \in \mathbb{F}_q^m$.
Sortie : $f(\alpha_i)$ pour i allant de 0 à $N-1$.

1. Calculer $h = p^c$ avec c le plus petit entier tel que $h \geq m^2 d$.
2. Calculer η un élément d'ordre $h-1$ dans R, avec $R = \mathbb{F}_q$ ou $\mathbb{F}_q[W]/P(W)$ (cf. algo. 3).
3. Définir le polynôme $E(Z) = Z^{h-1} - \eta$ et l'extension $S = R[Z]/E(Z)$.
4. Calculer $\phi(\alpha_i) \in R[Z]$ pour i allant de 0 à $N-1$.
5. Calculer $f^* = \psi_{h,m}^{-1}(f) \in S$ la réciproque de la substitution de Kronecker inverse (cf. 3.1.2) appliquée à f.
6. Appliquer l'évaluation multipoint de f^* en les $\phi(\alpha_i)$ et pour chaque évalué, renvoyer son image par π.

Théorème 2
L'algorithme 4 calcule correctement une EMM et sa complexité est bornée par $\widetilde{O}((N + d^m)(m^2 p)^m)$ [KU11, théorème 6.3].

Corollaire 1
Soient N et d assez grands pour avoir $m \leq d^{o(1)}$ et $p \leq d^{o(1)}$. La complexité de l'algorithme 4 est bornée par $\widetilde{O}(N + d^m)$ [KU11, corollaire 6.4].

Cet algorithme est quasi-linéaire en la taille de l'entrée mais a une forte dépendance en p. De ce fait, il faut un algorithme avec une dépendance logarithmique en p pour qu'il soit utilisable pour toute caractéristique.

3.2.2 Dans $\mathbb{F}_p$ avec p premier : évaluation par la FFT multidimensionnelle

L'algorithme d'EMM fonctionnant pour tout caractéristique (cf. algorithme 6) nécessite l'algorithme d'évaluation par transformée de Fourier rapide (FFT) multidimensionnelle. Pour

l'étudier, il faut d'abord définir la transformée de Fourier unidimensionnelle, et montrer en quoi elle permet d'évaluer un polynôme.

Problème 4 (*Transformée de Fourier [HVDHL14]*)
Étant donné une liste de points $(f_0, ..., f_{n-1}) \in R^n$ et ω une racine primitive n-ème de l'unité, la transformée de Fourier consiste à calculer $x_j = \sum_{k=0}^{n-1} f_k \times (\omega^j)^k$ pour j allant de 0 à $n-1$.

En posant $\omega = X$, on reconnait le problème d'évaluation multipoint d'un polynôme en les puissances de ω. De plus, la propriété de racine primitive implique que les puissances ω^i parcourent tous les éléments de $\mathbb{F}_p^*$. L'évaluation d'un polynôme en toutes les puissances de ω permet donc de l'évaluer en tout point de $\mathbb{F}_p^*$. La FFT s'effectue avec une complexité bornée par $\widetilde{O}(p)$. L'algorithme de FFT multidimensionnelle utilise un algorithme de FFT unidimensionnelle. Il existe plusieurs possibilités (cf. section 3.3) pour ce dernier algorithme. Nous en discutons le choix en section 4.7.3.

L'idée principale de l'algorithme d'évaluation est d'évaluer le polynôme multivarié avec une FFT multidimensionnelle, qui elle-même appellera des FFT unidimensionnelles. La FFT unidimensionnelle demandant d'avoir un polynôme de degré au plus $p-2$, il faut d'abord réduire le polynôme multivarié modulo $X_i^{p-1} - 1$. Zéro n'est pas racine de ce polynôme, on préfère donc réduire modulo $X_i^p - X_i$, puis évaluer en zéro et réduire modulo $X_i^{p-1} - 1$ au moment d'appliquer la FFT unidimensionnelle.

Algorithme 5 Évaluation par la FFT multidimensionnelle

Entrée : $f \in \mathbb{F}_p[X_0, ..., X_{m-1}]$ de degré au plus $d-1$ pour chaque variable et $\alpha_0, ..., \alpha_{N-1} \in \mathbb{F}_p^m$.
Sortie : $f(\alpha_i)$ pour i allant de 0 à $N-1$.

1. Calculer $\bar{f} := f \mod (X_j^p - X_j)$ pour j allant de 0 à $m-1$.
   Les réductions modulaires permettent d'ajouter l'information que le polynôme sera évalué dans $\mathbb{F}_p$.

2. Appliquer la FFT multidimensionnelle sur $\bar{f}$:
   - ☐ si $m = 1$, le polynôme est univarié, on peut donc appliquer la FFT unidimensionnelle (cf. section 3.3) pour l'évaluer pour tout $\beta \in \mathbb{F}_p$. Pour cela, on évalue $\bar{f}$ en 0 puis on le réduit modulo $X_0^{p-1} - 1$ avant d'effectuer la FFT unidimensionnelle.
   - ☐ si $m > 1$, on écrit $\bar{f} = \sum_{k=0}^{p-1} X_{m-1}^k \times f_k(X_0, ..., X_{m-2})$ et on applique une récursion (de l'étape 2) sur les f_k qui ont $m-1$ variables. Soit $f_k(\beta)$ le résultat de l'évaluation des polynômes f_k pour tout $\beta \in \mathbb{F}_p^{m-1}$, on évalue $\sum_{k=0}^{p-1} X_{m-1}^k f_k(\beta)$ en 0 puis on la réduit modulo $X_{m-1}^{p-1} - 1$ avant d'appliquer la FFT unidimensionnelle.

   Le résultat obtenu est $\bar{f}(\beta)$ pour tout $\beta \in \mathbb{F}_p^m$. Cela forme une table de p^m éléments.

3. Lire dans la table et renvoyer $\bar{f}(\alpha_i) = f(\alpha_i)$ pour i allant de 0 à $N-1$.

Analyse de complexité

Théorème 3
L'algorithme 5 calcule correctement une EMM et sa complexité est égale à $O(m(d^m + p^m + N)\log(p))$.

Démonstration. On analyse la complexité des différentes étapes de l'algorithme :

1. L'étape 1 demande m réductions modulaires pour un polynôme à d^m termes : sa complexité est bornée par $O(md^m \log(p))$.

2. L'étape 2 appliquée à un polynôme à m variables demande p récursions. Ces p récursions renvoient p^{m-1} évaluations, ce qui fait ensuite p^{m-1} polynômes univariés à évaluer en tout point de $\mathbb{F}_p$ avec la FFT unidimensionnelle. Le nombre de FFT est donc $p^{m-1} + p(p^{m-2} + p(p^{m-3} + ... + p(p^1 + p \times 1)...))$, le nombre 1 venant du cas de base où l'on applique une seule FFT. On remarque aisément que cette somme vaut mp^{m-1}, on peut par exemple reconnaître l'écriture par la méthode de Horner du polynôme $\sum_{i=0}^{m-1} p^{m-1-i}X^i$ évalué en p. Avec une FFT de complexité égale à $O(p\log(p))$, on arrive à une complexité égale à $O(mp^m \log(p))$.

3. L'étape finale demande de lire N coefficients de $\mathbb{F}_p^m$ dans une table de taille p^m. La complexité vaut $O(mN\log(p))$.

On a donc une complexité finale égale à $O(m(d^m + p^m + N)\log(p))$.

□

La complexité en mémoire vaut $O(p^m)$. Par conséquent, cet algorithme est impraticable pour des nombres premiers p trop grands. Il faut donc utiliser au préalable un algorithme permettant de réduire la taille du nombre premier utilisé avant de s'en servir. C'est ce que fait l'algorithme Multimodulaire dans la section suivante.

3.2.3 Dans $\mathbb{Z}/r\mathbb{Z}$, r non nécessairement premier

Cette partie a pour objectif de réduire la dépendance en p avant d'appeler l'algorithme précédent 5.

Pour ce faire, l'idée principale est de relever le problème dans $\mathbb{Z}$, et de le résoudre modulo de petits nombres premiers auxiliaires ℓ afin de pouvoir y utiliser l'algorithme 5 qui est polynomial en mémoire en fonction du module (ici ℓ). Une fois les problèmes résolus, on combine les solutions avec le théorème des restes chinois (CRT) pour obtenir un résultat dans $\mathbb{Z}$, et réduire modulo r permet de revenir dans $\mathbb{Z}/r\mathbb{Z}$.

Il faut remarquer qu'évaluer un monôme de f dans $(\mathbb{Z}/r\mathbb{Z})^m$ produit un résultat valant au plus $(r-1)^{m(d-1)+1}$, et f ayant d^m coefficients, l'évaluer produit une valeur maximale de $d^m(r-1)^{m(d-1)+1} \leq d^m(r-1)^{md}$. Choisir le produit des nombres premiers supérieur à $d^m(r-1)^{md}$ permet d'assurer la validité du calcul dans $\mathbb{Z}$ lors de la reconstruction par le CRT.

On remarque tout de suite que le procédé peut être appliqué récursivement, ce qui permet de réduire la taille des nombres premiers auxiliaires. Cependant, il existe un nombre premier maximal L_{max} tel que, pour $\ell \leq L_{max}$, le découpage modulo de petits nombres premiers n'est plus possible : c'est le plus grand nombre premier avec lequel on est obligé d'appliquer directement la FFT multidimensionnelle telle que décrite dans l'algorithme 5. C'est le point où la descente des nombres premiers ℓ se termine.

L'algorithme 7 calcule L_{max}. Cet algorithme termine car la valeur de la variable `Lmax` est minorée par 2, et car elle décroit strictement (l'algorithme termine quand ce n'est plus le cas). Cette valeur est utile pour déterminer quand arrêter la récursion.

Algorithme 6 Multimodulaire

Entrée : $f \in (\mathbb{Z}/r\mathbb{Z})[X_0, ..., X_{m-1}]$ de degré au plus $d-1$ en chaque variable, $\alpha_0, ..., \alpha_{N-1} \in (\mathbb{Z}/r\mathbb{Z})^m$, L_{max} et K un entier (valant 1 pour le moment).
Sortie : $f(\alpha_i)$ pour i allant de 0 à $N-1$.

1. Calculer $\prod_{i=1}^{k} \ell_i$ le produit des nombres premiers successifs, où k est le plus grand entier tel que le produit soit inférieur ou égal à $d^m(r-1)^{md}$.

2. Pour h allant de 1 à k, calculer $f_h = f \mod \ell_h$ et $\alpha_{h,i} = \alpha_i \mod \ell_h$ pour i allant de 0 à $N-1$.

3. Pour h allant de 1 à k, il y a deux cas :
   - ☐ $\ell_h > K\,L_{max}$: appeler récursivement l'algorithme 6 pour évaluer f_h en les $\alpha_{h,i}$, avec i allant de 0 à $N-1$.
   - ☐ $\ell_h \leq K\,L_{max}$: utiliser l'algorithme 5 pour évaluer f_h en les $\alpha_{h,i}$, avec i allant de 0 à $N-1$.

4. Pour i allant de 0 à $N-1$, on a le système suivant :
   $$\begin{cases} f(\alpha_i) = f_1(\alpha_i) \mod \ell_1 \\ ... \\ f(\alpha_i) = f_k(\alpha_i) \mod \ell_k \end{cases}$$
   Utiliser le théorème des restes chinois pour obtenir l'unique solution $f(\alpha_i) \mod \prod_{i=1}^{k} \ell_i$. Retourner ensuite $f(\alpha_i) \mod r$ pour revenir dans $\mathbb{Z}/r\mathbb{Z}$

Algorithme 7 Calcul de L_{max}

```
fonction LMAX(r,d,m)
    Lmax = r
    boucler                                          ▷ Boucle infinie calculant L_max
        i = 1, prod = 1
        tant que prod < d^m (Lmax − 1)^((d−1)m+1) faire   ▷ Calcul du produit des ℓ_i
            prod = prod*ℓ_i
            i++
        fin tant que
        si Lmax < ℓ_{i−1} alors   ▷ S'il faut plus de nombre premier qu'à l'itération précédente
            renvoyer Lmax                            ▷ Récursion impossible
        fin si
        Lmax = ℓ_{i−1}
    fin boucler
fin fonction
```

Exemple 6. $n = 10000$, $d = 10$, $m = 4$, $r \simeq 2^{273} \Rightarrow L_{max} = 233$.

