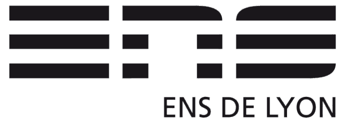

RAPPORT DE STAGE DE L3

Test rapide de cubicité modulaire

Élève :
Alice PELLET-MARY

Encadrant :
Pierrick GAUDRY

Du 3 juin au 19 juillet 2013

Table des matières

1 Carrés et cubes dans $\mathbb{Z}/p\mathbb{Z}$ 2

1.1 Étude des carrés . . . 3

1.2 Étude des cubes . . . 4

1.2.1 Propriétés de $\mathbb{Z}[\zeta]$ . . . 4

1.2.2 Passer de $\mathbb{F}_p$ à $\mathbb{Z}[\zeta]$ . . . 5

1.2.3 Déterminer les cubes modulo π dans $\mathbb{Z}[\zeta]$ . . . 5

1.3 Analogies et différences entre les cubes et les carrés . . . 6

2 Algorithme en $O(M(n)\log n)$ pour le symbole de résiduosité cubique 7

2.1 Algorithme quasi-quadratique pour le calcul de $\left(\frac{b}{a}\right)_3$ . . . 7

2.1.1 Division par les poids faibles . . . 8

2.1.2 L'algorithme quasi-quadratique . . . 9

2.2 Algorithme en $O(M(n)\log n)$ pour le calcul de $\left(\frac{b}{a}\right)_3$ . . . 11

2.2.1 Demi Algorithme . . . 11

2.2.2 Algorithme quasi-linéaire . . . 14

2.2.3 Complexité . . . 14

3 Implémentation 15

3.1 Améliorations de DemiAlgo . . . 15

3.1.1 Première amélioration . . . 15

3.1.2 Deuxième amélioration . . . 16

3.1.3 Troisième amélioration . . . 16

3.2 Vérification de l'exactitude des résultats . . . 16

3.3 Résultats Expérimentaux . . . 17

4 Puissances quatrièmes 17

4.1 Analogie avec les cubes . . . 17

4.2 Différences . . . 18

4.3 Résultats expérimentaux . . . 19

5 Conclusion 20

Introduction

Le symbole de Jacobi est un outil mathématique très utilisé pour les tests de primalité, notamment pour le test de primalité de Solovay-Strassen ou celui de Miller-Rabin. Il a été très étudié et il existe actuellement plusieurs algorithmes quasi-linéaires permettant de le calculer. Il permet entre autre de déterminer si un nombre est un carré dans $\mathbb{Z}/p\mathbb{Z}$ avec p premier.
L'objectif de ce stage était d'adapter un algorithme rapide de calcul du symbole de Jacobi (l'algorithme de [1]) pour déterminer si un nombre est un cube dans $\mathbb{Z}/p\mathbb{Z}$. On sait calculer la racine cubique d'un nombre dans $\mathbb{Z}/p\mathbb{Z}$ en temps polynomial, mais l'objectif ici était juste de savoir si on nombre est un cube, pas de trouver sa racine cubique, ce qu'on espère pouvoir faire plus rapidement (en temps quasi-linéaire). Pour cela on utilise un analogue du symbole de Jacobi appelé symbole de résiduosité cubique. Il existe déjà des algorithmes quadratique pour calculer le symbole de résiduosité cubique (par exemple celui de [2]), mais l'objectif était d'utiliser l'algorithme asymptotiquement rapide de [1] (qui calcule le symbole de Jacobi) pour obtenir un algorithme quasi-linéaire permettant de calculer le symbole de résiduosité cubique.

Dans tout le rapport, on s'intéressera à la complexité des algorithmes exprimée en nombre d'opérations sur les bits, et la taille des entrées sera leur nombre de bits. Par exemple, si l'algorithme prend en entrée un entier a, on dira qu'il est linéaire s'il est en $O(\log a)$. On notera souvent n la taille des entrées (ici $n = \log(a)$). La notion d'opérations sur les bits est difficile à définir. Ici, on se placera dans le modèle d'une machine de Turing à plusieurs rubans, et on considère, pour les opérations de base, les complexités suivantes :

Complexité des opérations de base : Si on prend deux éléments a et b de taille n, l'addition et la soustraction de a et b se font en $O(n)$ tandis que la multiplication et la division se font en $O(M(n))$, où $M(n) = n \log n \log \log n$, en utilisant un algorithme de multiplication asymptotiquement rapide, comme celui de Schönhage-Strassen dans [6]. Il existe un algorithme récent dû à Fürer légèrement plus rapide que celui de Schönage et Strassen (cf [7]), mais ce n'est pas celui qui sera utilisé ici.
Lorsqu'on parle d'algorithme quasi-linéaire c'est un algorithme en $O(n(\log n)^k)$ pour un certain k. Par exemple, la multiplication de deux entiers se fait en temps quasi-linéaire.

1 Carrés et cubes dans $\mathbb{Z}/p\mathbb{Z}$

Malgré quelques petites différences, il y a beaucoup de ressemblances entre l'étude des carrés et celle des cubes dans $\mathbb{Z}/p\mathbb{Z}$ (avec p premier). Nous verrons donc dans un premier temps des éléments clés et quelques idées pour tester si un nombre est un carré dans $\mathbb{Z}/p\mathbb{Z}$, puis nous verrons les bases de l'étude des cubes, avant de détailler les différences et les ressemblances entre les carrés et les cubes.

1.1 Étude des carrés

Dans un corps fini à p éléments ($p > 2$), on sait qu'exactement $\frac{p+1}{2}$ des éléments sont des carrés, ce qui justifie l'existence de tests pour déterminer si un élément est un carré. On verra que pour les cubes, selon la valeur de p, soit tous les éléments sont des cubes modulo p, soit un tiers seulement des éléments sont des cubes (lorsque tous les éléments sont des cubes, on n'a pas besoin de test). On ne cherche ici qu'à savoir si l'élément est un carré, on ne veut pas déterminer la valeur de sa racine carrée.

L'étude des carrés dans un corps fini à p éléments utilise le symbole de Jacobi $\left(\frac{a}{b}\right)$. Par définition, si p est un nombre premier impair,

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & \text{si} & p \mid a \\ 1 & \text{si} & p \nmid a \text{ et } \exists x : a \equiv x^2 \mod (p) \\ -1 & \text{sinon} & \end{cases}$$

et si $n = p_1 \cdots p_k$ (les p_i premiers impairs, pas forcément distincts), on définit

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right) \cdots \left(\frac{a}{p_k}\right)$$

Le symbole de Jacobi (aussi appelé symbole de Legendre quand on l'utilise pour p premier) répond donc parfaitement à la question posée : déterminer si a est un carré modulo p premier. Il s'agit maintenant de réussir à le calculer rapidement. Heureusement, le symbole de Jacobi vérifie de nombreuses propriétés qui vont permettre d'avoir des algorithmes rapides.

Propriétés. Soient a, b, c, d des entiers, avec c et d impairs.

1. Si p est un entier premier impair, $\left(\frac{a}{p}\right) \equiv a^{\frac{p-1}{2}} \mod p$
2. $\left(\frac{a}{c}\right) \equiv \left(\frac{b}{c}\right)$ si $a = b \mod c$
3. $\left(\frac{ab}{c}\right) = \left(\frac{a}{c}\right)\left(\frac{b}{c}\right)$
4. $\left(\frac{a}{cd}\right) = \left(\frac{a}{c}\right)\left(\frac{a}{d}\right)$
5. si a et b sont des entiers impairs, $\left(\frac{b}{a}\right) = (-1)^{\frac{a-1}{2}\frac{b-1}{2}}\left(\frac{a}{b}\right)$
   (loi de réciprocité quadratique)

La première propriété donne une première méthode pour calculer le symbole de Legendre : on calcule par exponentiation rapide $a^{\frac{p-1}{2}} \mod p$. On réalise $O(\log \frac{p-1}{2})$ multiplications d'entiers de taille inférieure à n, d'où une complexité dans le pire cas en $O(nM(n))$. L'inconvénient de cet algorithme est qu'il ne s'applique que pour des entiers p premiers, et il faut donc hypothétiquement l'utiliser $O(n)$ fois (une fois pour chaque facteur premier de b) si on veut calculer le symbole de Jacobi d'un entier b non premier (n étant la taille de b).
Les propriétés 2 à 5 permettent cependant d'obtenir des algorithmes plus rapides (les algorithmes les plus rapides actuellement sont en $O(M(n)\log n)$) et qui fonctionnent pour n'importe quel couple de nombres (a, b) avec b impair. Nous verrons plus loin un de ces algorithmes adapté au cas des cubes.