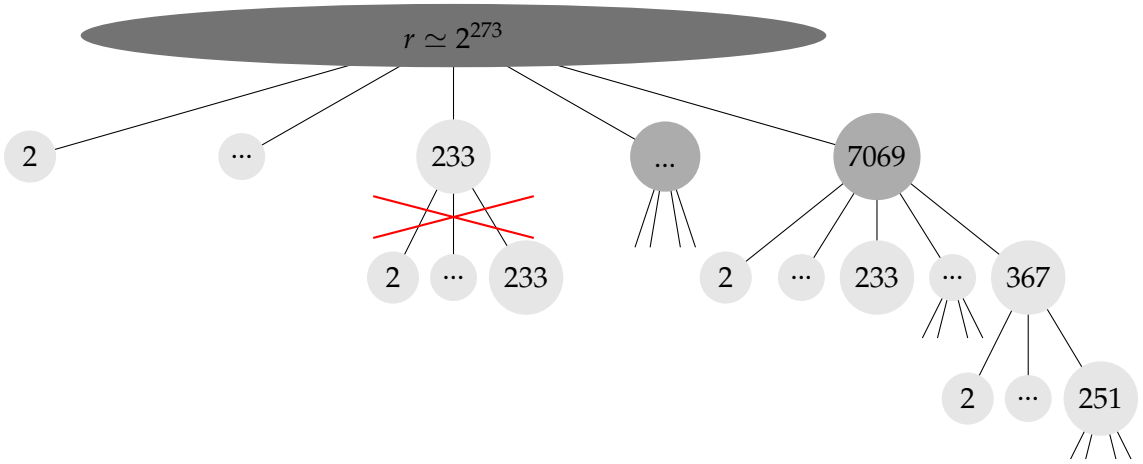

FIGURE 2 – Algorithme multimodulaire

Sur la figure 2, on remarque que pour tout module $p \leq L_{max} = 233$, il est impossible de faire un appel récursif : en effet, ceci nécessiterait de calculer modulo des nombres $\ell \geq p$. On peut remarquer que si on a un nombre premier légèrement supérieur à L_{max}, il est probablement plus lent de lancer une récursion que de calculer l'évaluation directement. Une optimisation possible serait de trouver une heuristique sur la meilleure borne pratique pour la descente des nombres premiers. C'est ici que le paramètre K intervient. Nous avons utilisé la borne expérimentale $K = 1.2$ dont le choix est discuté dans la section 4.6.2.

Remarque. L_{max} ne dépend pas de r, il ne dépend que de d et m.

Une propriété remarquable est qu'il y a autant d'évaluations dans $\mathbb{F}_\ell$ à faire pour chaque nombre premier ℓ inférieur ou égal à L_{max}. Cela vient du fait que lorsque l'on coupe un pro-

blème modulo r en des problèmes modulo des petits nombres premiers, il y aura forcément une FFT multidimensionnelle pour tous les nombres inférieurs ou égaux à L_{max} car ce sont les nombres que l'on ne peut pas découper modulo des entiers plus petits avec le CRT.

Analyse de complexité : évolution du plus grand nombre premier utilisé

Pour évaluer la complexité de l'algorithme, nous devons d'abord évaluer à quelle vitesse décroissent les nombres premiers utilisés.

Lemme 2 ([KU11], lemme 2.4)**.** Pour tout entier $x \geq 2$, le produit des nombres premiers inférieurs ou égaux à $16 \log x$ est supérieur à x.

Remarque. La constante 16 est non optimale. On peut toujours trouver une constante $c > 1$ telle que $c \log x > \prod_{i=1}^{k} \ell_i$, mais il y a une dépendance entre c et x.

À l'aide du lemme 2, on peut modéliser l'évolution du plus grand nombre premier L_i intervenant à la i-ème étape de descente, comme la suite suivante :

$$\begin{array}{rcl} L_0 & = & r\,; \\ L_{i+1} & \leq & \begin{cases} \lceil 16 \log(d^m (L_i - 1)^{(d-1)m+1}) \rceil & \text{si } \lceil 16 \log(d^m (L_i - 1)^{(d-1)m+1}) \rceil < L_i, \\ L_i & \text{sinon.} \end{cases} \end{array}$$

Cette suite décroissante d'entiers a pour limite une majoration de L_{max}.

Si on regarde la complexité par rapport à r, à chaque récursion L_i est borné par le log de L_{i-1}. Il convient donc de définir $\log^{(i)}(r)$ comme le log imbriqué i fois de r. On s'attend à voir apparaître un facteur $O(\log^{(i)}(r))$ au niveau de la grandeur de L_i.

Théorème 4
On pose $\lambda_i(x) = x \log(x) \log(\log(x)) ... \log^{(i-1)}(x)$. Pour tout i, $L_i = O(\lambda_i(m)\lambda_i(d) \log^{(i)}(r))$.

Démonstration. Démontrons cette propriété par récurrence.

Pour le cas de base, $L_0 = r = O(\lambda_0(m)\lambda_0(d) \log^{(0)}(r))$, donc la propriété est vraie pour $i = 0$.

Supposons que $L_i = O(\lambda_i(m)\lambda_i(d) \log^{(i)}(r))$. Démontrons qu'alors la propriété est vraie au rang $i + 1$.

$$\begin{aligned} L_{i+1} &< 16m \log(d) + 16dm \log(L_i) \text{ par définition,} \\ &= O(dm \log(\lambda_i(m)\lambda_i(d) \log^{(i)}(r))), \\ &= O(m \log(\lambda_i(m)) d \log(\lambda_i(d)) \log(\log^{(i)}(r))) \text{ car } \log(abc) \leq \log(a) \log(b) \log(c), \\ &= O(\lambda_{i+1}(m)\lambda_{i+1}(d) \log^{(i+1)}(r)). \end{aligned}$$

□

Cela repose sur la très mauvaise majoration que la somme de trois entiers est plus petite que leur produit, mais permettra tout de même d'obtenir une complexité quasi-linéaire pour l'algorithme 6.

La majoration de L_i permet d'avoir une majoration grossière sur k à la i-ème récursion, en remarquant que $k < \ell_k = L_i$. La valeur de k de l'algorithme dépend de i, on renomme donc k_i cette variable.

Analyse de complexité de l'algorithme 6

Théorème 5
Soit t la constante définie ci-dessous, l'algorithme 6 résout le problème d'EMM et a une complexité bornée par $\widetilde{O}((\lambda_t(d)^m + N)\lambda_{t-1}(\log(r))\lambda_t(d)^t\lambda_t(m)^{m+t+1}\log^{(t)}(r)^m)$.

Démonstration. Pour simplifier l'analyse de complexité, supposons que l'arbre de récursion soit complet, c'est-à-dire que l'on ne fait les évaluations par la FFT multidimensionnelle qu'aux feuilles de l'arbre (ceci correspond au pire des cas). Notons t la profondeur de l'arbre, on suppose t constant. Cette hypothèse est vérifiée en pratique, on a toujours $t \leq 5$. Pour avoir $t = 6$, la décroissance de $\log^{(t)}(x)$ dans la borne de L_t fait que r doit au moins être de l'ordre de $2^{2^{64}}$, ce qui est impossible en pratique. Remarquons que $L_t = L_{max}$.

Au i-ème niveau de récursion, l'algorithme est appelé $O(L_1L_2...L_{i-1})$ fois (car $k_i < L_{i-1}$ et car les appels de l'étage $i-1$ créent l'étage i).

Si $i = t$, le nombre global d'évaluation par la FFT multidimensionnelle est $O(L_1L_2...L_{t-1}) \times L_t$ (car $k_t < L_t$). Or la complexité de cette FFT est égale à $\widetilde{O}(m(L_t^m + d^m + N))$. La complexité finale pour la partie évaluation est donc égale à $\widetilde{O}((\prod_{i=1}^{t} L_i) \times m(L_t^m + d^m + N))$.

Pour les autres étapes de l'algorithme, regardons combien coûte chaque étape au i-ème niveau de récursion.

L'étape 1 demande de rechercher k_i entiers premiers successifs, le plus grand étant L_i. Cela peut se faire avec une recherche naïve du prochain nombre premier, avec une complexité bornée par $\widetilde{O}(\sqrt{L_i})$. Les k_i entiers sont multipliés entre eux successivement, et le produit final vaut au plus $O(L_{i-1})$. On en déduit que la complexité est égale à $O(L_i \log^{1+o(1)}(L_{i-1})) = \widetilde{O}(L_i)$.

L'étape 2 demande k_i réductions modulaires d'un polynôme modulo des nombres premiers, ce qui donne une complexité égale à $\widetilde{O}(L_i d^m)$. Elle demande ensuite k_i réductions modulaires de N points vectoriels, ce qui donne une complexité égale à $\widetilde{O}(L_i Nm)$. La complexité de l'étape 2 est donc bornée par $\widetilde{O}((d^m + Nm)L_i)$.

L'étape 4 demande N reconstructions d'entiers dans $\mathbb{Z}$ à partir de leurs restes modulo k_i entiers, ce qui donne une complexité égale à $\widetilde{O}(NL_i)$.

Parmi ces trois étapes, c'est l'étape 2 qui domine. La complexité est donc bornée par $\widetilde{O}((d^m + Nm)\prod_{j=1}^{i}(L_j))$.

Il faut maintenant sommer le coût de chaque étage. La complexité globale pour les trois étapes est bornée par $\widetilde{O}((d^m + Nm)\sum_{i=1}^{t-1}(\prod_{j=1}^{i} L_j))$.

La complexité finale de l'algorithme vaut donc $\widetilde{O}((\prod_{i=1}^{t} L_i) \times m(L_t^m + d^m + N) + (d^m + Nm)\sum_{i=1}^{t-1}(\prod_{j=1}^{i} L_j)) = \widetilde{O}((\prod_{i=1}^{t} L_i) \times m(L_t^m + d^m + N)$ car l'étape 3 domine sur les autres étapes. En effet, $d^m + Nm < (d^m + N)m < m(L_t^m + d^m + N)$, et

$$\begin{aligned}\sum_{i=1}^{t-1}\prod_{j=1}^{i} L_j &< \sum_{i=1}^{t-1}\prod_{j=1}^{t-1} L_j,\\ &< (t-1)\prod_{j=1}^{t-1} L_j,\\ &< \prod_{i=1}^{t} L_i \text{ car en pratique } t \leq 5.\end{aligned}$$

Il reste donc à exprimer $\widetilde{O}((\prod_{i=1}^{t} L_i) \times m(L_t^m + d^m + N))$ en fonction des paramètres. On a :

$$\prod_{i=1}^{t} L_i \leq \lambda_t(m)^t \lambda_t(d)^t \lambda_{t-1}(\log(r)) \text{ car } L_i \leq \lambda_t(m)\lambda_t(d)\log^{(i)}(r)$$

et

$$\begin{aligned}m(L_t^m + d^m + N) &\leq \lambda_t(m)(L_t^m + \lambda_t(d)^m + N),\\ &\leq (2\lambda_t(d)^m + N)\lambda_t(m)^{t+1}\log^{(t)}(r).\end{aligned}$$

D'où :

$$\begin{aligned}\widetilde{O}((\prod_{i=1}^{t} L_i) \times m(L_t^m + d^m + N)) &\leq \widetilde{O}(\lambda_t(m)^t\lambda_t(d)^t\lambda_{t-1}(\log(r)) \times (\lambda_t(d)^m + N)\lambda_t(m)^{t+1}\log^{(t)}(r)),\\ &\leq \widetilde{O}((\lambda_t(d)^m + N)\lambda_{t-1}(\log(r))\lambda_t(d)^t\lambda_t(m)^{m+t+1}\log^{(t)}(r)^m).\end{aligned}$$

La complexité est finalement bornée par $\widetilde{O}((\lambda_t(d)^m + N)\lambda_{t-1}(\log(r))\lambda_t(d)^t\lambda_t(m)^{m+t+1}\log^{(t)}(r)^m)$. □

Corollaire 2
Soit d assez grand pour avoir $m \leq d^{o(1)}$. La complexité de l'algorithme 6 est bornée par $\widetilde{O}((d^m + N)\log(r))$ [KU11, corollaire 4.3].

La complexité est finalement quasi-linéaire en la taille de l'entrée.

3.2.4 Dans $(\mathbb{Z}/r\mathbb{Z})[Y]/(E(Y))$

L'idée de l'algorithme est de voir les éléments de $(\mathbb{Z}/r\mathbb{Z})[Y]/(E(Y))$ comme des entiers en base M afin de se ramener au cas $(\mathbb{Z}/r'\mathbb{Z})$ pour un r' bien choisi, et de pouvoir appliquer l'algorithme 6.

La méthode utilisée pour passer de polynôme à entier et réciproquement est la substitution de Kronecker (définie en 3.1.1).

Algorithme 8 Multimodulaire pour les extensions d'anneaux

Entrée : $f \in (\mathbb{Z}/r\mathbb{Z})[Y]/(E(Y))[X_0, ..., X_{m-1}]$ de degré au plus $d-1$ pour chaque variable, $\alpha_0, ..., \alpha_{N-1} \in ((\mathbb{Z}/r\mathbb{Z})[Y]/(E(Y)))^m$.
Sortie : $f(\alpha_i)$ pour i allant de 0 à $N-1$.

1. Poser $\widetilde{f} = f$ et $\widetilde{\alpha}_i = \alpha_i$ pour i allant de 0 à $N-1$, avec $\widetilde{f} \in (\mathbb{Z}[Y])[X_0, ..., X_m - 1]$ et $\widetilde{\alpha}_i \in \mathbb{Z}[Y]$. Cette étape est purement symbolique.