1.2 Étude des cubes

La première différence entre les carrés et les cubes est que si $p \equiv 2 \mod 3$, tous les éléments de $\mathbb{Z}/p\mathbb{Z}$ sont des cubes (la fonction $f : x \mapsto x^3$ est une bijection sur $\mathbb{F}_p$). En revanche, si $p \equiv 1 \mod 3$, on peut montrer qu'exactement $\frac{p-1}{3}$ éléments de $\mathbb{F}_p^*$ sont des cubes. On n'a donc besoin de tests que dans le cas où $p \equiv 1 \mod 3$.

Si $p \equiv 1 \mod 3$ est premier, on a, comme dans le cas des carrés

$$a \text{ est un cube modulo } p \Longleftrightarrow a^{\frac{p-1}{3}} \equiv 1 \mod p.$$

Cette équivalence nous donne encore un algorithme en $O(nM(n))$ pour tester si un entier a est un cube modulo p (a et p de taille n). Ici, on observe quand même une différence avec les carrés : même si on définit un symbole pour les cubes comme le symbole de Jacobi, ce symbole ne vérifiera pas les propriétés 2 à 5 qui permettent de le calculer rapidement. Pour résoudre ce problème, on va considérer le corps $\mathbb{Q}(\zeta)$, où ζ est une racine cubique primitive de l'unité ($\zeta^2 + \zeta + 1 = 0$) et plus précisément son anneau des entiers (appelé anneau des entiers d'Eisenstein) $\mathbb{Z}[\zeta] = \{a + b\zeta, a \in \mathbb{Z}, b \in \mathbb{Z}\}$. Cet anneau est euclidien, on peut donc faire de l'arithmétique dedans en utilisant l'algorithme d'Euclide. Et on va voir qu'on peut définir dans cet anneau un analogue au symbole de Jacobi pour les cubes, qui vérifiera des propriétés similaires.

1.2.1 Propriétés de $\mathbb{Z}[\zeta]$

On va voir dans ce paragraphe une liste de définitions et de propriétés des entiers d'Eisenstein qui seront utiles dans la suite. Les propriétés ne sont pas démontrées ici mais on peut trouver une partie des preuves dans l'article de Bosma [3] ainsi que dans le livre de Ireland et Rosen [8].

Soit $z = a + b\zeta$, avec $a, b \in \mathbb{Z}$ un élément de $\mathbb{Z}[\zeta]$. On définit le conjugué de z par $\bar{z} = a + b\zeta^2 = a – b – b\zeta$ ainsi que sa norme par $N(z) = z\bar{z} = a^2 + b^2 – ab = \frac{a^2+b^2+(a-b)^2}{2}$. La norme est multiplicative ($N(ab) = N(a)N(b)$), et l'anneau est euclidien pour N : si $a, b \in \mathbb{Z}[\zeta]$, b non nul, il existe $q, r \in \mathbb{Z}[\zeta]$ tels que $a = bq + r$ et $N(r) < N(b)$. De plus, cette division peut se faire en temps quasi-linéaire, comme dans le cas des entiers relatifs.

Les unités de $\mathbb{Z}[\zeta]$ sont les 6 éléments $\{\pm 1, \pm\zeta, \pm\zeta^2\}$. Ce sont les seuls éléments inversibles dans l'anneau, et les seuls éléments de norme 1. On dit que a est associé à b s'il existe une unité u telle que $a = ub$.

L'élément $1 - \zeta$ joue un rôle important dans la suite. C'est un élément premier de l'anneau, de norme 3.

Définition et propriétés. *Éléments primaires.*
On dit que $a \in \mathbb{Z}[\zeta]$ est primaire si $a \equiv 1 \mod 3$. Dans ce cas, $a = 1 + 3(m + n\zeta)$ pour un certain couple d'entiers (m, n). Si $1 - \zeta$ ne divise pas a, alors il existe une unique unité u telle que ua soit primaire. Dans ce cas, on a donc $N(a) \equiv 1 \mod 3$.

Si $a \in \mathbb{Z}[\zeta]$, on a donc soit $N(a) \equiv 0 \mod 3$ (et dans ce cas $(1-\zeta) \mid a$), soit $N(a) \equiv 1 \mod 3$ (et dans ce cas $(1-\zeta) \nmid a$ et a est associé à un nombre primaire).

L'élément $1-\zeta$ est en fait l'analogue de 2 dans les entiers : si on considère deux éléments primaires de $\mathbb{Z}[\zeta]$, leur différence est alors un multiple de $3 = -\zeta^2(1-\zeta)^2$. Donc si deux éléments a et b ne sont pas divisibles pas $1-\zeta$, en choisissant a_1 et b_1 des éléments primaires associés à a et b, leur différence est un multiple de $1-\zeta$ (et même de $(1-\zeta)^2$). Cette propriété permet de calculer un PGCD binaire dans $\mathbb{Z}[\zeta]$, de la même manière qu'on le ferait dans $\mathbb{Z}$ avec 2 :

$$\text{pgcd}(a,b) = \begin{cases} (1-\zeta)\text{pgcd}(a',b') & \text{si} \quad a=(1-\zeta)a',\ b=(1-\zeta)b' \\ \text{pgcd}(a',b) & \text{si} \quad a=(1-\zeta)a',\ (1-\zeta)\nmid b \\ \text{pgcd}(a,b') & \text{si} \quad b=(1-\zeta)b',\ (1-\zeta)\nmid a \\ \text{pgcd}(a,a_1-b_1) & \text{sinon} \quad (a_1 \text{ et } b_1 \text{ primitifs, associés à a et b}) \end{cases}$$

En pratique, pour que l'algorithme termine il faut faire un peu plus attention (remplacer a par $a_1 - b_1$ au lieu de remplacer b si $N(a) > N(b)$), mais l'idée est la même que pour le PGCD binaire.

1.2.2 Passer de $\mathbb{F}_p$ à $\mathbb{Z}[\zeta]$

Soit p un nombre premier congru à 1 modulo 3. On factorise p dans $\mathbb{Z}[\zeta]$ en $p = \pi\bar{\pi}$, avec π premier (p est premier dans $\mathbb{Z}$ mais pas dans $\mathbb{Z}[\zeta]$). On peut trouver un tel π en calculant r tel que $r^2 + r + 1 \equiv 0 \mod p$, puis en prenant $\pi = \text{pgcd}(p, r-\zeta)$ (dans $\mathbb{Z}[\zeta]$). Le calcul de π peut prendre du temps (plus que le $O(M(n)\log n)$ de l'algorithme que l'on veut obtenir), mais une fois calculé il peut être réutilisé pour tous les entiers a tels qu'on cherche si a est un cube modulo p. On peut donc considérer le calcul de π comme un pré-traitement.

Proposition 1. *Soient $p \in \mathbb{Z}$ premier impair et $\pi \in \mathbb{Z}[\zeta]$ tel que $p = \pi\bar{\pi}$, alors $a \in \mathbb{Z}$ est un cube modulo p dans $\mathbb{Z}$ si et seulement si a est un cube modulo π dans $\mathbb{Z}[\zeta]$.*

1.2.3 Déterminer les cubes modulo π dans $\mathbb{Z}[\zeta]$

Dans $\mathbb{Z}[\zeta]$, on définit un symbole analogue au symbole de Jacobi (appelé symbole de résiduosité cubique).

Définition 1. *Symbole de résiduosité cubique.*

$$\left(\frac{\cdot}{\cdot}\right)_3 : \mathbb{Z}[\zeta] \times (\mathbb{Z}[\zeta] - (1-\zeta)\mathbb{Z}[\zeta]) \mapsto \{0, 1, \zeta, \zeta^2\}.$$

Soient a,$\pi \in \mathbb{Z}[\zeta]$, avec π premier, non divisible par $1-\zeta$, on définit

$$\left(\frac{a}{\pi}\right)_3 \equiv a^{(N(\pi)-1)/3} \mod \pi.$$

Si $b = \pi_1 \cdots \pi_k$ (les π_i premiers),

$$\left(\frac{a}{b}\right)_3 = \left(\frac{a}{\pi_1}\right)_3 \cdots \left(\frac{a}{\pi_k}\right)_3 .$$

Ce symbole possède des propriétés analogues au symbole de Jacobi.

Propriétés. Soient $a, b, c, d \in \mathbb{Z}[\zeta]$ tels que c et d ne sont pas des multiples de $1 - \zeta$.

1. Si $\pi \in \mathbb{Z}[\zeta]$ est premier, $\pi \neq 1 - \zeta$ et a n'est pas un multiple de π, $\left(\frac{a}{\pi}\right)_3 = 1$ si et seulement si a est un cube modulo π.
2. Si $\pi \in \mathbb{Z}[\zeta]$ est premier non divisible pas $(1 - \zeta)$, $\left(\frac{a}{\pi}\right)_3 \equiv a^{\frac{N(\pi)-1}{3}} \mod \pi$
3. $\left(\frac{a}{c}\right)_3 = \left(\frac{b}{c}\right)_3$ si $a \equiv b \mod c$
4. $\left(\frac{ab}{c}\right)_3 = \left(\frac{a}{c}\right)_3 \left(\frac{b}{c}\right)_3$ (si $(1 - \zeta) \nmid c$)
5. $\left(\frac{a}{cd}\right)_3 = \left(\frac{a}{c}\right)_3 \left(\frac{a}{d}\right)_3$ (si $(1 - \zeta) \nmid c, d$)
6. si a et b sont primaires, $\left(\frac{a}{b}\right)_3 = \left(\frac{b}{a}\right)_3$ (loi de réciprocité cubique)

Ici, si on veut échanger a et b non primaires, mais tels que $(1-\zeta) \nmid a, b$, il faut les transformer d'abord en éléments primaires, et ne pas oublier les facteurs que ces transformations vont ajouter.
On a aussi, si $b = 1 + 3(m + n\zeta)$ est primaire (avec m et n des entiers) :

7. $\left(\frac{1-\zeta}{b}\right)_3 = \zeta^m$
8. $\left(\frac{\zeta}{b}\right)_3 = \zeta^{-(m+n)}$
9. $\left(\frac{-1}{b}\right)_3 = 1$

Avec ces propriétés et les cas de base, on peut adapter l'algorithme de R.P. Brent et P. Zimmermann ([1]) qui permet de calculer le symbole de Jacobi en $O(M(n) \log n)$ pour obtenir un algorithme également en $O(M(n) \log n)$ permettant de calculer le symbole de résiduosité cubique.

1.3 Analogies et différences entre les cubes et les carrés

On vient de voir qu'il n'existe pas de symbole adapté dans $\mathbb{Z}$ pour étudier les cubes. En revanche, en passant dans $\mathbb{Z}[\zeta]$, on obtient un symbole très proche de celui de Jacobi. De plus, l'algorithme de [1] est basé sur la division binaire, qui peut être adaptée dans $\mathbb{Z}[\zeta]$ grâce à l'analogie entre $1 - \zeta$ et 2 vue précédemment. En revanche, on a quelques différences car on ne peut plus parler de plus petit reste positif pour fixer un représentant unique d'une classe d'équivalence (on ne peut pas définir $a \mod b$ en prenant le plus petit reste positif de la division de a par b car on n'a plus d'ordre sur $\mathbb{Z}[\zeta]$). On peut cependant s'en sortir quand même et définir mod de manière unique :
Si $a, b \in \mathbb{Z}[\zeta]$ avec b non nul, on définit q_1, q_2, e_1, e_2 tels que

$$\frac{a}{b} = \frac{a\bar{b}}{N(b)} = (q_1 + e_1) + (q_2 + e_2)\zeta, \text{ avec } q_1, q_2 \in \mathbb{Z} \text{ et } e_1, e_2 \in] - \frac{1}{2}, \frac{1}{2}].$$

On pose ensuite $q = (q_1 + q_2\zeta)$ et $r = b(e_1 + e_2\zeta)$. On a bien $r \in \mathbb{Z}[\zeta]$, et $N(r) = N(b)N(e_1 + e_2\zeta) \leq \frac{3}{4}N(b)$. De plus, si $a_1 \equiv a_2 \mod b$ et r_1 et r_2 sont les restes associés à a_1 et a_2 pour la division par b avec la méthode précédente, alors $r_1 = r_2$: on peut déterminer un représentant unique pour chaque classe d'équivalence modulo b.

Une autre différence entre $\mathbb{Z}$ et $\mathbb{Z}[\zeta]$ est qu'on ne peut pas toujours s'arranger pour que $N(a - b) \leq \max(N(a), N(b))$ (soustraire deux éléments ne donne pas forcément un élément plus petit). En revanche, on a $N(a + b) + N(a - b) = 2(N(a) + N(b))$, donc on peut toujours majorer $N(a + b)$ et $N(a - b)$ par $2(N(a) + N(b))$. Cette majoration suffit pour montrer la terminaison et la complexité des algorithmes que l'on va étudier.

2 Algorithme en $O(M(n) \log n)$ pour le symbole de résiduosité cubique

L'article de R.P. Brent et P. Zimmermann [1] donne un algorithme en $O(M(n) \log n)$ pour calculer le symbole de Jacobi $\left(\frac{b}{a}\right)$ avec a, b de taille n, et tels que a est impair et b est pair. La condition a impair est nécessaire pour la définition même du symbole de Jacobi. La condition b pair en revanche n'est pas toujours satisfaite, mais on peut toujours se ramener à ce cas en considérant $\left(\frac{a+b}{a}\right) = \left(\frac{b}{a}\right)$ si jamais b est impair ($a + b$ est aussi de taille n).
J'ai adapté cet algorithme au cas du calcul de $\left(\frac{b}{a}\right)_3$ dans $\mathbb{Z}[\zeta]$ avec $1 - \zeta \nmid a$ et $1 - \zeta \mid b$. Comme précédemment, la condition $1 - \zeta \nmid a$ est nécessaire pour la définition du symbole, mais pas la condition $1 - \zeta \mid b$. Si on n'a pas $1 - \zeta \mid b$, il suffit de remplacer b par $b' - a'$ où a' et b' sont les éléments primaires associés à a et b.
Avant d'arriver à l'algorithme sous-quadratique, l'article [1] détaille un algorithme cubique, amélioré en algorithme quadratique, amélioré enfin en algorithme quasi-linéaire. Je vais suivre le même principe ici, avec une légère différence puisque l'algorithme cubique une fois adapté à $\mathbb{Z}[\zeta]$ est en fait déjà quasi-quadratique. Il n'y a donc pas besoin de l'améliorer pour qu'il devienne quadratique. Nous verrons donc un premier algorithme quasi-quadratique pour calculer $\left(\frac{b}{a}\right)_3$, puis une amélioration pour obtenir un algorithme en $O(M(n) \log n)$.

2.1 Algorithme quasi-quadratique pour le calcul de $\left(\frac{b}{a}\right)_3$

On définit la valuation d'un élément a de $\mathbb{Z}[\zeta]$ par $\nu(a) = \max\{n \in \mathbb{N} : (1-\zeta)^n \mid a\} \in \mathbb{N}$ (par convention, $\nu(0) = \infty$). Avec cette définition, notre algorithme calcule $\left(\frac{b}{a}\right)_3$ avec $\nu(a) = 0 < \nu(b)$.
L'algorithme est en fait un simple algorithme de calcul de PGCD légèrement modifié. On part de a et b et on calcule la suite des restes r_i jusqu'à obtenir un reste nul. Ici, on fait des divisions par les poids faibles au lieu des poids forts, mais l'algorithme d'Euclide fonctionne de la même manière avec cette division. Pendant le calcul des restes r_i, on en profite pour conserver une variable s telle qu'à chaque étape de l'algorithme $\left(\frac{b}{a}\right)_3 = \zeta^s \left(\frac{r'_{i+1}}{r'_i}\right)_3$, où $r'_i = r_i/(1 - \zeta)^{\nu(r_i)}$ et $r'_{i+1} = r_{i+1}/(1 - \zeta)^{\nu(r_i)}$: on normalise les restes

pour que le symbole cubique soit bien défini. Les propriétés 2 à 8 permettent de mettre à jour s à chaque étape, en fonction des r_i.

2.1.1 Division par les poids faibles

Pour cet algorithme, on a besoin d'un algorithme de division par les poids faibles adapté à $\mathbb{Z}[\zeta]$ (toujours avec l'analogie entre $(1-\zeta)$ et 2). Étant donné a et b tels que $\nu(a) < \nu(b)$ (en pratique on ne l'utilisera que pour $\nu(a) = 0 < \nu(b)$), on veut calculer q et r tels que

$$r = a + q\frac{b}{(1-\zeta)^{\nu(b)-\nu(a)}} \text{ et } \nu(r) > \nu(b).$$

On remarque que si $q = -\frac{a}{(1-\zeta)^{\nu(a)}}(\frac{b}{(1-\zeta)^{\nu(b)}})^{-1} \mod ((1-\zeta)^{\nu(b)-\nu(a)+1})$ et $r = a + q\frac{b}{(1-\zeta)^{\nu(b)-\nu(a)}}$, alors q et r vérifient les conditions précédentes ($\frac{b}{(1-\zeta)^{\nu(b)}}$ est bien inversible car il est premier avec $(1-\zeta)$).