2. Soient $M = d^m(e(r-1))^{(d-1)m+1} + 1$ et $r' = M^{(e-1)dm+1}$, calculer $\bar{f} = (\widetilde{f} \mod r') \mod (Y - M)$ et $\bar{\alpha}_i = (\widetilde{\alpha}_i \mod r') \mod (Y - M)$ pour i allant de 0 à $N-1$.
Le modulo r' n'est que symbolique, pour permettre l'utilisation de l'algorithme Multimodulaire à l'étape 3. Le modulo $Y - M$ est une évaluation de $\widetilde{f}$ en M.

3. Calculer L_{max} avec l'algorithme 7. Utiliser l'algorithme 6 dans $\mathbb{Z}/r'\mathbb{Z}$ pour évaluer $\bar{f}$ en les $\bar{\alpha}_i$.

4. Calculer les $\widetilde{f}(\widetilde{\alpha}_i)$ à partir des $\bar{f}(\bar{\alpha}_i)$ pour revenir à la forme polynomiale des entiers en base M. Pour cela, il suffit de voir les coefficients de $\widetilde{f}(\widetilde{\alpha}_i)$ comme les bits de $\bar{f}(\bar{\alpha}_i)$ par paquet de $\lfloor \log_2(M) \rfloor + 1$. Cette opération permet de passer de $\mathbb{Z}/r'\mathbb{Z}$ à $\mathbb{Z}[Y]$.
Appliquer ensuite une réduction de $\widetilde{f}(\widetilde{\alpha}_i)$ modulo r et $E(Y)$ pour obtenir $f(\alpha_i)$.

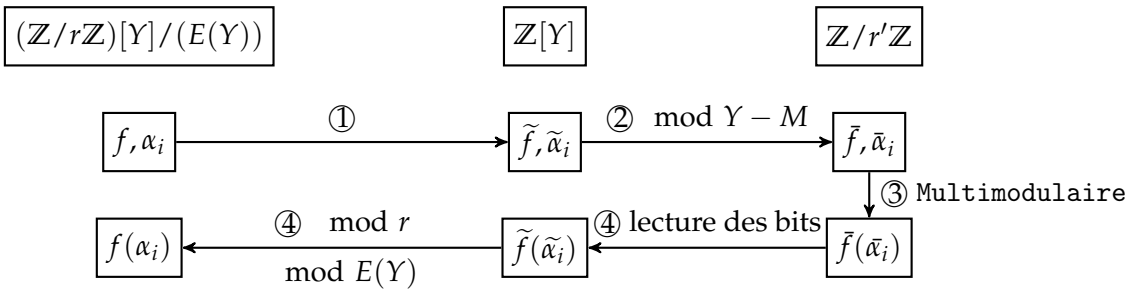

FIGURE 3 – Algorithme Multimodulaire pour les extensions

Explication de l'algorithme

1. On passe de l'ensemble $(\mathbb{Z}/r\mathbb{Z})[Y]/(E(Y))$ à l'ensemble $\mathbb{Z}[Y]$, pour permettre à l'étape 2 le passage d'un polynôme à un entier en base M.

2. L'objectif principal est de passer de $\mathbb{Z}[Y]$ à $\mathbb{Z}/r'\mathbb{Z}$ pour un r' bien choisi, afin de pouvoir utiliser l'algorithme 6 à l'étape suivante. On utilise donc le procédé décrit précédemment pour passer d'un polynôme à un entier en base M, et réciproquement. Le problème ici est de prévoir l'inversion de ce procédé, il est aisé de remarquer que les coefficients de $\widetilde{f}$ sont strictement inférieurs à r, mais le résultat de l'inversion ne sera pas $\widetilde{f}$ mais $\widetilde{f}(\widetilde{\alpha_i})$. Pour que l'inversion réussisse, il faut déterminer la valeur maximale $M-1$ que peut valoir un coefficient de $\widetilde{f}(\widetilde{\alpha_i})$.
   Pour cela, on majore d'abord les éléments de $(\mathbb{Z}/r\mathbb{Z})[Y]/(E(Y))$. On pose $Y=1$: cela revient à considérer que tous les coefficients sont devant la même puissance de Y (ici Y^0). Avec cette hypothèse, on obtient une majoration égale à $e(r-1)$. On en déduit que $\widetilde{f}(\widetilde{\alpha_i})$ vaut au plus $d^m(e(r-1))^{(d-1)m+1}$ (f a d^m termes, chaque terme possède un coefficient et m variables, chacune est de degré au plus $d-1$), ce qui nous donne la valeur de $M-1$.
   Connaissant la valeur de M, il faut maintenant déterminer r'. Cela revient à borner $\bar{f}(\bar{\alpha_i})$, et le borner revient à trouver le degré en Y de $\widetilde{f}(\widetilde{\alpha_i})$. Un élément de $(\mathbb{Z}/r\mathbb{Z})[Y]/(E(Y))$ est de degré au plus $e-1$, et l'évaluation de $\widetilde{f}$ en un tel élément engendre un degré d'au plus $(e-1)(d-1)m$. Par conséquent, on choisit $r'=M^{(e-1)(d-1)m+1}$, afin que le modulo r' ne conserve que la partie intéressante du résultat.

3. On appelle l'algorithme 6 dans $\mathbb{Z}/r'\mathbb{Z}$.

4. On applique la réciproque de la substitution de Kronecker (3.1.1) pour passer d'un entier à sa représentation polynomiale. Le résultat étant dans $\mathbb{Z}[Y]$, la réduction modulaire permet de revenir dans $(\mathbb{Z}/r\mathbb{Z})[Y]/(E(Y))$.

Analyse de complexité

Théorème 6
L'algorithme 8 résout le problème d'EMM et a une complexité bornée par
$\widetilde{O}((\lambda_t(d)^m+N)\lambda_t(\log(q))\lambda_t(d)^{t+2}\lambda_t(m)^{m+t+3}\times\log^{(t)}(d^2m^2\log(q)\log(\log(q)))^m)$.

Démonstration. On analyse la complexité des différentes étapes de l'algorithme :

1. f possède d^m coefficients dans $(\mathbb{Z}/r\mathbb{Z})[Y]/(E(Y))$, et les α_i contiennent en tout Nm éléments de $(\mathbb{Z}/r\mathbb{Z})[Y]/(E(Y))$, ce qui fait d^m+Nm éléments. Changer de représentation a donc une complexité bornée par $\widetilde{O}(d^m+Nm)$.

2. Comme précédemment, il y a d^m+Nm éléments (de $\mathbb{Z}[Y]$) à évaluer. Avec la méthode de Horner, évaluer un polynôme de degré $e-1$ en M coûte e multiplications, mais sachant que la taille des opérandes augmente linéairement, la complexité est égale à $O(e^2\log^{1+o(1)}(M))$.

On a donc une complexité totale égale à $O((d^m + Nm)e^2 \log^{1+o(1)}(M))$.

3. La complexité de l'algorithme 6 est égale à $\widetilde{O}((\lambda_t(d)^m + N)\lambda_{t-1}(\log(r'))\lambda_t(d)^t\lambda_t(m)^{m+t+1} \times \log^{(t)}(r')^m)$.

4. Pour revenir à $\widetilde{f}(\widetilde{\alpha_i})$, il faut convertir N entiers de valeur au plus r', en polynôme. Cela se fait avec une complexité égale à $O(N\log(r'))$. On obtient N polynôme de degré au plus $(e-1)(d-1)m$, avec des coefficients valant au plus $M-1$. La réduction modulaire par r coûte $\widetilde{O}(Nedm\log(M))$, et la réduction modulo $E(Y)$ coûte ensuite $\widetilde{O}(Nedm\log(r))$.

En remarquant que $\log(r') = O(edm\log(M))$, l'étape 3 domine. Réécrivons r' en fonction de $q = r^e$:

$$\begin{aligned}
\log(r') &= O(edm\log(M)),\\
&\leq O(edm\log(d^m(er)^{dm})),\\
&\leq O(edm(m\log(d) + dm\log(er))),\\
&\leq O(ed^2m^2(\log(e)+\log(r))) \text{ car } m\log(d) \text{ est négligeable face à } dm,\\
&\leq O(ed^2m^2(\log(e)\log(r)+\log(r)\log(\log(r)))),\\
&\leq O(d^2m^2e\log(r)(\log(e)+\log(\log(r)))),\\
&\leq O(d^2m^2\log(q)\log(\log(q))).
\end{aligned}$$

Sachant que $\lambda_t(\log(r')) = O(\lambda_t(d^2m^2\log(q)\log(\log(q)))) = \widetilde{O}(\lambda_t(d)^2\lambda_t(m)^2\lambda(\log(q)))$, la complexité finale peut donc être exprimée comme $\widetilde{O}((\lambda_t(d)^m + N)\lambda_t(\log(q))\lambda_t(d)^{t+2}\lambda_t(m)^{m+t+3} \times \log^{(t)}(d^2m^2\log(q)\log(\log(q)))^m)$.

□

Corollaire 3
Soit d assez grand pour avoir $m \leq d^{o(1)}$. La complexité de l'algorithme 8 est bornée par $\widetilde{O}((d^m + N)\log(q))$ [KU11, corollaire 4.5].

Une fois encore la complexité est quasi-linéaire en la taille de l'entrée.

3.3 Transformée de Fourier rapide dans un corps premier

La FFT dans $\mathbb{F}_p$ est un facteur multiplicatif dans le coût global de l'algorithme d'évaluation multipoint multivariée, l'algorithme utilisé se doit donc d'être le plus rapide possible. La difficulté de trouver un algorithme efficace vient du fait que classiquement, la FFT ne s'applique que si le nombre de point est une puissance de 2. Il existe cependant des algorithmes plus généraux.

3.3.1 Algorithmes classiques

Tout d'abord, un premier algorithme qui n'est pas aussi efficace que les FFT, mais qui a le mérite de s'appliquer quel que soit le nombre premier p, avec une complexité de $O(M(p)\log(p))$,

est l'algorithme d'évaluation multipoint univariée rapide [Pot14]. Il peut être vu comme une FFT généralisée, et c'est cette généralité qui fait perdre en performance. L'algorithme est pratique pour avoir une première version de l'algorithme d'évaluation multipoint multivariée fonctionnelle, mais n'est pas le plus efficace puisqu'il ne tire pas pleinement parti de la structure de $\mathbb{F}_p$. Une manière de l'exploiter est de résoudre le problème de transformée de Fourier (problème 4), car il utilise l'existence d'une racine primitive $(p-1)$-ème de l'unité .

L'algorithme de FFT le plus connu est celui où la racine primitive est d'ordre une puissance de 2. Elle est un cas particulier de l'algorithme précédent, et le fait que l'ordre soit une puissance de 2 est nécéssaire pour suivre la structure d'arbre binaire de l'algorithme. L'avantage de cette méthode par rapport à l'algorithme précédent est qu'elle simplifie les calculs, et permet d'avoir une complexité bornée par $O(2^k \log(2^k))$. Pour $p = 2^k + 1$, on obtient donc une complexité bornée par $O(p \log(p))$, ce qui semble optimal : cette méthode permet de ne pas avoir à effectuer de multiplication polynomiale.
Dans $\mathbb{F}_p$, le défaut de cet algorithme est qu'il ne s'applique qu'au cas où les nombres premiers sont de Fermat (cinq sont connus), il est donc inutilisable ici.

L'idée des prochaines méthodes est de ramener le problème à celui d'une multiplication polynomiale. Cette multiplication étant optimisée dans `FLINT`, cela permet de ne pas avoir à recoder des algorithmes optimisés et potentiellement compliqués. De plus, avoir une complexité égale à $O(M(p) + p\log(p))$ donnera un meilleur résultat asymptotique qu'avoir $O(M(p)\log(p))$.

3.3.2 Évaluation multipoint d'une suite géométrique

Proposition 3. Soit $F(X) = \sum_{i=0}^{n-1} f_j X^j$ un polynôme, l'évaluer en les n premiers points d'une suite géométrique de raison q se fait avec une complexité bornée par $O(M(n))$ opérations [Bos07, proposition 7].

Démonstration. La méthode consiste à poser $b_i = q^{\frac{i(i-1)}{2}}$ pour i allant de 0 à $2n-2$, et $c_i = \frac{f_i}{b_i}$ pour i allant de 0 à $n-1$. Ces suites se calculent en $O(n)$ opérations car $b_{i+1} = b_i q^i$. Pour i allant de 0 à $n-1$, on réécrit $F(q^i)$ comme $b_i^{-1} \sum_{j=0}^{n-1} c_j b_{i+j}$. Cette somme est le terme de degré $n-1+i$ du produit polynomial de $\sum_{i=0}^{n-1} c_i X^{n-1-i}$ par $\sum_{i=0}^{2n-2} b_i X^i$. Ce produit s'effectue avec une multiplication de complexité $O(M(n))$. D'où un algorithme avec une complexité bornée par $O(M(n))$ opérations.