L'objectif de l'algorithme est donc de calculer efficacement $(\frac{b}{(1-\zeta)^{\nu(b)}})^{-1} \mod ((1-\zeta)^{\nu(b)-\nu(a)+1})$. On peut le faire en $O(M(n))$ en utilisant les lemmes de Hensel. On obtient l'algorithme 1.

Algorithm 1 DivisionPoidsFaibles

Entrée: a et b tels que $\nu(a) < \nu(b) < \infty$
Sortie: (q, r) tels que $r = a + qb/(1-\zeta)^{\nu(b)-\nu(a)}$ et $\nu(r) > \nu(b)$

1: $A = -\frac{a}{(1-\zeta)^{\nu(a)}}$
2: $B = \frac{b}{(1-\zeta)^{\nu(b)}}$
3: $n = \nu(b) - \nu(a) + 1$
4: **si** $B = 1 \mod (1-\zeta)$ **alors**
5: $\quad q = 1$
6: **sinon**
7: $\quad q = 2$
8: **fin si**
9: **pour** i de 1 à $\lceil \log n \rceil$ **faire**
10: $\quad q = q + q(1 - Bq) \mod ((1-\zeta)^{2^i})$
11: **fin pour**
12: $q = Aq \mod ((1-\zeta)^n)$
13: $r = a + q\frac{b}{(1-\zeta)^{n-1}}$
14: **renvoyer** (q, r)

La boucle **pour** de l'algorithme permet de calculer $B^{-1} \mod ((1-\zeta)^n)$: à chaque passage dans la boucle, on a $q = B^{-1} \mod ((1-\zeta)^{2^i})$. On obtient à la fin $q = -\frac{a}{(1-\zeta)^{\nu(a)}}(\frac{b}{(1-\zeta)^{\nu(b)}})^{-1} \mod ((1-\zeta)^{\nu(b)-\nu(a)+1})$, c'est bien ce qu'on voulait.

La complexité de l'algorithme est $O(\sum_{i=1}^{\lceil \log n \rceil} M(2^i)) = O(M(n))$. On peut en fait obtenir un algorithme presque deux fois plus rapide en pratique en l'implantant de façon itérative et pas récursive. Comme cet algorithme est appelé très souvent dans les algorithmes suivants, cette amélioration permet de gagner un temps non négligeable dans les algorithmes suivant, comme nous le verrons dans la partie implémentation.

2.1.2 L'algorithme quasi-quadratique

À partir de l'algorithme de division par les poids faibles, on obtient l'algorithme 2 qui est quasi-quadratique.

Algorithm 2 Algo quasi-quadratique de symbole de résiduosité cubique

---

Entrée: a et b tels que $\nu(a) = 0 < \nu(b)$
Sortie: $\left(\frac{b}{a}\right)_3$
1: $s = 0$
2: $j = \nu(b)$
3: **tant que** $b \neq 0$ et $a(1-\zeta)^j \neq b$ **faire**
4: $b' = b/(1-\zeta)^j$
5: $(q, r) = \text{DivisionPoidsFaibles}(a, b)$
6: calculer i_1 et a_1 primaire, tels que $a = (-\zeta)^{i_1} a_1$
7: calculer i_2 et b'_1 primaire, tels que $b' = (-\zeta)^{i_2} b'_1$
8: calculer m_1, n_1 tels que $a_1 = 1 + 3(m_1 + n_1\zeta)$
9: calculer m_2, n_2 tels que $b'_1 = 1 + 3(m_2 + n_2\zeta)$
10: $s \leftarrow (s + j(m_1 + m_2) + i_1(m_2 + n_2) - i_2(m_1 + n_1)) \mod 3$
11: $(a, b) \leftarrow (b', \frac{r}{(1-\zeta)^j})$
12: $j \leftarrow \nu(b)$
13: **fin tant que**
14: **si** $N(a) = 1$ **alors**
15: **renvoyer** ζ^s
16: **sinon**
17: **renvoyer** 0
18: **fin si**

Proposition 1 :
Cet algorithme calcule $\left(\frac{b}{a}\right)_3$ où $\nu(a) = 0 < \nu(b)$, en temps $O(nM(n))$, où n est la taille des entrées : $n = log(N(a) + N(b))$.

Preuve :
1. Correction :
On montre qu'à chaque passage dans la boucle while, si on note a_i, b_i et s_i les valeurs prises par a, b et s au début de cette boucle, alors $\left(\frac{b}{a}\right)_3 = \zeta^{s_i}\left(\frac{b_i}{a_i}\right)_3$, et $\nu(a_i) = 0 < \nu(b_i)$. On le montre par induction. Au départ, $s_0 = 0$, $a_0 = a$ et $b_0 = b$, donc la propriété est vérifiée. Si maintenant $\left(\frac{b}{a}\right)_3 = \zeta^{s_i}\left(\frac{b_i}{a_i}\right)_3$, on utilise les règles du symbole de résiduosité

cubique vues plus haut pour passer de (a_i, b_i) à (a_{i+1}, b_{i+1}) :

$$\begin{aligned}
\left(\frac{b_i}{a_i}\right)_3 &= \left(\frac{b'_i}{a_i}\right)_3 \left(\frac{(1-\zeta)^j}{a_i}\right)_3 \\
&= \left(\frac{b'_{i,1}}{a_{i,1}}\right)_3 \left(\frac{-\zeta}{a_{i,1}}\right)_3^{i_2} \left(\frac{1-\zeta}{a_{i,1}}\right)_3^j \\
&= \left(\frac{a_{i,1}}{b'_{i,1}}\right)_3 \zeta^{-i_2(m_1+n_1)}\zeta^{jm_1} \\
&= \left(\frac{a_i}{b'_i}\right)_3 \left(\frac{(-\zeta)^{-i_1}}{b'_i}\right)_3 \zeta^{jm_1-i_2(m_1+n_1)} \\
&= \left(\frac{r}{b'_i}\right)_3 \zeta^{jm_1+i_1(m_2+n_2)-i_2(m_1+n_1)} \\
&= \left(\frac{r/(1-\zeta)^j}{b'_i}\right)_3 \zeta^{j(m_1+m_2)+i_1(m_2+n_2)-i_2(m_1+n_1)} \\
&= \left(\frac{a_{i+1}}{b_{i+1}}\right)_3 \zeta^{j(m_1+m_2)+i_1(m_2+n_2)-i_2(m_1+n_1)}
\end{aligned}$$

De plus, $\nu(b') = 0 < \nu(\frac{r}{(1-\zeta)^j})$. D'où la propriété au rang $i+1$.

À la fin de l'algorithme, si on note a_k et b_k les valeurs de a et b, on a soit $b_k = 0$, soit $(1-\zeta)^j a_k = b_k$. Dans les deux cas, $\text{pgcd}(a_k, b_k) = a_k$. Si a_k n'est pas une unité, a_k et b_k ne sont pas premiers entre eux, donc $\left(\frac{b_k}{a_k}\right)_3 = 0$. Si en revanche a_k est une unité, $\left(\frac{b_k}{a_k}\right)_3 = 1$, donc $\left(\frac{b}{a}\right)_3 = \zeta^s$. D'où la correction de l'algorithme.

2. Terminaison :
On va montrer que tous les deux passages dans la boucle while, la quantité $N(a) + N(b)$ est divisée par un facteur au moins $9/7$. À chaque étape, on a

$$\begin{aligned}
a_{i+1} &= b_i/(1-\zeta)^j \\
b_{i+1} &= \frac{a_i}{(1-\zeta)^j} + b_i\frac{q}{(1-\zeta)^{2j}}.
\end{aligned}$$

Comme $N(q) < N((1-\zeta)^{j+1}) = 3^{j+1}$, on a

$$\begin{aligned}
N(a_{i+1}) &= N(b_i)/3^j \\
\text{et} \quad N(b_{i+1}) &\leq 2\left(\frac{N(a_i)}{3^j} + \frac{N(b_i)}{3^{j-1}}\right).
\end{aligned}$$

Si $j \geq 2$, on a $N(a_{i+1}) + N(b_{i+1}) \leq 7/9(N(a_i) + N(b_i))$, donc $N(a) + N(b)$ décroît de la quantité voulue en seulement une étape. En revanche, si $j = 1$, on va montrer que $N(a) + N(b)$ décroît toujours mais on ne peut pas minorer le facteur de décroissance. On peut montrer que $N(q) \leq 3$, ce qui est mieux que $N(q) \leq (1-\zeta)^{j+1} = 9$, et qui donne $N(a_{i+1}) + N(b_{i+1}) \leq \frac{2N(a_i)}{3} + N(b_i)$. On sait donc que $(N(a) + N(b))$ ne peut que diminuer à chaque étape, mais si $j = 1$, $(N(a) + N(b))$ peut diminuer d'un facteur

très petit. En revanche, si on regarde deux étapes successives, soit on a $j \geq 2$ dans une des deux étapes, et donc on a bien $N(a_{i+2}) + N(b_{i+2}) \leq 7/9(N(a_i) + N(b_i))$, soit $j = 1$ dans les deux étapes, et on a que $N(a_{i+2}) + N(b_{i+2}) \leq \frac{2}{3}(N(a_i) + N(b_i))$.
En deux étapes, on a donc bien une diminution d'un facteur au moins $9/7$, donc l'algorithme termine, et on sait même que le nombre de passages dans la boucle while effectués par l'algorithme est en $O(\log(N(a) + N(b))) = O(n)$.

3.Complexité :
On a vu que le nombre de passage dans la boucle while est $O(n)$, il reste à évaluer le temps de calcul pour chaque passage dans la boucle. L'algorithme de division binaire prend un temps $O(M(n))$ et c'est l'opération la plus coûteuse de la boucle, donc la complexité totale de l'algorithme est $O(nM(n))$, ce qui est quasi-quadratique.

2.2 Algorithme en $O(M(n)\log n)$ pour le calcul de $\left(\frac{b}{a}\right)_3$

L'algorithme en $O(M(n)\log n)$ suit le même principe que l'algorithme de [1]. Il est basé sur le lemme suivant :

Lemme. *Soient a, b, a', b' et k tels que $a = a' \mod (1-\zeta)^{2k+1}$ et $b = b' \mod (1-\zeta)^{2k+1}$.*
On considère la suite des restes et des quotients de a et b (pour la division par les poids faibles vue plus haut) : $r_0 = a,\ r_1 = b,\ r_2 = a + q_1 b_1$ ... (où $b_1 = \frac{b}{(1-\zeta)^{\nu(b)}}$), jusqu'au premier reste r_j tel que $\nu(r_j) > k$. On considère de même la suite des restes et des quotients r'_i et q'_i de a' et b'. On a alors pour tout $i \leq j$, $q_i = q'_i$ et $r_{i+1} \equiv r'_{i+1} \mod (1-\zeta)^{2k+1-\nu(r_i)}$.

Grâce à ce lemme, on peut gagner du temps dans l'algorithme précédent : chaque étape revient à calculer un nouveau reste r_i de la suite des restes de a et b. On les normalise en divisant par $(1-\zeta)$, mais à chaque étape de l'algorithme, il existe i tel que $a = \frac{r_i}{(1-\zeta)^{\nu(r_i)}}$ et $b = \frac{r_{i+1}}{(1-\zeta)^{\nu(r_i)}}$. Comme on a vu que les quotients restaient les mêmes si on prenait a' et b' congrus à a et b, on peut réduire le temps de calcul en prenant des a' et b' plus petits que a et b (congrus à a et b modulo $(1-\zeta)^k$ pour k bien choisi).

2.2.1 Demi Algorithme

L'algorithme est en fait une adaptation d'un algorithme de PGCD rapide, dans lequel on calcule à chaque étape s tel que $\left(\frac{b}{a}\right)_3 = \zeta^s \left(\frac{b_i}{a_i}\right)_3$. Dans l'algorithme quadratique précédent, on calculait la suite des restes, en partant de a et b, jusqu'à arriver à un reste nul. L'idée pour accélérer le calcul est de profiter du fait que les quotients sont égaux, pour calculer plusieurs restes avec des entrées plus petites. Cette idée est utilisée dans l'algorithme 3 et va permettre d'obtenir l'algorithme quasi-linéaire final.

Théorème 1. *L'algorithme DemiAlgo prend en entrée deux éléments a et b de $\mathbb{Z}[\zeta]$, et un entier k et renvoie deux entiers s, j et une matrice 2×2, R tels que :*

Algorithm 3 DemiAlgo

Entrée: a et b tels que $\nu(a) = 0 < \nu(b)$ et un entier k

Sortie: deux entiers s, j et R une matrice 2×2 comme définis dans le théorème 1

1: **si** $\nu(b) > k$ **alors**

2:     **renvoyer** $0, 0, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$

3: **fin si**

4:

5: [premier appel récursif]

6: $k_1 = \lfloor k/2 \rfloor$

7: $a_1 = a \mod (1-\zeta)^{2k_1+4},\ b_1 = b \mod (1-\zeta)^{2k_1+4}$

8: $s_1, j_1, R \leftarrow$ DemiAlgo (a_1, b_1, k_1)

9: $a' = (1-\zeta)^{-2j_1}(R_{1,1}a + R_{1,2}b),\ b' = (1-\zeta)^{-2j_1}(R_{2,1}a + R_{2,2}b)$

10: $j_0 \leftarrow \nu(b')$

11: **si** $j_0 + j_1 > k$ **alors**

12:     **renvoyer** s_1, j_1, R

13: **fin si**

14:

15: [une étape de calcul]

16: $b'' = b'/(1-\zeta)^{j_0}$

17: $(q, r) =$ Division binaire(a', b')

18: calculer i_1 et a'_1 primaire, tels que $a' = (-\zeta)^{i_1} a'_1$

19: calculer i_2 et b''_1 primaire, tels que $b'' = (-\zeta)^{i_2} b''_1$

20: calculer m_1, n_1 tels que $a'_1 = 1 + 3(m_1 + n_1\zeta)$

21: calculer m_2, n_2 tels que $b''_1 = 1 + 3(m_2 + n_2\zeta)$

22: $s_0 \leftarrow (j(m_1 + m_2) + i_1(m_2 + n_2) - i_2(m_1 + n_1)) \mod 3$

23: $(a_2, b_2) \leftarrow (b'', \frac{r}{(1-\zeta)^{j_0}})$

24: $Q \leftarrow \begin{pmatrix} 0 & 2^{j_0} \\ 2^{j_0} & q \end{pmatrix}$

25:

26: [deuxième appel récursif]

27: $k_2 = k - (j_0 + j_1)$

28: $s_2, j_2, S \leftarrow$ DemiAlgo$(a_2 \mod (1-\zeta)^{2k_2+4}, b_2 \mod (1-\zeta)^{2k_2+4})$

29: **renvoyer** $(s_0 + s_1 + s_2) \mod 3,\ j_0 + j_1 + j_2,\ S \times Q \times R$

– *si* $\begin{pmatrix} c \\ d \end{pmatrix} = (1-\zeta)^{-2j} R \begin{pmatrix} a \\ b \end{pmatrix}$*, alors* $r_i = (1-\zeta)^j c$ *et* $r_{i+1} = (1-\zeta)^j d$ *sont deux restes successifs de la suite des restes de* a *et* b *tels que* $\nu(r_i) \leq k < \nu(r_{i+1})$*, et* $\nu(r_i) = j$ *(ie* $\nu(c) = 0$*).*

– $\left(\frac{b}{a}\right)_3 = \zeta^s \left(\frac{d}{c}\right)_3$.

L'algorithme s'appelle récursivement sur des entrées plus petites, et comme on sait que les quotients sont égaux pour un certain nombre de restes, on avance directement jusqu'à ces restes (voir Figure 1).