□

En posant $n = p-1$, et en remarquant que les racines n-ème de l'unité forment une suite géométrique de raison une racine primitive n-ème de l'unité, cet algorithme permet d'évaluer un polynôme en tout point de $\mathbb{F}_p^*$.

3.3.3 FFT de Rader pour un nombre de point premier

Cet méthode [Smi07b] ne s'applique que dans le cas où le nombre de points est premier, ce qui donne dans notre cas $p-1$ est premier donc nécessairement $p = 3$. Par conséquent la version corps fini de l'algorithme n'a aucun intérêt en pratique. Par contre, elle donne un bon exemple d'algorithme qui permet de passer d'une évaluation à une multiplication polynomiale.

3.3.4 FFT de Bluestein

On souhaite évaluer un polynôme en les racines N-ème de l'unité (dans $\mathbb{F}_p$, $N = p-1$), c'est-à-dire calculer, pour k allant de 0 à $N-1$, l'évaluation $X(k) = \sum_{n=0}^{N-1} x_n(\omega^{-k})^n$.

À un facteur près, la méthode de Bluestein [Smi07a] permet de réécrire cette expression comme les coefficients résultants d'une multiplication polynomiale :

$$\begin{aligned}
X(k) &= \sum_{n=0}^{N-1} x_n \omega^{-kn} \omega^{\frac{n^2+k^2}{2}} \omega^{-\frac{n^2+k^2}{2}} \text{ en décomposant 1 comme un nombre multiplié par son inverse,} \\
&= \omega^{-\frac{k^2}{2}} \sum_{n=0}^{N-1} x_n \omega^{-\frac{n^2}{2}} \omega^{\frac{(k-n)^2}{2}} \text{ en reconnaissant une identité remarquable,} \\
&= \omega^{-\frac{k^2}{2}} \sum_{n=0}^{N-1} a_n b_{k-n} \text{ en posant les notations suivantes :}
\end{aligned}$$

$a_n = x_n \omega^{-\frac{n^2}{2}}$ qui est défini pour n allant de 0 à $N-1$,
$b_n = \omega^{\frac{n^2}{2}}$ qui est défini pour n allant de $-(N-1)$ à $N-1$.

Pour réécrire la somme comme un produit de polynômes, une première idée est de considérer que a_n et b_n sont les coefficients respectifs du monôme en X^n de polynômes $A(X)$ et $B(X)$. Pour éviter d'avoir des polynômes avec des monômes de degré négatif, on choisit de les multiplier par X^{N-1}. On obtient donc $A(X) = \sum_{n=N-1}^{2N-2} a_{n-(N-1)} X^n$ et $B(X) = \sum_{n=0}^{2N-2} b_{n-(N-1)} X^n$.

On remarque alors que le terme de $A(X) \times B(X)$ de degré $2N-2+k$ est $(\sum_{n=0}^{N-1} a_n b_{k-n}) X^{2N-2+k}$. Les termes qui nous intéressent sont les termes des puissances $2N-2$ à $3N-3$.

Il y a d'autres manières de réécrire la somme comme produits polynomiaux. Je propose en annexe (cf. section A) la démonstration de ma propre méthode (non connue à ma connaissance). En posant $A(X) = \sum_{n=0}^{N-1} a_n X^n$ et $B(X) = \sum_{n=0}^{N-1} b_n X^n$, et en notant $A^*(X) = \sum_{n=0}^{N-1} a_{N-1-n} X^n$ le polynôme aux inverses de A, je démontre que le terme de degré k de $(A(X) \times B(X)) + (A^*(X) \times B(X))^*$ est $(\sum_{n=0}^{N-1} a_n b_{k-n}) X^k$ pour $k \neq N-1$ (il faut poser le second produit à 0 pour $k = N-1$). Ici les termes qui nous intéressent dans les deux produits polynomiaux sont ceux de degré 0 à $N-1$, on peut donc se limiter à ne calculer que les N premiers termes des produits. C'est cette dernière méthode qui a été choisie pour l'implémentation de la FFT de Bluestein.

L'inconvénient de la méthode de Bluestein est qu'elle ne s'applique pas directement dans $\mathbb{F}_p$, du fait qu'une racine primitive n'a pas de racine carré. Toutefois, il est possible de passer dans une extension quadratique pour créer la racine carré, et permettre ainsi à l'algorithme de s'appliquer. Une autre solution face à ce problème est la variante de l'algorithme proposée ci-dessous.

Remarquons également que cette méthode permet d'appliquer la FFT sur un polynôme de degré au plus $p-2$, il est donc nécessaire d'utiliser le fait que $X^p - 1 = 1$ avant d'appliquer cet algorithme.

3.3.5 Une variante de la FFT de Bluestein

La méthode originale [HVDHL14] est valide pour ω une racine N-ème, nous spécialisons cette méthode pour $N = p - 1$, N pair.

Posons les suites $f_i = \omega^{i^2}$, $f'_i = \omega^{i^2+i}$, $g_i = \omega^{-i^2} + \omega^{-i^2-i}$ et $\sigma = (-1)^{\frac{p-1}{2}}$.

Soient $F(X) = \sum_{n=0}^{p-2} x_n f_n X^n$ et $G(X) = \sum_{n=0}^{p-2} g_n X^n$, la méthode utilise le fait que l'on peut se ramener à calculer $F(X) \times G(X) \mod X^{p-1} - 1$. En posant $H(X)$ le polynôme résultat et h_i son i-ème coefficient, on a :

— pour k pair, on pose $k = 2i$:

$$X(2i) = \frac{1}{2} f_i (h_i + \sigma h_{i+\frac{p-1}{2}})$$

— pour k impair, on pose $k = 2i + 1$:

$$X(2i+1) = \frac{1}{2} f'_i (h_i - \sigma h_{i+\frac{p-1}{2}})$$

3.4 Complexité finale de la composition modulaire

Maintenant que la complexité de l'EMM est connue, on peut revenir sur la complexité de la composition modulaire.

Théorème 7
Soit d suffisamment grand pour avoir $m \leq d^{o(1)}$, si on a accès à $d^m md$ coefficients dans R dont les différences sont deux à deux inversibles, alors l'algorithme 2 calcule une composition modulaire en $\widetilde{O}(n^{1+\frac{2}{m}})$ opérations dans R.

Démonstration. La complexité de la composition modulaire est $\widetilde{O}(T(d, m, N) + n^{1+\frac{2}{m}} m^2)$. Comme $N = d^m md \leq nmd^2$, on obtient $T(d, m, N) = \widetilde{O}(n^{1+\frac{2}{m}} m)$. L'algorithme de réduction domine donc, et la complexité finale de l'algorithme de composition modulaire est bornée par $\widetilde{O}(n^{1+\frac{2}{m}} m^2) = \widetilde{O}(n^{1+\frac{2}{m}})$ en supposant $m \leq d^{o(1)}$.

□

La complexité est quasi-linéaire en le degré pour m assez grand.

4 Implémentation et résultats expérimentaux

4.1 Caractéristique de la machine

La machine utilisée est un ordinateur 64 bits doté de 4 cœur physiques, 8 coeurs logiques, d'un processeur `Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz` et d'une RAM de 16 `Go`. La capacité de la RAM est significative puisque l'algorithme 5 nécessite beaucoup de mémoire ($O(p^m)$). De plus, l'avantage de l'algorithme de Kedlaya et Umans sur celui de Brent et Kung est sur la complexité asymptotique quasi-linéaire : pour faire apparaître cet avantage, nous devons utiliser de grandes valeurs de p ($p \sim 2^{200}$).

4.2 Choix du langage

L'algorithme de Kedlaya-Umans a été programmé en `C` à l'aide de la bibliothèque `FLINT` (Fast Library for Number Theory, version 2.5.2) [HJP13]. Programmer en `C` permet d'avoir un code efficace, et la bibliothèque `FLINT` est très pratique car elle gère les entiers de taille arbitraire ainsi que les structures mathématiques élémentaires (éléments de $\mathbb{F}_p$, $\mathbb{F}_q$, ...). Elle possède un grand nombre de fonctions classiques implémentées de manière efficace, comme par exemple l'algorithme d'évaluation multipoint univariée rapide. Cela permet de se concentrer sur l'algorithme de Kedlaya-Umans, bien qu'il a fallu ajouter quelques éléments manquants (polynômes multivariés, interpolation rapide dans les corps finis, FFT dans $\mathbb{F}_p$). La bibliothèque `FLINT` repose sur les bibliothèques `GMP` et `MPFR`.

Pour tester l'algorithme à un plus haut niveau, afin de faciliter le débug, il a d'abord été codé dans le langage `Julia` (version 0.4.6) à l'aide du package `Nemo` (version 0.4.0) [FHHJ16]. `Julia` est un langage destiné au calcul scientifique, et est performant, notamment de par sa compilation à la volée (*Just-In-Time compilation*), ce qui permet d'avoir des performances proches du C.
Le package `Nemo` permet d'utiliser des structures mathématiques comme les corps finis. Son fonctionnement est principalement celui d'un *wrapper* : il utilise des objets Julia qui reposent sur des appels `C` de certaines bibliothèques (principalement `FLINT` et `PARI`), et permet ainsi de programmer avec ces bibliothèques mais à un plus haut niveau. Le côté haut niveau est utile du fait qu'il permet notamment d'utiliser des polynômes multivariés (construit comme un polynôme à coefficients des polynômes), la partie multivariée étant l'une des tâches les plus compliquées à gérer. On peut donc dans un premier temps s'initier à `FLINT` en restant plus haut niveau, tout en n'ayant pas besoin de gérer soi-même la structure multivariée, permettant ainsi de coder une première version valide de l'algorithme. La capacité d'appeler des fonctions `FLINT` directement depuis `Nemo` permet aussi de garder ce code `Julia`, et de remplacer une brique de l'algorithme par une version `C` pour vérifier la validité. Toutefois, il existe une incompatibilité entre les tableaux `Julia` et les vecteurs de `FLINT`, obligeant à créer ses propres *wrappers* pour faire certains appels `C`.

4.2.1 Exemple de code `Julia`

Voici une implémentation en `Julia` de la fonction qui vérifie si $g^k = h \mod p$.

```
using Nemo # Appel au package Nemo

function verif_dlp(g,k,h,p)
  return h == powmod(g,k,p) # g^k mod p
end

g = fmpz(2)
k = 3
h = fmpz(1)
p = fmpz(7)
res = verif_dlp(g,k,h,p)
println(res) # Renvoie true
```

On constate que le code en `Julia` est générique, ce qui permet d'avoir un code plutôt simple.

4.2.2 Exemple de code FLINT

On propose d'implémenter la même fonction en FLINT.

```
// g^k mod p == h ?
void verif_dlp(fmpz_t g, mp_limb_t k, fmpz_t h, fmpz_t p)
{
  fmpz_t gk; // Déclaration d'un entier de taille arbitraire
  fmpz_init(gk); // Initialisation
  fmpz_powm_ui(gk, g, k, p); // gk = g^k mod p

  if(fmpz_equal(gk, h)) {
    flint_printf("g^k == h mod p\n");
  } else {
    flint_printf("g^k != h mod p\n");
  }
  fmpz_clear(gk); // Libération
}
```

4.2.3 Exemple de *wrapper* Julia

Julia possède une structure générique Array qui permet de faire des tableaux d'éléments d'un type donné. Un défaut de ce langage est que demander de faire un tableau de fmpz produira automatiquement un tableau de fmpz_t. Or FLINT utilise une structure de type fmpz* pour les vecteurs de fmpz, ce qui fait qu'elle est incompatible avec les tableaux de Julia. Les vecteurs de FLINT servent dès qu'il y a besoin d'une liste de points en argument, comme par exemple pour l'évaluation multipoint univariée. Ce défaut dans le langage explique pourquoi le package Nemo ne propose pas d'appels à certaines fonctions de FLINT qui sont cruciales pour nos algorithmes. Je propose donc une solution qui est de définir un type vecteur en Julia, à l'aide d'un appel C à la fonction d'initialisation de vecteur de FLINT. Une meilleure solution aurait été de trouver comment modifier la structure Array pour qu'elle génère des tableaux de fmpz.

```
# Déclaration d'un type Julia fmpz_vec
type fmpz_vec
  coeffs::Ptr{fmpz}  # Liste des éléments du vecteur
  n::Int             # Nombre d'élément du vecteur

  # Déclaration d'une fonction Julia d'initialisation d'un objet fmpz_vec
  function fmpz_vec(n::Int)
    z = new() # Création de l'objet
    # Appel C à _fmpz_vec_init pour initialiser la liste
    z.coeffs = ccall((:_fmpz_vec_init, :libflint), Ptr{fmpz}, (Int,), n)
    z.n = n
    return z
  end

end
```

4.3 Vue d'ensemble du travail d'implémentation

L'algorithme de composition modulaire a été codé dans quatre versions, chaque version dépendant du type des coefficients. À partir de la bibliothèque `GMP`, `FLINT` définit les quatre types suivants :

1. `mp_limb_t` qui est équivalent à un `uint64`, il permet de travailler dans $\mathbb{F}_p$ en petite caractéristique (p un mot machine).
2. `fmpz_t` qui correspond aux entiers multiprécisions, il permet de travailler dans $\mathbb{F}_p$ en grande caractéristique.
3. `fq_nmod_t` qui correspond à un élément de $\mathbb{F}_q$ en petite caractéristique. Il est implémenté comme un polynôme à coefficients des éléments de type `mp_limb_t`, c'est-à-dire un élément de type `nmod_poly_t`.
4. `fq_t` qui correspond à un élément de $\mathbb{F}_q$ en grande caractéristique. Il est implémenté comme un polynôme à coefficients des éléments de type `fmpz`, c'est-à-dire un élément de type `fmpz_poly_t`.

On a le graphe de dépendances suivant entre les différents algorithmes (figure 4).

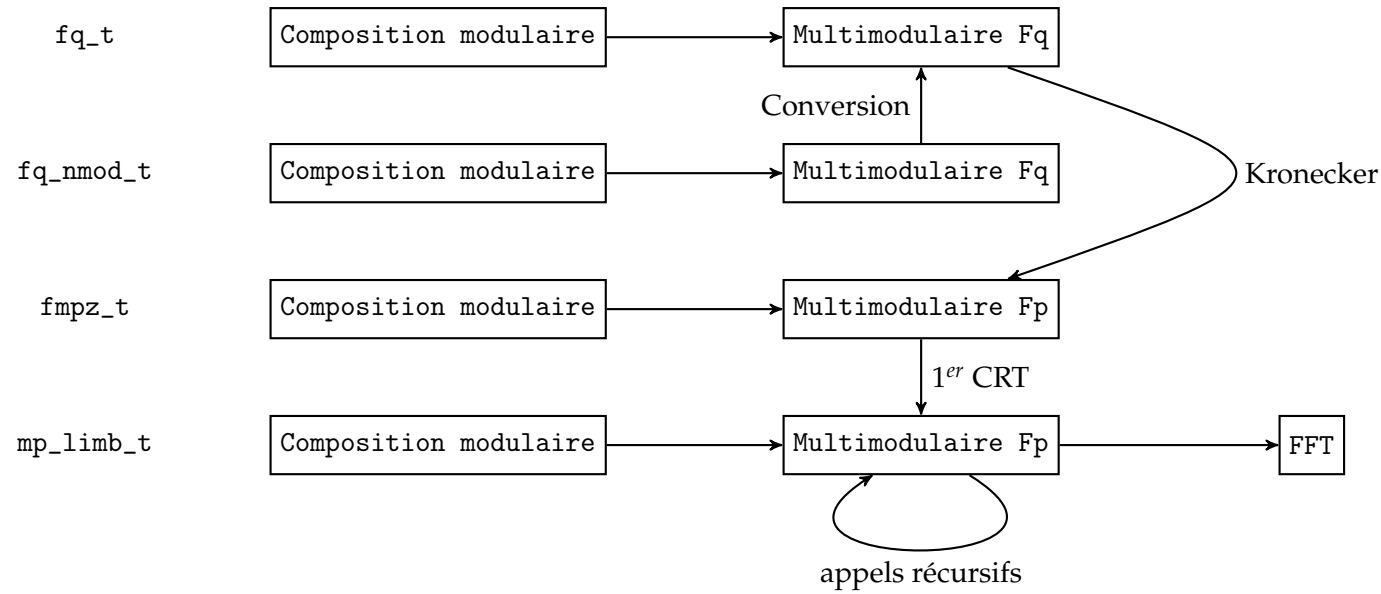

FIGURE 4 – Graphe de dépendances entre les différents algorithmes

4.4 Choix de la structure multivariée

Pour le choix de la structure en mémoire qui représentera un polynôme multivarié à m variables, de degré au plus $d-1$ en chaque variable, il y a principalement deux façons de procéder :

- Soit le représenter comme un polynôme en X_{m-1} dont les coefficients sont des polynômes à $m-1$ variables, et appliquer cette représentation récursivement sur les coefficients. C'est une représentation récursive.
- Soit le représenter comme un tableau en m dimensions de taille $d \times d \times ... \times d$. C'est une représentation hypercubique.

Les deux méthodes permettent d'avoir une structure prenant d^m coefficients en mémoire. L'un des avantages de la représentation hypercubique est que l'on peut la voir comme un tableau unidimensionnel avec accès multidimensionnel. Cela coïncide en mémoire avec la représentation d'un polynôme univarié (tableau unidimensionnel) : passer d'un polynôme univarié à sa substitution de Kronecker ne change rien en mémoire ! Ce n'est que la façon de voir le polynôme qui change.

Exemple 7 (Dimension 2 : $n = 9$, $d = 3$, $m = 2$)**.**

$$\begin{aligned} f(x) &= 9 + x + 2x^2 + 3x^3 + 4x^4 + \\ &\quad 5x^5 + 6x^6 + 7x^7 + 8x^8 \end{aligned} \qquad \begin{aligned} \psi_{3,2}(f) &= 9 + x_0 + 2x_0^2 + \\ &\quad 3x_1 + 4x_0x_1 + 5x_0^2x_1 + \\ &\quad 6x_1^2 + 7x_0x_1^2 + 8x_0^2x_1^2 \end{aligned}$$

$$\begin{pmatrix} 9 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{pmatrix} \qquad \begin{pmatrix} 9 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix}$$

C'est la représentation hypercubique qui a été choisie pour le code `FLINT`, bien qu'en `Nemo` ce soit la structure récursive car c'est la seule déjà implémentée. La représentation hypercubique utilise ici un tableau unidimensionnel avec accès multidimensionnel.

Implémentation

La manière de créer la structure polynôme multivariée est basée sur la structure de polynôme déjà existante dans `FLINT`, notamment dans l'objectif de suivre la logique des développeurs afin que le langage reste cohérent.

La structure reprend les mêmes noms que ceux des polynômes, en remplaçant le mot poly par multi_poly dans le nom. Elle a pour attribut un pointeur sur les coefficients du polynôme, le nombre de coefficients, la caractéristique p modulo laquelle sont définis les coefficients, et les paramètres d et m. Pour les coefficients dans $\mathbb{F}_q$, l'accès à l'écriture de q comme p^e se fait par l'ajout d'une structure `contexte` qui est donnée en argument à toute fonction demandant un polynôme multivariée. Elle n'est donc pas stockée au sein de la structure. Il y a également un attribut pour le coût mémoire de l'allocation des coefficients, qui a été laissé en commentaire car il n'a pas été utilisé dans le code produit.

Des fonctions d'initialisation et de libération ont été implémentées. La première permet d'initialiser tous les attributs, et alloue le vecteur des coefficients. La seconde libère le vecteur des coefficients. Pour simplifier l'implémentation, il a été décidé de toujours allouer d^m coefficients. Cela diffère des polynômes univariés où les fonctions n'allouent les coefficients que

jusqu'au coefficient dominant (qui est non nul, si le coefficient dominant devient nul, alors les coefficients après le nouveau coefficient dominant sont désalloués).

4.5 Composition modulaire rapide (algorithme de réduction)

Implémentation

L'implémentation en `FLINT` consiste principalement à des appels de fonction déjà existantes. Cependant, la fonction d'interpolation de `FLINT` ne prend pas en compte la réduction modulaire. Elle ne s'applique que si le polynôme interpolé est à coefficients dans $\mathbb{Z}$. Par conséquent, il faut coder un algorithme d'interpolation rapide dans les corps finis. C'est la méthode d'interpolation de Lagrange [Pot14] qui a été choisie, notamment car elle permet de réutiliser la structure d'arbre des sous-produits et la fonction d'évaluation multipoints univariée rapide déjà existantes.

Voici un résumé de ma fonction `fmpz_mod_poly_compose_mod_kedlaya_umans` :

Entrée : $f(X)$, $g(X)$ et $h(X)$ de type `fmpz_mod_poly_t`.
Sortie : $res(X)$ de type `fmpz_mod_poly_t`.

1. Initialisation de la structure multivariée avec ma fonction `fmpz_mod_multi_poly_init`, et copie des coefficients de $f(X)$ un à un avec la fonction `fmpz_mod_poly_get_coeff_fmpz`.
2. Utilisation de la fonction `fmpz_mod_poly_powmod_ui_binexp` pour calculer les puissances d-ème de $g(X)$ modulo $h(X)$.
3. Choix des points d'évaluation, et utilisation de la fonction `fmpz_mod_poly_evaluate_fmpz_vec` pour évaluer les $g_i(X)$ en ces points.
4. Utilisation de ma fonction `fmpz_mod_multi_poly_multimodular` pour résoudre l'EMM.
5. Utilisation de ma fonction `fmpz_mod_poly_interpolate_fmpz_vec_fast` pour interpoler.
6. Utilisation de la fonction `fmpz_mod_poly_rem` pour effectuer la réduction modulaire du résultat précédent par $h(X)$.

Pour le choix des points d'évaluation à l'étape 3, les points choisis dans des corps finis ont juste besoin d'être distincts pour avoir leur différence inversible. Par conséquent, dans $\mathbb{F}_p$ on choisit tout simplement les entiers de 0 à $N-1$. S'il n'y a pas assez de points, on renvoie une erreur. Dans $\mathbb{F}_q$, s'il n'y a pas assez de points avec la méthode précédente, on ajoute des éléments de l'extension. S'il n'y a toujours pas assez de points, on renvoie une erreur. Les erreurs peuvent être évitées, il suffit de construire une extension plus grande pour avoir suffisamment de point.

Une idée d'amélioration qui n'a pas été implémentée est de choisir comme points une suite géométrique. Il existe des algorithmes de complexité égale à $O(M(N))$ pour l'évaluation multipoint de suite géométrique (cf. section 3.3.2) et l'interpolation associée [Bos07]. La seule difficulté est de trouver un élément q tel que pour tout $i \in [\![1, N-1]\!]$, $q^i \neq 1$. Dans $\mathbb{F}_p$ en grande caractéristique, on peut supposer qu'en prenant les premiers entiers, on en trouvera rapidement un qui convient. C'est d'ailleurs la même méthode pour trouver un générateur de $\mathbb{F}_p^\times$.

4.6 Multimodulaire dans $\mathbb{Z}/r\mathbb{Z}$

4.6.1 Implémentation

Il faut remarquer que si r est un entier inférieur ou égal à $2^{2^{64}}$, dès le premier appel récursif, l'algorithme 6 est appelé avec r un mot machine. Il faut donc coder l'algorithme avec deux types différents (cf. la figure 4). Une chose à bien faire attention, est que quelque soit la taille de r, le nombre obtenu en remontant avec le théorème chinois des restes peut être un entier multiprécision : il faut donc récupérer le résultat en tant que tel, et également calculer la borne $d^m(r-1)^{m(d-1)+1}$ avec des entiers multiprécisions, ainsi que l'accumulateur du produit des nombres premiers.

Pour la recherche des nombres premiers lors de l'étape 1, on utilise une fonction `FLINT` qui propose une table des nombres premiers en lecture seule. D'après la documentation, cette table est précalculée, et est agrandie si l'on a besoin de plus de nombres premiers. Sachant que le nombre premier le plus grand dont on aura besoin sera le p_k du premier appel à l'algorithme 6, on récupère cette table lors de cet appel, et on la transmet à tous les appels récursifs suivants, ce qui évite de refaire des recherches de nombres premiers.

4.6.2 Étude de L_{max}

Les tableaux suivants représentent l'évolution du plus grand nombre premier intervenant en fonction du niveau de récursion et des paramètres de l'algorithme. L'entier t est le nombre maximal de récursion possible et $L_t = L_{max}$.

Résultats pour $n = 10000$, $r = 2^{200} + 235$.

d	m	L_1	t	L_t
2	14	2141	4	89
3	9	2707	4	109
4	7	3137	4	137
5	6	3557	5	151
7	5	4391	5	191
10	4	5233	4	233
22	3	9007	4	431
100	2	27803	5	1511

Résultats pour $n = 10000$, $r = 2^{1000} + 297$.

d	m	L_1	t	L_t
2	14	10501	4	89
3	9	13309	4	109
4	7	15383	4	137
5	6	17489	5	151
7	5	21673	5	191
10	4	25849	4	233
22	3	44587	5	431
100	2	138311	5	1511

Résultats pour $n = 100$, $r = 2^{200} + 235$.

d	m	L_1	t	L_t
2	7	1153	4	43
3	5	1579	4	61
4	4	1871	5	71
5	3	1871	4	71
10	2	2707	4	107

Évolution de L_i.

d	m	L_1	L_2	L_3	L_4	L_5
2	7	1153	71	47	43	
3	5	1579	101	67	61	
4	4	1871	113	79	73	71
5	3	1871	113	79	71	
10	2	2707	173	113	107	

On constate que comme prévu, L_t ne dépend pas du nombre premier r initial. Par contre, la différence de la valeur choisie pour r se voit nettement sur la valeur de L_1. Dans l'exemple,

on remarque que prendre un nombre premier r ayant 5 fois plus de bits engendre des valeurs de L_1 environ 5 fois plus grandes. Cela est cohérent avec le fait que L_1 a une valeur bornée par $O(\log(r))$.