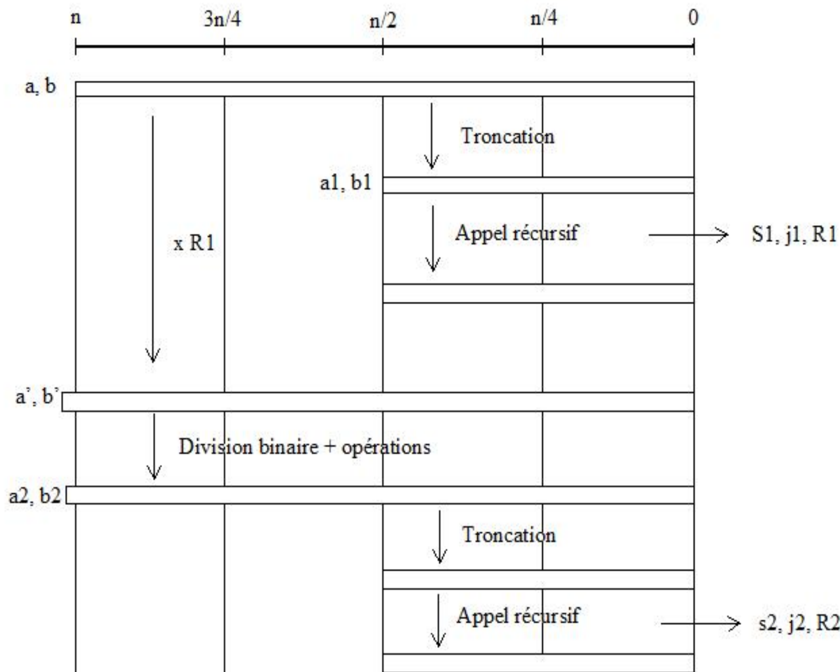

FIGURE 1 – Structure de DemiAlgo, adapté d'un schéma de [4]

Preuve du théorème 1.
On prouve le théorème par induction. Si $\nu(b) > k$, les deux restes successifs tels que $\nu(r_i) \leq k < \nu(r_{i+1})$ sont a et b, donc la sortie est correcte. Sinon, on calcule s_1, j_1 et R ; par hypothèse d'induction ce calcul est correct. Les coefficients de R sont des combinaisons linéaires des quotients de la suite des restes de a_1 et b_1. Comme on a vu que les quotients étaient égaux, a' et b' sont bien les premiers restes obtenus à partir de a et b tels que $\nu((1-\zeta)^j a') \leq k < \nu((1-\zeta)^j b')$. La matrice a été obtenue en appliquant l'algorithme à des éléments plus petits, mais lorsqu'on l'applique aux éléments initiaux, on obtient bien les bons restes. Il faut maintenant montrer que le calcul de s fait avec des éléments plus petits reste correct pour les éléments initiaux.

Comme on a choisi $a_1 \equiv a \mod ((1-\zeta)^{2k_1+4}$ et $b_1 \equiv b \mod ((1-\zeta)^{2k_1+4}$, si on note r_i les restes de la suite partant de a et b, et r'_i ceux de la suite partant de a_1 et b_1, et si j est le plus petit entier tel que $\nu(r_{j+1}) > k$, on a vu que $r_{i+1} \equiv r'_{i+1} \mod (1-\zeta)^{2k_1+4-\nu(r_i)}$ pour tout $i \leq j$. Comme $\nu(r_i) \leq k_1$, on a donc $r_{i+1} \equiv r'_{i+1} \mod (1-\zeta)^{k_1+4}$. C'est cette égalité qui va nous permettre d'assurer que le s calculé avec les petites entrées reste le bon. C'est pour ça qu'on prend $a_1 \equiv a \mod ((1-\zeta)^{2k_1+4}$ et pas $a_1 \equiv a \mod ((1-\zeta)^{2k_1+1}$. Le $2k_1+1$ permet d'assurer l'égalité des quotients, mais pas que le calcul de s est le bon.
Le calcul de s dans l'algorithme ne dépend que des restes intermédiaires. Il suffit donc de montrer qu'à chaque étape de calcul, calculer s avec r_i et r_{i+1} ou avec $r'_i \equiv r_i \mod (1-\zeta)^{k_1+4}$ et $r'_{i+1} \equiv r_{i+1} \mod (1-\zeta)^{k_1+4}$ donne le même résultat. À chaque étape de calcul, si les restes sont r_i et r_{i+1}, on considère les restes normalisés : $r_{i,1} = r_i/(1-\zeta)^{\nu(r_i)}$ et $r_{i+1,1} = r_{i+1}/(1-\zeta)^{\nu(r_i)}$. Comme on sait que $\nu(r_i) \leq k_1$, on a alors égalité entre les restes normalisés modulo $(1-\zeta)^4 = 9$: $r_{i,1} \equiv r'_{i,1} \mod 9$ et $r_{i+1,1} \equiv r'_{i+1,1} \mod 9$. Pour calculer s, on a besoin de i, m et n tels que $r_{i,1} = (-\zeta)^i(1+3(m+n\zeta))$ (et pareil pour $r_{i+1,1}$). Or si $(-\zeta)^i(1+3(m+n\zeta)) \equiv (-\zeta)^{i'}(1+3(m'+n'\zeta)) \mod 9$, alors $(-\zeta)^i = (-\zeta)^{i'}$, puis $m+n\zeta \equiv m'+n'\zeta \mod 3$, ce qui donne $m \equiv m' \mod 3$, $n \equiv n' \mod 3$ et $i \equiv i' \mod 3$. Comme le calcul de s se fait modulo 3, le s calculé avec les petits éléments est bien le même que celui qui aurait été calculé avec les grands éléments.

L'étape de calcul au milieu de l'algorithme se fait avec les grands éléments, donc l'algorithme reste correct. Et le deuxième appel récursif s'étudie comme le premier. Le $2k_2+4$ permet de s'assurer que le s calculé est le bon. Enfin, comme $k_2 = k-(j_0+j_1)$, le reste obtenu en sortie est bien le premier tel que $\nu(r_{i+1})$ dépasse k. □

On peut maintenant passer à l'algorithme rapide, qui utilise DemiAlgo.

2.2.2 Algorithme quasi-linéaire

L'algorithme 4 calcule le symbole de résiduosité cubique en temps quasi-linéaire. La correction de cet algorithme vient de la correction de DemiAlgo et de celle de l'algorithme quadratique. C'est le même algorithme que le quadratique sauf qu'on calcule les restes par paquets en appelant DemiAlgo.

2.2.3 Complexité

Théorème 2. *Complexité de DemiAlgo.*
Si a et b sont des éléments de taille n (ie $\log_3(N(a)) = n$ et $\log_3(N(b)) = n$), et si $k = n/2$, alors la complexité en nombre d'opérations sur les bits de DemiAlgo(a,b,k) est $T(n) = O(M(n)\log n)$.

Preuve. Soient a et b de taille n et $k = n/2$. Alors si $a_1 \equiv a \mod (1-\zeta)^{2k_1+4}$, $N(a_1) \leq N(1-\zeta)^{2k_1+4} \leq 3^{k+4}$, donc a_1 est de taille k ($\log_3 a_1$ est de l'ordre de k). De même, b_1 est de taille k, donc l'appel à DemiAlgo pour a_1, b_1 et k_1 se fait en temps $T(n/2)$. Comme $j_0 + j_1 > k_1$, on a $k_2 \leq \lfloor k/2 \rfloor$ donc le deuxième appel récursif à

Algorithm 4 AlgoRapide

Entrée: a et b tels que $\nu(a) = 0 < \nu(b)$

Sortie: $\left(\frac{b}{a}\right)_3$

1: $s = 0$
2: $j = \nu(b)$
3: **tant que** $b \neq 0$ et $a(1-\zeta)^j \neq b$ **faire**
4: $\quad k = \max(\nu(b), \log(N(b))/2)$
5: $\quad s', j, R \leftarrow \text{DemiAlgo}(a, b, k)$
6: $\quad s \leftarrow (s + s' \mod 3)$
7: $\quad (a, b) \leftarrow (1-\zeta)^{-2j}(R_{1,1}a + R_{1,2}b, R_{2,1}a + R_{2,2}b)$
8: $\quad j \leftarrow \nu(b)$
9: **fin tant que**
10: **si** $N(a) = 1$ **alors**
11: $\quad$ **renvoyer** ζ^s
12: **sinon**
13: $\quad$ **renvoyer** 0
14: **fin si**

DemiAlgo se fait aussi en temps $T(n/2)$. Le reste de l'algorithme effectue une étape de calcul de l'algorithme quadratique, et on a vu qu'on pouvait le faire en $O(M(n))$.
On a donc $T(n) = 2T(n/2) + cM(n)$. D'après le master théorème, on obtient $T(n) = O(M(n) \log n)$. $\square$

On déduit de la complexité de DemiAlgo la complexité de AlgoRapide.

Théorème 3. *Complexité de AlgoRapide.*
Si a et b sont des éléments de taille n, la complexité en nombre d'opérations sur les bits de AlgoRapide est en $O(M(n) \log n)$.

3 Implémentation

J'ai implanté les algorithmes précédents en Sage ([5]), ainsi que l'algorithme quadratique de Damgård et Frandsen [2] pour calculer le symbole de résiduosité cubique.

3.1 Améliorations de DemiAlgo

Les deux premières améliorations décrites ci-dessous utilisent le fait que DemiAlgo est un algorithme récursif qui s'appelle deux fois sur des entrées de taille réduite de moitié. Même pour de grandes entrées, DemiAlgo fait donc de nombreux appels récursifs avec de petites entrées. Si on arrive à accélérer DemiAlgo pour de petites entrées, il devient plus rapide aussi pour de grandes entrées.

3.1.1 Première amélioration

Même si AlgoRapide est asymptotiquement plus rapide que l'algorithme quadratique de Damgård et Frandsen, pour des entrées de taille inférieure à 260 000 bits (où la taille de a est toujours le nombre de bits de $N(a)$), l'algorithme quadratique est plus rapide que

le linéaire. On peut donc essayer de l'utiliser au lieu de DemiAlgo pour des entrées de petite taille (il faut l'adapter un peu car les deux algorithmes ne calculent pas exactement la même chose).
J'ai adapté l'algorithme de [2] pour qu'il renvoie la matrice et les valeurs de j et s que renverrait un appel à DemiAlgo. Pour chaque appel à DemiAlgo, si la taille des entrées est suffisamment petite, on utilise l'algorithme quadratique adapté de [2] au lieu de DemiAlgo pour aller plus vite. Expérimentalement, j'ai observé que la limite qui donnait les meilleurs résultats était 400 bits : si la valeur de k dans DemiAlgo(a,b,k) est plus petite que 400, on utilise l'algorithme quadratique modifié. Comme k est de l'ordre de la taille de b divisée par deux, c'est un moyen rapide d'obtenir la taille des entrées.
L'algorithme obtenu, appelé AlgoFusion, reste quasi-linéaire et devient meilleur que l'algorithme quadratique de [2] à partir d'entrées de taille 60 000 bits.

3.1.2 Deuxième amélioration

Pour accélérer DemiAlgo sur de petites entrées, j'ai également pré-calculé les valeurs de $(1-\zeta)^k$ pour k allant de 1 à 1000. Pour l'algorithme de Jacobi, on avait besoin de 2^k, ce qui se calcule très rapidement, mais pour $(1-\zeta)^k$ le calcul est plus long, et le pré-calcul permet d'accélérer un peu l'algorithme.

3.1.3 Troisième amélioration

On peut gagner encore du temps en modifiant légèrement l'algorithme DivisionPoidsFaibles, pour qu'il soit récursif au lieu d'itératif. Dans la version exposée plus haut, on calculait l'inverse modulo $(1-\zeta)^{2^i}$ jusqu'à ce que i dépasse $\lceil \log n \rceil$. On s'arrête donc à la première puissance de 2 dépassant n, or on ne s'intéresse à l'inverse de B que modulo $(1-\zeta)^n$, donc dans la plupart des cas on fait trop de calculs. En procédant de manière récursive (en partant de n, puis en considérant $n/2$ etc), on sait qu'on ne fera pas une grosse quantité de calculs en trop. La complexité théorique reste $O(M(n))$ mais en pratique l'algorithme récursif est presque deux fois plus rapide sur de petites entrées. Comme l'algorithme quasi-linéaire décrit ci-dessus fait de nombreux appels à DivisionPoidsFaibles, cette amélioration permet de gagner beaucoup de temps (à peu près un facteur deux). En revanche, AlgoFusion utilise l'algorithme quadratique de [2] pour les petites entrées, et cet algorithme n'utilise pas DivisionPoidsFaibles, donc on ne gagne pas tant que ça pour l'algorithme déjà amélioré. Cette amélioration permet de diminuer le temps de calcul d'environ 5% pour AlgoFusion.

3.2 Vérification de l'exactitude des résultats

J'ai commencé par vérifier l'exactitude des résultats que fournissait AlgoRapide et j'ai ensuite comparé les autres algorithmes à AlgoRapide.
Pour vérifier l'exactitude d'AlgoRapide, j'ai comparé, pour des éléments p premiers, le calcul de $\left(\frac{a}{p}\right)_3$ renvoyé par l'algorithme et $a^{\frac{N(p)-1}{3}} \mod p$ (en partant du principe que mon algorithme pour calculer $a^{\frac{N(p)-1}{3}} \mod p$ était correct). Puis j'ai vérifié que pour

des éléments quelconques (pour lesquels le symbole est défini),

$$\left(\frac{a_1 a_2}{b_1 b_2}\right)_3 = \left(\frac{a_1}{b_1}\right)_3 \left(\frac{a_1}{b_2}\right)_3 \left(\frac{a_2}{b_1}\right)_3 \left(\frac{a_2}{b_2}\right)_3 .$$

3.3 Résultats Expérimentaux

La figure 2 montre le temps de calcul en fonction de la taille des entrées pour l'algorithme quadratique de [2], l'algorithme quasi-linéaire AlgoRapide (avec DivisionPoidsFaibles itératif ou récursif), et l'algorithme final AlgoFusion (avec DivisionPoidsFaibles récursif).

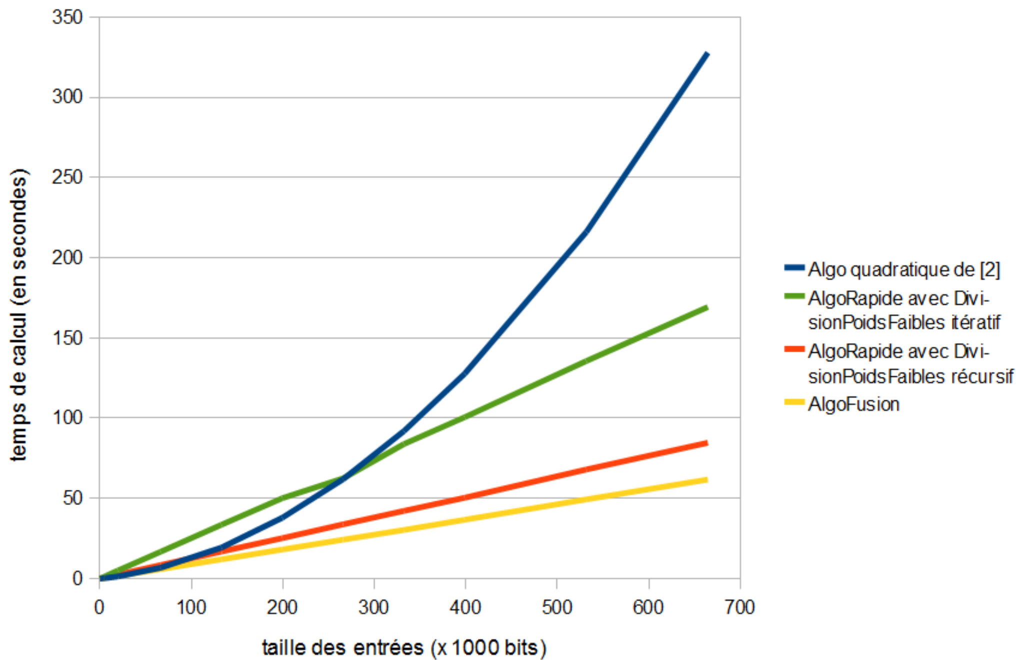

FIGURE 2 – Comparaison des différents algorithmes pour le symbole cubique

4 Puissances quatrièmes

On peut adapter les algorithmes précédents pour calculer le symbole de résiduosité quartique, défini dans $\mathbb{Z}[i]$ cette fois.

4.1 Analogie avec les cubes

Le symbole de résiduosité quartique est l'analogue du symbole de résiduosité cubique mais pour les puissances quatrièmes. Il est défini dans l'anneau des entiers de Gauss $\mathbb{Z}[i]$.

Définition 2. *Symbole de résiduosité quartique.*

$$\left(\frac{\cdot}{\cdot}\right)_4 : \mathbb{Z}[i] \times (\mathbb{Z}[i] - (1+i)\mathbb{Z}[i]) \to \{0, 1, i, -1, -i\}$$
$$(a,p) \mapsto a^{\frac{N(p)-1}{4}} \text{ si } p \text{ est premier}$$
$$(a,b) \mapsto \left(\frac{a}{p_1}\right)_4 \cdots \left(\frac{a}{p_k}\right)_4 \text{ si } b = p_1 \cdots p_k \text{ (avec les } p_i \text{ premiers).}$$

Si $p \in \mathbb{Z}[i]$ est premier et p ne divise pas a, $\left(\frac{a}{p}\right)_4 = 1$ si et seulement si a est une puissance quatrième dans $\mathbb{Z}[i]$ modulo p.