Pour l'évolution des L_i, on observe bien le comportement de logarithme imbriqué. De ce fait, il semble logique que ce ne soit pas intéressant de descendre jusqu'au niveau de récursion maximal. À partir de $i = 4$, on perd nettement du temps puisque L_3 est à peu près égal à L_t, ce qui fait que continuer la récursion revient quasiment à doubler le nombre de FFT pour chaque nombre premier inférieur ou égal à L_t, sans compter les CRT à faire pour reconstruire la solution. Cela nous amène à vouloir faire l'évaluation par la FFT multidimensionnelle qu'à partir d'une certaine valeur KL_t. Si on choisit par exemple la borne $2L_t$, on effectue au plus 2 étages de CRT (d'après les tableaux). Si on abaisse la borne à $1.2L_t$, on effectue au plus 3 étages de CRT. Il convient de déterminer expérimentalement la valeur optimale de la constante K.

L'algorithme 5 a une complexité mémoire bornée par $O(p^m)$. On constate une hausse assez significative de la valeur de L_t pour m petit (deux ou trois), mais elle reste négligeable fasse à la croissance de m dans la complexité mémoire. L'algorithme 5 reste donc coûteux en mémoire de manière croissante en m.

4.6.3 Étude de K

On étudie expérimentalement la borne à partir de laquelle il est plus rapide de faire un CRT qu'une FFT directe. Les tests sont effectués avec $r = 2^{200} + 235$.

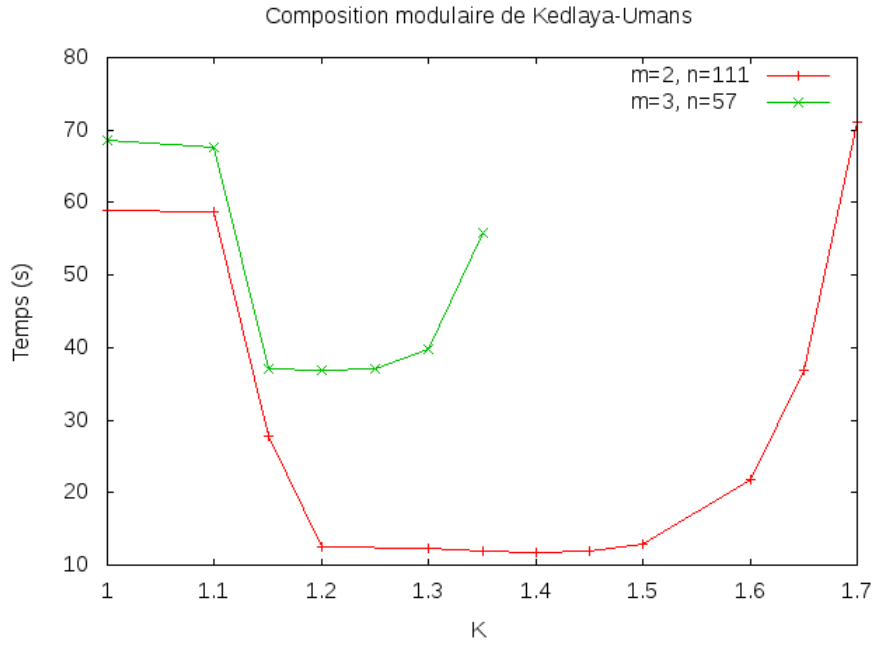

FIGURE 5 – Temps de la composition modulaire dans $\mathbb{F}_{2^{200}+235}$ en fonction de K

D'après la figure 5, pour $m = 2$ la constante optimale semble être 1.4. Pour $m = 3$, elle descend à 1.2. Sachant que l'algorithme a une complexité décroissante pour m croissant, choisir de poser K à 1.2 semble être un bon choix.

Chaque récursion demande au moins tous les nombres premiers jusqu'à L_{max}. De ce fait, pour chaque nombre premier inférieur à L_{max}, on a autant de FFT multidimensionnelle à faire. Sachant qu'une FFT multidimensionnelle demande mp^{m-1} FFT dans $\mathbb{F}_p$, l'évolution du nombre de FFT unidimensionnelle est polynomiale en p. Il est donc plus important d'avoir une FFT efficace pour des grands nombres premiers (de l'ordre de L_{max}) que pour des petits $(2, 3, 5, ...)$.

4.7 Transformée de Fourier rapide dans un corps premier

4.7.1 Implémentation

L'algorithme d'évaluation multipoint univariée est déjà implémenté sous `FLINT`. J'ai implémenté minutieusement toutes les FFT, et la méthode la plus rapide en pratique est la variante de la FFT de Bluestein (cf. section 3.3.5). On présente donc ici son implémentation.

L'algorithme a été implémenté en `C`, d'abord dans une version opérationnelle, puis dans une version optimisée. Il prend en entrée un polynôme A à évaluer, de degré au plus $p - 2$, et une racine primitive ω dans $\mathbb{F}_p$. Il renvoie la liste contenant $G(i)$ pour i allant de 1 à $p - 1$.

La première version de l'algorithme traite d'abord le cas $p = 2$ en calculant la somme des coefficients de A. Pour les autres valeurs de p, on déroule l'algorithme :

- Stocker dans un tableau `w_pow` les valeurs de ω^i pour i allant de 0 à $p - 2$.

- Calculer f_i pour i allant de 0 à $p - 2$ et f'_i pour i allant de 0 à $\frac{p-1}{2} - 1$. Pour cela on calcule ces puissances de ω : $i^2 \mod p - 1$ puis $(i^2 + i) \mod p - 1$, et on en déduit respectivement f_i et f'_i en recherchant ces puissances dans `w_pow`.

- Créer le polynôme F et calculer ses coefficients.

- Calculer les coefficients de G, c'est-à-dire les g_i pour i allant de 0 à $p - 2$, en calculant d'abord w^{-i^2} et w^{-i^2-i}. Ce calcul se fait en calculant les puissances $-i^2 \mod p - 1$ puis $-i^2 - i \mod p - 1$, et en recherchant ensuite les résultats dans `w_pow`. On finalise le calcul de chaque g_i en sommant modulo p les deux résultats précédents.

- Calculer H le produit de F par G (sans réduire modulo $X^{p-1} - 1$ pour le moment).

- Réduire modulo $X^{p-1} - 1$ en sommant la partie haute de H avec sa partie basse, la partie basse étant un polynôme de degré $p - 2$.

- Calculer $X(2i)$ et $X(2i + 1)$ pour i allant de 0 à $\frac{p-1}{2} - 1$. Pour cela on multiplie d'abord H par l'inverse de 2 $(= \frac{p+1}{2})$, puis on stocke la partie haute et la partie basse dans deux

polynômes différents H_h et H_l, la partie basse H_l donnant un polynôme de degré $\frac{p-1}{2} - 1$. Il reste à calculer $f_i(H_l + \theta H_h)$ et $f_i'(H_l - \theta H_h)$.

- On obtient un tableau `A` contenant $A(w^i)$ à l'index i. Soit `B` le tableau qui contiendra les $A(i)$, réordonner `A` en écrivant `B[w_pow[i]] = A[i]` pour i allant de 0 à $p-2$.

La version optimisée comporte quelques améliorations :

☐ On utilise la symétrie de f_i venant du fait que $i^2 = (-i)^2 = (p-1-i)^2$, pour ne calculer (et ne stocker) les f_i que pour i allant jusqu'à $\frac{p-1}{2}$. Pour les autres valeurs de f_i, on écrit que $f_{\frac{p-1}{2}+i} = f_{\frac{p-1}{2}-i}$ pour i allant de 1 à $\frac{p-1}{2} - 1$. Ces autres valeurs ne servent qu'à calculer les coefficients de F.

☐ On utilise l'égalité $g_{i+\frac{p-1}{2}} = \sigma(\omega^{-i^2} - \omega^{-i^2-i})$. Sachant que calculer g_i demande de connaître w^{-i^2} et w^{-i^2-i}, on utilise ces données pour calculer à la fois g_i et $g_{i+\frac{p-1}{2}}$ pour i allant de 0 à $\frac{p-1}{2} - 1$. Le calcul des $g_{i+\frac{p-1}{2}}$ ne demande plus qu'une soustraction modulaire (la valeur de σ est gérée avec un branchement conditionnel).

☐ Les autres améliorations ne sont que très mineures, l'une consiste à calculer $(i+1)^2 \mod p-1$ comme $i^2 + 2 \times (i+1) - 1 \mod p-1$, sachant que l'on connait $i^2 \mod p-1$ et $i+1$. L'autre consiste à remplacer les fonctions modulaires d'addition, de soustraction et de calcul d'opposé, par un branchement conditionnel. Ces fonctions utilisant également un branchement plutôt qu'une réduction modulaire naïve, on en déduit que c'est l'appel de fonction qui fait perdre du temps. Il est également préférable d'éviter les réductions modulaires inutiles. Par exemple, pour i allant de 0 à $\frac{p-1}{2} - 1$, on démontre aisément que $2 \times (i+1) - 1$ n'a pas besoin d'être réduit modulo $p-1$, puis lui ajouter i^2 modulo $p-1$ demande juste de retirer $p-1$ si le résultat est supérieur ou égal à $p-1$.

4.7.2 Résultats

Les tests pour les FFT n'ont pas pris en compte le temps de calculer la racine primitive. En pratique, on peut considérer qu'elle est précalculée. Le temps de réduire modulo $X^{p-1} - 1$ chaque polynôme de degré $p-1$ est pris en compte dans les mesures.

Sur la figure 6, on constate un gain asymptotique de 90% environ entre l'évaluation multipoint univariée et la dernière version de la FFT de Bluestein. Ce gain atteint 60% vers $p = 73$, et pour les p plus petits on a en général 50% de gain. Il y a un gain global d'environ 20% entre l'ancienne version de la FFT et la nouvelle exploitant certaines propriétés décrites précédemment.

Au vu des résultats expérimentaux, et pour avoir le meilleur des deux mondes, l'algorithme de FFT appelle l'évaluation multipoint univariée si $p < 20$.

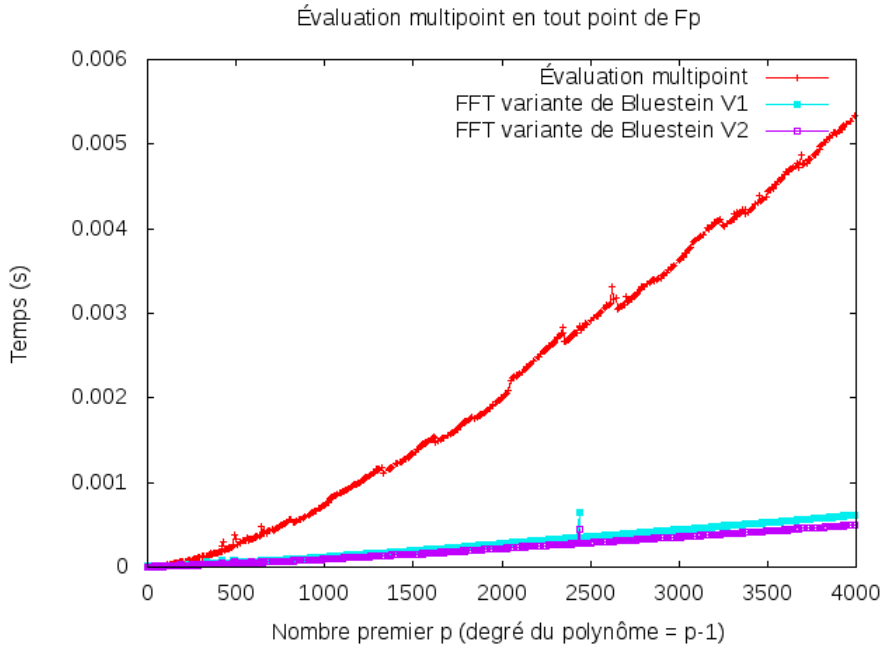

FIGURE 6 – Temps pour évaluer en tous les points de $\mathbb{F}_p$ en fonction de p