Cette fois les unités de l'anneau sont $\{1, i, -1, -i\}$, l'élément premier jouant le rôle de 2 est $1+i$ et on dit que a est primitif si $a \equiv 1 \mod (1+i)^3$. Comme précédemment, si $1+i$ ne divise pas a, il existe un et un seul des associés de a qui est primitif, et la différence de deux éléments primitifs est divisible par $1+i$. L'anneau $\mathbb{Z}[i]$ est toujours euclidien, ce qui permet d'utiliser l'algorithme d'Euclide et donc les algorithmes précédents.

Enfin, on a toujours les mêmes propriétés pour le symbole de résiduosité quartique, qui vont permettre de le calculer en même temps que le pgcd.

Propriétés. Soient $a, b, c, d \in \mathbb{Z}[i]$ tels que $1+i$ ne divise pas c et d.

1. $\left(\frac{a}{c}\right)_4 = \left(\frac{b}{c}\right)_4$ si $a = b \mod c$
2. $\left(\frac{ab}{c}\right)_4 = \left(\frac{a}{c}\right)_4 \left(\frac{b}{c}\right)_4$ (si $(1-\zeta) \nmid c$)
3. $\left(\frac{a}{cd}\right)_4 = \left(\frac{a}{c}\right)_4 \left(\frac{a}{d}\right)_4$ (si $(1-\zeta) \nmid c, d$)
4. si a et b sont primaires, $\left(\frac{a}{b}\right)_4 = i^{\frac{(N(a)-1)(N(b)-1)}{8}} \left(\frac{b}{a}\right)_4$ (loi de réciprocité quartique)

Si $b = 1 + (1+i)^3(m+ni)$ est primaire, on a aussi les relations suivantes :

5. $\left(\frac{i}{b}\right)_4 = i^{2(m^2+n^2)-(m+n)}$
6. $\left(\frac{1+i}{b}\right)_4 = i^{-(m-n)^2-m}$

Avec toutes ces ressemblances on peut transformer assez facilement les algorithmes précédents pour calculer le symbole de résiduosité quartique. Il y a quand même quelques petites différences.

4.2 Différences

Dans l'algorithme quadratique (et aussi dans l'algorithme rapide), lorsqu'on mettait à jour s à chaque étape pour avoir $\left(\frac{a}{b}\right)_3 = \zeta^s \left(\frac{r'_{i+1}}{r'_i}\right)_3$, on faisait

$$s \leftarrow (s + j(m_1+m_2) + i_1(m_2+n_2) - i_2(m_1+n_1)) \mod 3.$$

Maintenant, la mise à jour de s devient

$$s \leftarrow (s - j((m_1+n_1)^2 + (m_2-n_2)^2 + m_1 + m_2) - i_1(2(m_2^2+n_2^2) - (m_2+n_2)) + i_2(2(m_1^2+n_1^2) - (m_1+n_1)) + \frac{(N(a)-1)(N(b)-1)}{8}) \mod 4.$$

En plus d'être plus longue que la formule précédente, cette nouvelle formule fait intervenir m et n au carré ainsi que la norme de a et b, ce qui fait que le temps de calcul de cet étape n'est plus linéaire. Il faut donc réduire m et n modulo 4 avant de faire des calculs. De même, pour le calcul de la norme, il suffit de la calculer modulo 32, ce qui permet de faire le calcul en temps linéaire (voire constant car les réductions modulo une puissance de 2 peuvent se faire en temps constant en ne regardant que les derniers bits de l'entier).

La preuve de terminaison de l'algorithme quadratique est aussi un peu différente de celle dans $\mathbb{Z}[\zeta]$.

4.3 Résultats expérimentaux

J'ai transformé les trois algorithmes précédents (l'algorithme quadratique de [2], AlgoRapide et AlgoFusion) pour calculer le symbole de résiduosité quartique au lieu du symbole de résiduosité cubique. Je n'ai pas eu le temps d'implémenter la troisième optimisation décrite ci-dessus (avec DivisionPoidsFaibles en récursif), mais si on compare ces algorithmes avec les algorithmes pour les cubes sans cette optimisation, les temps de calcul sont globalement les mêmes (voir figure 3).

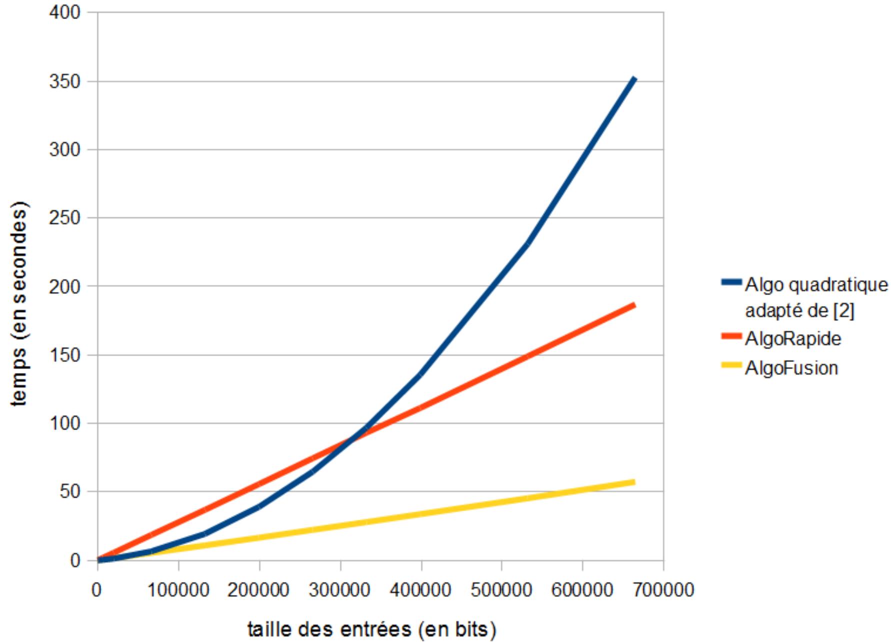

FIGURE 3 – Comparaison des différents algorithmes pour le symbole quartique

5 Conclusion

On a vu que les symboles de résiduosité cubique et quartique étaient très proches du symbole de Jacobi. De plus, les propriétés des anneaux dans lesquels on travaille ($\mathbb{Z}[\zeta_3]$ et $\mathbb{Z}[\zeta_4]$, où ζ_n est une racine nième primitive de l'unité) sont Euclidiens pour la norme complexe, ce qui permet d'adapter facilement les algorithmes de PGCD de $\mathbb{Z}$ à ces anneaux.
Même si cela ne semble pas avoir beaucoup d'application, on pourrait continuer à adapter les algorithmes précédents aux puissances 5, 6... Cependant, l'adaptation ne sera pas forcément toujours aussi facile que pour les cas 3 et 4 (ni même possible). L'anneau $\mathbb{Z}[\zeta_n]$ peut ne pas être Euclidien, ni même principal. Dans ce cas il semble difficile d'adapter les algorithmes de PGCD basés sur l'algorithme d'Euclide. Même si l'anneau est Euclidien, il peut avoir une infinité d'unités, ce qui interdit de parcourir tous les associés d'un élément pour trouver celui qui nous convient.

Références

[1] Brent, R.P., Zimmermann, P., *An $O(M(n)\log n)$ Algorithm for the Jacobi Symbol*. ANTS-IX 2010, LNCS 6197, pp83-95. Springer, Heidelberg (2010)

[2] Damgård, I.B., Frandsen, G.S., *Efficient Algorithms for GCD and Cubic Residuosity in the Ring of Eisenstein Integers*. FCT 2003, LNCS 2751, pp 109-117. Springer, Heidelberg (2010).

[3] Bosma, W., *Cubic reciprocity and explicit primality tests for h· 3k±1*. High primes and misdemeanours : lectures in honour of the 60th birthday of Hugh Cowie Williams, Fields Inst. Commun, 41, 77-89 (2004).

[4] Stehlé, D., Zimmermann, P., *A binary recursive gcd algorithm*. ANTS 2004. LNCS, vol. 3076, pp 411-425. Springer, Heidelberg (2004).

[5] W. A. Stein et al., *Sage Mathematics Software (Version 5.9)*, The Sage Development Team, 2013 `http://www.sagemath.org`.

[6] Schonhage, A., Strassen, V, *Fast multiplication of large numbers.* Computing, vol 7, no 3-4, p. 281-292 (1971).

[7] Fürer, M., *Faster integer multiplication.* SIAM Journal on Computing, 39(3), 979-1005 (2009).

[8] Ireland, K., Rosen, M. I., *A classical introduction to modern number theory* (Vol. 84). Springer (1990).