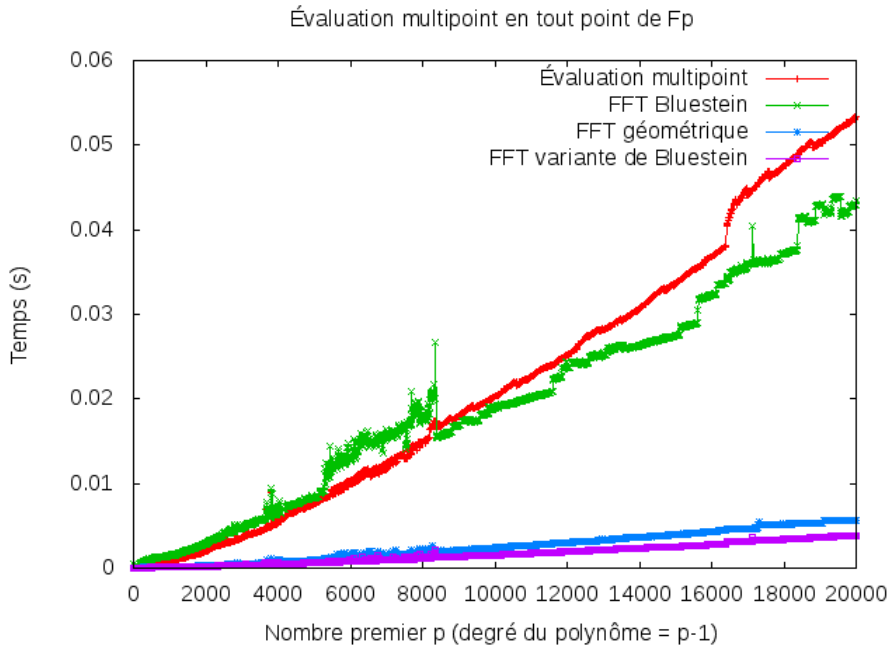

FIGURE 7 – Temps pour évaluer en tous les points de $\mathbb{F}_p$ en fonction de p

4.7.3 Comparaison de toutes les méthodes

On constate sur la figure 7 que toutes les méthodes battent asymptotiquement l'algorithme d'évaluation multipoint rapide. C'est parce que l'on se ramène à chaque fois à un nombre de

produit polynomial constant (1 ou 2). Cependant, le fait de passer dans une extension pour la méthode de Bluestein ne rend l'algorithme intéressant que pour une valeur de p trop grande (8000), ce qui n'apportera pas de gain dans l'algorithme 6. Les deux autres méthodes apportent un gain considérable même pour p petit, la variante de Bluestein s'avère légèrement plus rapide que la FFT géométrique.

Remarque. À l'origine, la FFT de Bluestein avait en pratique une complexité quadratique. Pour l'expliquer, il faut définir la multiplication haute de deux polynômes de degré n. C'est la multiplication qui ne calcule que les coefficients de degré supérieur à n du produit. De même la multiplication basse ne calcule que les coefficients de degré inférieur à n du produit. La fonction de multiplication haute de `FLINT` a une complexité quadratique, tandis que la fonction de multiplication basse a une complexité quasi-linéaire. En transformant le problème de multiplication haute en problème de multiplication basse (cf. annexe section A), on obtient l'algorithme actuel de complexité quasi-linéaire. Ce défaut d'implémentation dans `FLINT` a été pris en compte dans toutes les implémentations.

4.8 EMM par la FFT multidimensionnelle

On détaille ici l'imlémentation des trois étapes de l'algorithme.

4.8.1 Algorithme de réduction modulaire par $X_i^p = X_i$

On veut effectuer la réduction modulaire par $X_i^p = X_i$ pour les m variables. Pour simplifier l'algorithme, on suppose que la réduction des exposants modulo $p-1$ renvoie un entier compris dans $[\![1, p-1]\!]$.

L'idée basique est, pour chaque coefficient du polynôme, d'écrire ses coordonnées comme $\sum_{i=0}^{m-1} a_i d^i$, et de l'ajouter au coefficient situé à la coordonnée $\sum_{i=0}^{m-1}(a_i \mod (p-1))p^i$. Avec cette méthode, on doit pour chaque entier j de 0 à $d^m - 1$, l'écrire comme $\sum_{i=0}^{m-1} a_i d^i$ puis comme $\sum_{i=0}^{m-1}(a_i \mod (p-1))p^i$.

Voici le pseudocode associé, on suppose que les coefficients de *f_mod* sont initialisés à 0.

Algorithme 9 Version 1 de l'algorithme de réduction modulaire par $X_i^p = X_i$

```
fonction MODULOFERMAT(f_mod,f)
    A[m]
    pour j = 0 → d^m − 1 faire
        A = Décomposition en base d de j                ▷ Calcul des a_i
        A = A mod (p-1)                                 ▷ Calcul des a_i mod (p − 1)
        f_mod->coeffs[A] = (f_mod->coeffs[A] + f->coeffs[j]) mod p
    fin pour
fin fonction
```

Remarque. L'accès en mémoire à `f_mod->coeffs[A]` demande de calculer $\sum_{i=0}^{m-1} A[i]p^i$. Cela est fait de manière naïve, mais en stockant les p^i dans un tableau avant l'appel à `ModuloFermat`, afin de ne les calculer qu'une seule fois. Ce calcul force la complexité à être au moins égale à $O(d^m m)$.

On peut éviter le calcul des a_i. Pour cela, on initialise un tableau de m cases à 0 qui contiendra les a_i pour le j courant, et à chaque incrémentation de j, on incrémente la première case du tableau de 1. Si la valeur de la case arrive à d, comme pour une propagation de retenue, on met la case à 0 et on incrémente la case suivante, qui appliquera ce procédé récursivement.

Algorithme 10 Version 2 de l'algorithme de réduction modulaire par $X_i^p = X_i$

```
fonction MODULOFERMAT2(f_mod,f)
    A[m] = [0, ... , 0]
    pour j = 0 → d^m − 1 faire
        A[0]++
        k = 0
        tant que ( (A[k] == d) && (k != (m-1)) ) faire        ▷ Gestion de la retenue
            A[k] = 0
            k++
            A[k]++
        fin tant que
        A = A mod (p-1)
        f_mod->coeffs[A] = (f_mod->coeffs[A] + f->coeffs[j]) mod p
    fin pour
fin fonction
```

On peut également éviter de calculer a_i modulo $p-1$ si on leur créé un tableau dédié, et si comme pour le tableau des a_i, on le met à jour à chaque incrémentation de j (en prenant en compte que si la case vaut $p-1$, elle devient 0).

4.8.2 FFT multidimensionnelle

L'algorithme de FFT multidimensionnelle a été implémenté comme une fonction récursive. Dans le cas $m = 1$, on lance une FFT unidimensionnelle. Dans le cas $m > 1$, la principale difficulté vient de la gestion de la mémoire.

En mémoire, récupérer les coefficients du polynôme en X_{m-1} se fait facilement puisqu'un coefficient est stocké toutes les p^{m-1} cases mémoires. On évalue chacun des p coefficients avec une récursion, et chaque résultat étant nécessairement de dimension $m-1$ (on a p^{m-1} évaluations), ils constituent naturellement les lignes (de dimension $m-1$) d'un tableau de dimension m. Pour récupérer les p^{m-1} polynômes en X_{m-1}, il faut récupérer chaque colonne (de dimension 1) du tableau. On évalue ensuite les p^{m-1} polynômes correspondants avec la FFT unidimensionnelle, et les résultats étant chacun de dimension 1, on écrase les polynômes en X_{m-1} par leur vecteur résultat correspondant. Cette façon de procéder permet de n'avoir besoin que d'un seul tableau de dimension p^m, d'un tableau de taille p pour convertir une colonne de coefficient en ligne de coefficient (afin d'avoir un polynôme univarié) et d'un second pour stocker le résultat de la FFT avant de le transformer en colonne dans le tableau final.

Le schéma de l'exemple 8 suit la méthode décrite précédemment, et se lit de gauche à droite.

Exemple 8. $p = 3$, $m = 2$,
$M(X_0, X_1) = (f_0 + f_1X_0 + f_2X_0^2) + (g_0 + g_1X_0 + g_2X_0^2)X_1 + (h_0 + h_1X_0 + h_2X_0^2)X_1^2$

$M(X_0, X_1)$:

	1	X_0	X_0^2
1	f_0	f_1	f_2
X_1	g_0	g_1	g_2
X_1^2	h_0	h_1	h_2

Polynômes en X_0 :

	1	X_0	X_0^2
1	$F(X_0)$		
X_1	$G(X_0)$		
X_1^2	$H(X_0)$		

Récursion sur F, G, H :

	$X_0 = 0$	$X_0 = 1$	$X_0 = 2$
1	$F(0)$	$F(1)$	$F(2)$
X_1	$G(0)$	$G(1)$	$G(2)$
X_1^2	$H(0)$	$H(1)$	$H(2)$

Polynômes en X_1 :

	$X_0 = 0$	$X_0 = 1$	$X_0 = 2$
1 X_1 X_1^2	$P(X_1)$	$Q(X_1)$	$R(X_1)$

FFT sur P, Q, R :

	$X_0 = 0$	$X_0 = 1$	$X_0 = 2$
$X_1 = 0$	$P(0)$	$Q(0)$	$R(0)$
$X_1 = 1$	$P(1)$	$Q(1)$	$R(1)$
$X_1 = 2$	$P(2)$	$Q(2)$	$R(2)$

4.8.3 Recherche d'un élément dans la table de taille $\mathbb{F}_p^m$

Les α_i étant stockés sous forme d'éléments de $\mathbb{F}_p^m$, il suffit de faire le produit scalaire d'un α_i avec le vecteur des p^i pour obtenir l'index dans le tableau résultat de la FFT multidimensionnelle.

Sur l'exemple précédent, si on a $\alpha_0 = (1, 2)$, on cherche donc l'évaluation en $X_0 = 1$ et $X_1 = 2$. La coordonnée correspondante est $1 \times 3^0 + 2 \times 3^1 = 7$, ce qui correspond bien à $Q(2)$.

4.9 Multimodulaire pour les extensions de corps

Implémentation

L'implémentation de l'algorithme 3 demande de savoir qu'un élément de type `fq_t` est défini comme un élément de type `fmpz_mod_poly_t`. Il demande également de gérer quelques conversions de type.

1. Cette étape est inutile en pratique.
2. Pour les polynômes f de type `fq_multi_poly_t`, la substitution de Kronecker demande de déclarer $\bar{f}$ comme un polynôme de type `fmpz_mod_multi_poly_t`, et de calculer ses coefficients en évaluant chaque coefficient de type `fq_t` de f en M avec la fonction `fmpz_poly_evaluate_fmpz`. La conversion de `fmpz_poly` à `fmpz_mod_poly` du résultat est faite automatiquement.
   Pour les polynômes f de type `fq_nmod_multi_poly_t`, il n'est pas possible d'évaluer les coefficients de f en M avec la fonction `nmod_poly_evaluate_nmod`, car la fonction n'évalue qu'en des entiers 64 bits. Cela oblige de convertir les éléments de type `fq_nmod_t` en `fq_t`. De ce fait, à moins d'implémenter une nouvelle fonction d'évaluation, la méthode la plus logique est de faire les conversions de petite caractéristique à grande caractéristique, puis d'appeler l'algorithme en grande caractéristique.

3. On appelle la fonction `fmpz_mod_multi_poly_multimodular` que j'ai implémentée.
4. Il n'y a pas de fonction pour lire les bits de $\bar{f}(\bar{\alpha}_i)$ par paquet. On utilise alors des divisions euclidiennes successives par M pour retrouver les coefficients de $\widetilde{f}(\widetilde{\alpha}_i)$, comme on le ferait pour obtenir les chiffres en base M d'un entier.

5 Conclusion

Étudions les résultats obtenus avec mon implémentation de la composition modulaire. Les tests ont été effectués avec p de l'ordre de 200 bits.

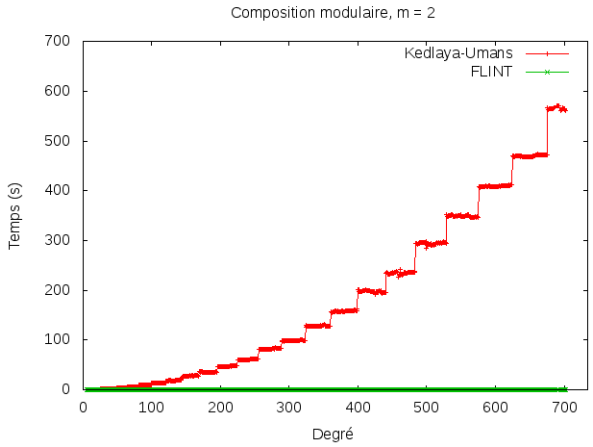

FIGURE 8 – Composition modulaire, $m = 2$

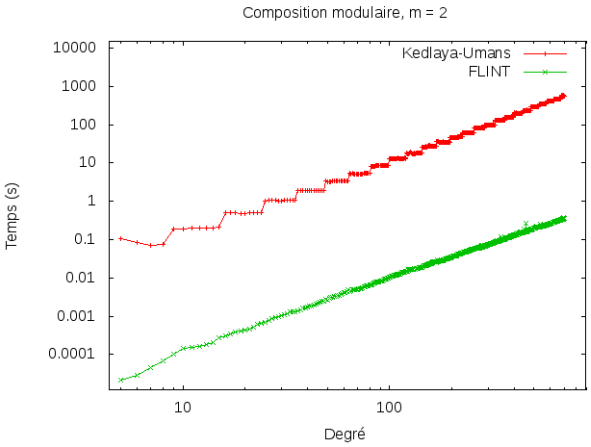

FIGURE 9 – Échelle logarithmique

Les résultats obtenus pour $m = 2$ montre que la composition modulaire de Kedlaya-Umans a une complexité quadratique, comme Brent et Kung pour des petits degrés. Ce résultat est attendu au vu du facteur $n^{1+\frac{2}{m}}$ dans la complexité (3.4). Les sauts de la courbe de la figure 8 se produisent quand $d^m = n$, donc quand n est un carré.

Les résultats obtenus pour $m = 3$ (cf. figure 11) semblent montrer qu'en extrapolant, l'algorithme de Kedlaya-Umans finira par battre celui de Brent et Kung. Cette extrapolation ne prendrait pas en compte l'aspect sous-quadratique de l'algorithme de Brent et Kung, dont l'implémentation par `FLINT` a une complexité asymptotique d'au mieux $\widetilde{O}(n^{1,67})$ en le degré. Sachant qu'on a une complexité attendue quasiment identique ($\widetilde{O}(n^{1+\frac{2}{3}})$) pour l'algorithme de Kedlaya-Umans, l'extrapolation devrait donner des courbes parallèles. On constate également qu'en extrapolant, l'algorithme de composition modulaire pour $m = 3$ finira par battre celui pour $m = 2$.

Les résultats obtenus pour $m = 4$ s'arrête à $n = 256$. À partir de $n = 257$, le choix de paramètre passe de $d = 4$ à $d = 5$ et l'algorithme nécessite plus de 16 `Go`, ce qui dépasse la capacité de la RAM. Cela s'explique par le fait que L_{max} dépend de d et de m, et que l'algorithme 5 nécessite une table de taille $(K \times L_{max})^m$, $K \geq 1$. Pour $K = 1$ et $L_{max} = 359$, 359^4 est déjà de l'ordre de 16 `Go`, ce qui pose une contrainte assez forte sur le choix de m. En choisissant $K = 1.2$, $K^4 \simeq 2$, donc la valeur maximale de L_{max} descend à 179.

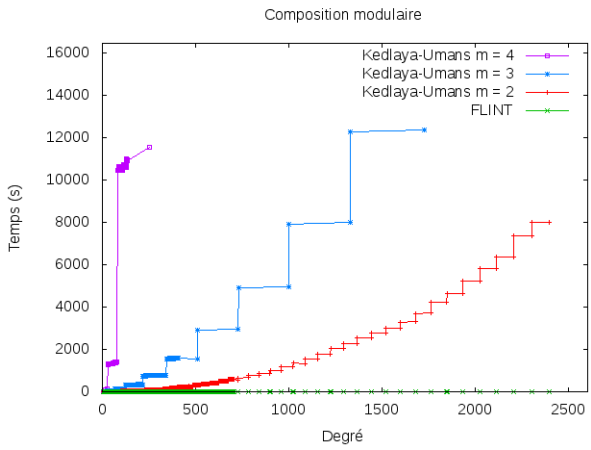

FIGURE 10 – Composition modulaire

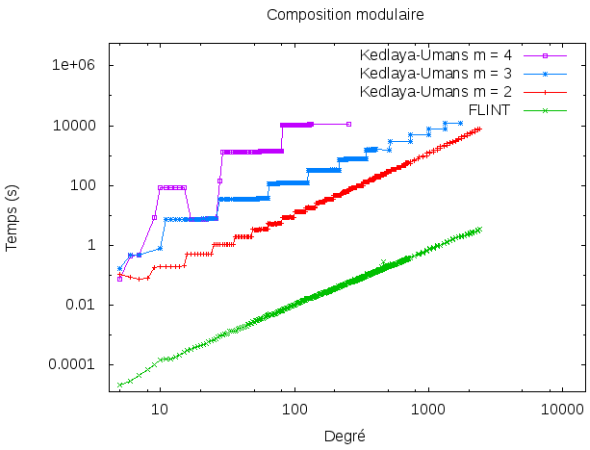

FIGURE 11 – Échelle logarithmique

On constate une intersection des courbes $m = 3$ et $m = 4$ pour un degré n allant de 17 à 27. Ceci est dû au fait que pour d passant de 2 à 3, il est inutile de prendre $m = 4$ pour la partie où $n \leq d^3$. Ce cas n'arrive pas asymptotiquement, et pour bien extrapoler la courbe $m = 4$, il faut considérer que le saut qui est fait à $n = 28$ est fait à $n = 17$, ce qui implique que la courbe forme des paliers de plus en plus petits.

Le choix de m permet de décider de la complexité asymptotique de l'algorithme. La meilleure complexité asymptotique est donc obtenue en choisissant m maximal. Mais en pratique, on est limité par la capacité mémoire de la RAM. Il faut donc choisir la plus grande valeur de m qui ne dépasse pas cette capacité mémoire.

Remerciements

Je tiens à remercier mes encadrants Jean-Pierre FLORI et Jérome PLÛT pour m'avoir permis d'effectuer ce stage très constructif, ainsi que pour le temps qu'ils m'ont patiemment consacré.

Références

[BK78] R. P. BRENT et H. T. KUNG : *Fast algorithms for manipulating formal power series*. Journal of the Association for Computing Machinery, Vol. 25, No. 4, pp. 581–595, 1978.

[BLS03] A. BOSTAN, G. LECERF et E. SCHOST : *Tellegen's principle into practice*. In ISSAC '03 : Proceedings of the 2003 International Symposium on Symbolic and Algebraic Computation, J. R. Sendra, ed., ACM, New York, pp. 37–44, 2003.

[Bos07] A. BOSTAN : *Calculs modulaires, évaluation et interpolation*. Cours du MPRI, 2007. `http://www.ens-lyon.fr/denif/data/algos_calcul_formels_mpri/2007/cours/Cours5.pdf`.

[FHHJ16] C. FIEKER, W. HART, T. HOFMANN et F. JOHANSSON : Nemo, 2016. Version 0.5, `http://nemocas.org`.

[Har09] D. HARVEY : *Faster polynomial multiplication via multipoint Kronecker substitution*. Conférence, 2009. `http://web.maths.unsw.edu.au/~davidharvey/talks/kronecker-talk.pdf`.

[HJP13] W. HART, F. JOHANSSON et S. PANCRATZ : FLINT : Fast Library for Number Theory, 2013. Version 2.5.2, `http://flintlib.org`.

[HP98] X. HUANG et V. Y. PAN : *Fast rectangular matrix multiplication and applications*. Journal of complexity 14, pp. 257–299, 1998.

[HVDHL14] D. HARVEY, J. VAN DER HOEVEN et G. LECERF : *Faster polynomial multiplication over finite fields*. 2014. `https://hal.archives-ouvertes.fr/hal-01022757`.

[KS98] E. KALTOFEN et V. SHOUP : *Subquadratic-time factoring of polynomials over finite fields*. Mathematics of Computation, 67 (223) : pages 1179–1197, 1998.

[KU11] K. S. KEDLAYA et C. UMANS : *Fast polynomial factorization and modular composition*. SIAM Journal on Computing, Vol. 40, No. 6, pp. 1767–1802, 2011. `http://hdl.handle.net/1721.1/71792`.

[Ost54] A. M. OSTROWSKI : *On two problems in abstract algebra connected with Horner's rule*. Studies in Mathematics and Mechanics presented to Richard von Mises, pages 40–48, Academic Press San Diego, 1954.

[Pot14] A. POTEAUX : *Évaluation et Interpolation rapide*. Cours de l'université de Lille 1, 2014. `http://www.lifl.fr/~poteaux/fichiers/gdt-eval-interpol.pdf`.

[Rab80] M. O. RABIN : *Probabilistic algorithms in finite fields*. SIAM Journal on Computing, vol. 9, no 2, p. 273–280, 1980.

[Sho99] V. SHOUP : *Efficient computation of minimal polynomials in algebraic extensions of finite fields*. ISSAC, pages 53–58, 1999.

[Smi07a] J. O. SMITH : *Bluestein's FFT algorithm*. Mathematics of the discrete fourier transform (DFT) with audio applications, 2007. `https://www.dsprelated.com/freebooks/mdft/Bluestein_s_FFT_Algorithm.html`.

[Smi07b] J. O. SMITH : *Rader's FFT algorithm*. Mathematics of the discrete fourier transform (DFT) with audio applications, 2007.

[Uma08] C. UMANS : *Fast polynomial factorization and modular composition in small characteristic*. In Richard E. Ladner and Cynthia Dwork, editors, STOC, pages 481–490. ACM, 2008. `http://users.cms.caltech.edu/~umans/papers/U07.pdf`.

[VZGG99] J. VON ZUR GATHEN et J. GERHARD : *Modern computer algebra*. Cambridge University Press, 1999.

[Wil12] V. V. WILLIAMS : *Multiplying matrices faster than Coppersmith-Winograd*. In Howard J. Karlo and Toniann Pitassi, editors, STOC, pages 887–898. ACM, 2012.

Appendices

A FFT de Bluestein avec deux produits de polynômes

On rappelle les définitions de a_n et b_n :
$a_n = x_n \omega^{-\frac{n^2}{2}}$ qui est défini pour n allant de 0 à $N-1$,
$b_n = \omega^{\frac{n^2}{2}}$ qui est défini pour n allant de $-(N-1)$ à $N-1$.

On définit la multiplication haute de deux polynômes de degré $N-1$ comme la multiplication qui ne calcule que les coefficients de degré supérieur à $N-1$ du produit. De même la multiplication basse ne calcule que les coefficients de degré inférieur à $N-1$ du produit.

Posons $A(X) = \sum_{n=0}^{N-1} a_n X^n$ et $B(X) = \sum_{n=0}^{N-1} b_n X^n$.
On souhaite réécrire $\sum_{n=0}^{N-1} a_n b_{k-n}$ comme produits de polynômes. On a :

$$\sum_{n=0}^{N-1} a_n b_{k-n} = \sum_{n=0}^{k} a_n b_{k-n} + \sum_{n=k+1}^{N-1} a_n b_{k-n}.$$

Cette formule est vraie pour $k \neq N-1$ dans la seconde somme (dans ce cas là, on pose que cette somme vaut 0).
On reconnaît en la première somme les coefficients d'un produit de polynôme $A(X) \times B(X)$ (multiplication basse car k vaut au plus $N-1$). On s'intéresse donc à transformer la seconde somme en produit de polynôme :

$$\begin{aligned}
\sum_{n=k+1}^{N-1} a_n b_{k-n} &= \sum_{n=N-k'}^{N-1} a_n b_{N-1-k'-n} \text{ en posant } k+k' = N-1, \\
&= \sum_{n=0}^{k'-1} a_{N-1-n} b_{n-k'} \text{ avec la réindexation } n \text{ devient } N-1-n, \\
&= \sum_{n=0}^{k'-1} a_{N-1-n} b_{k'-n} \text{ car pour tout } i \in \mathbb{N},\ b_{-i} = b_i, \\
&= \sum_{n=0}^{k'-1} a_n^* b_{k'-n} \text{ avec } a_n^* \text{ le } n\text{-ème coefficient du polynôme aux inverses de } A(X).
\end{aligned}$$

Le polynôme aux inverses est défini par $A(X) = \sum_{n=0}^{N-1} a_{N-1-n} X^n$. Cela revient à lire les coefficient de droite à gauche plutôt que de gauche à droite.

On voit apparaître dans cette somme le produit polynomial $A^*(X)$ par $B(X)$ (multiplication basse ici aussi car k' vaut au plus $N-1$).

Au final, en posant ab_i le coefficient du terme de degré i de $A(X)B(X)$, et c_i pour celui de $A^*(X)B(X)$, on obtient donc :

$$\sum_{n=0}^{N-1} a_n b_{k-n} = ab_k + c_k^* \text{ en posant } c_{N-1}^* = 0.$$

En écrivant la formule de multiplication haute comme $\sum_{k=0}^{N-1}(\sum_{n=0}^{k} a_{N-1-n}b_{N-1+n-k})X^{N-1+(N-1-k)}$, on peut également transformer la seconde somme comme une multiplication haute :

$$\sum_{n=k'+1}^{N-1} a_n b_{k'-n} = \sum_{n=0}^{k'-1} a_{N-1-n} b^*_{N-1+n-k'}.$$

Dans ce cas là, en posant d_i le coefficient du terme de degré i de $\frac{1}{X^{N-1}}A(X)B^*(X)$, la somme finale s'écrit comme :

$$\sum_{n=0}^{N-1} a_n b_{k-n} = ab_k + d_k \text{ avec } d_{N-1} = 0.$$

En conclusion, la méthode vue dans la section 3.3.4 pour calculer la somme demande une multiplication haute de polynôme de degré $2N-1$, tandis que la seconde demande deux multiplications basses de degré $N-1$. La méthode a été implémentée avec les deux multiplications basses.