Recherche efficace de vecteur court dans un réseau euclidien

Xavier Pujol

Rapport de stage de M2
Encadrant : Damien Stehlé

16 juin 2008

Ce rapport décrit le travail que j'ai effectué pendant mon stage de Master 2 entre le 21 janvier et le 13 juin 2008. Ce stage s'est déroulé au Laboratoire de l'Informatique du Parallélisme (LIP) à l'ENS de Lyon, dans l'équipe-projet Arénaire, et a été encadré par Damien Stehlé. Le principal domaine de recherche d'Arénaire est l'arithmétique des ordinateurs, et en particulier l'arithmétique flottante.

1 Introduction

Un *réseau euclidien* est une grille régulière de points dans $\mathbb{R}^n$. Plus précisément, si $b = (\mathbf{b_1}, \ldots, \mathbf{b_d})$ est une famille de vecteurs linéairement indépendants de $\mathbb{R}^n$, le réseau euclidien engendré par b, noté $L(b)$, est l'ensemble des combinaisons linéaires à coefficients entiers de $\mathbf{b_1}, \ldots, \mathbf{b_d}$. On dit que b est une *base* du réseau $L(b)$. Plusieurs bases peuvent engendrer le même réseau, mais elles ont toutes le même cardinal. Ce cardinal est la *dimension* du réseau. Un réseau de dimension 2 ou plus possède une infinité de bases distinctes.

Étant donné que les réels sont indénombrables, on ne peut pas coder un vecteur quelconque de $\mathbb{R}^n$ en espace fini. En particulier, il est impossible de le représenter sur ordinateur. Par la suite, on se restreindra aux réseaux engendrés par des familles de vecteurs à coefficients rationnels.

Dans ce rapport, nous nous intéressons à la résolution de deux problèmes liés aux réseaux euclidiens, SVP principalement et CVP, en utilisant l'algorithme de Kannan-Fincke-Pohst, abrégé KFP [8, 5]. Le problème du plus court vecteur, noté SVP pour *Shortest Vector Problem*, consiste à trouver un plus court vecteur non nul dans un réseau rationnel L à partir d'une base de ce réseau. La norme de ce vecteur est appelée minimum du réseau. Cette définition dépend de la norme dont est muni $\mathbb{R}^n$. Dans ce rapport, on ne considérera que la norme euclidenne. Le problème du plus proche vecteur, noté CVP (*Closest Vector Problem*) prend en entrée en base b d'un réseau rationnel et un vecteur cible $t \in \mathbb{Q}^n$. Le but est trouver un vecteur de $L(b)$ le plus proche possible de t. Ces deux problèmes sont NP-difficiles, et on ne sait pas s'ils sont dans NP. Avec les algorithme connus, il faut un temps exponentiel en la dimension du réseau pour les résoudre ou même pour en vérifier la solution.

Plusieurs cryptosystèmes, dont NTRU [7] qui est utilisé en pratique, reposent sur la difficulté de résolution de SVP ou de CVP. Pour choisir raisonnablement les paramètres de sécurité de ces cryptosystèmes, il est important de savoir précisément jusqu'à quelle dimension on sait résoudre ces problèmes. La résolution de CVP a également des applications en télécommunications, pour la technologie MIMO (qui permet d'augmenter la capacité des réseaux sans fil en utilisant plusieurs antennes émettant à la même fréquence). Dans ce cas, la dimension des réseaux est très petite mais la résolution doit être très rapide.

L'algorithme KFP résout à la fois SVP et CVP. Théoriquement, il requiert des calculs exacts sur des rationnels de grande taille, ce qui est très coûteux par rapport à des calculs en virgule flottante. Dans ce rapport, nous apportons les premières garanties théoriques sur l'algorithme KFP en virgule flottante et décrivons la première implémentation certifiée de cet algorithme.

Dans la section 2, nous rappelons plusieurs notions essentielles sur les nombres flottants et les réseaux euclidiens. La section 3 explique l'algorithme KFP. Puis nous montrons dans la section 4 que le résultat de l'algorithme KFP flottant pour SVP est correct sous certaines conditions. Nous donnons un résultat de complexité sur l'algorithme KFP pour CVP dans la

section 5. Enfin, dans la section 6, nous présentons notre implémentation de l'algorithme KFP utilisant les résultats des deux parties précédentes, puis une version parallèle de cet algorithme et une version matérielle sur FPGA en cours de développement.

Dans ce rapport et en particulier dans la section 4, nous avons besoin d'introduire un grand nombre de notations. Le lecteur pourra au besoin se reporter à l'annexe B dans laquelle elles sont résumées.

2 Contexte

Dans cette section, nous rappelons la définition de l'arithmétique flottante et en donnons les propriétés essentielles qui servent en analyse d'erreur. Ensuite, nous présentons la réduction de réseaux euclidiens et son rapport avec l'algorithme KFP.

2.1 Arithmétique flottante

La plupart des processeurs actuels ne peuvent calculer qu'avec deux types de données, les entiers et les nombres à virgule flottante (ou plus simplement *flottants*). En machine, les entiers sont codés en binaire sur un nombre fixé de bits (typiquement 32 ou 64), ce qui restreint la plage des valeurs représentables (par exemple, $[\![-2^{31}, 2^{31}-1]\!]$ avec 32 bits). Tant qu'on reste dans cette plage, les opérations en machine produisent les mêmes résultats qu'en théorie. Les flottants sont utilisés pour représenter les réels. Étant donnée une précision $p > 0$, l'ensemble FL_p des flottants de précision p est formé des nombres de la forme :

$$\pm m \cdot 2^{e-p}$$

où $m \in [\![2^{p-1}, 2^p - 1]\!]$ et e est un entier relatif borné. On appelle m la *mantisse* du flottant et e son *exposant*. Zéro est également inclus dans FL_p bien qu'il ne respecte pas ce format. En machine, un flottant est représenté sur un nombre de bits fixé en fonction de p et de la plage de e. Deux formats de flottants sont standardisés par la norme IEEE 754 [1] : la simple précision (codage sur 32 bits) et la double précision (codage sur 64 bits).

Le calcul avec des flottants pose plus de difficultés qu'avec les entiers. D'une part, comme pour les entiers, la plage des nombres représentable est bornée. Ce n'est pas un problème ici, car on peut facilement borner la taille des nombres manipulés dans tous les algorithmes utilisés. D'autre part, les opérateurs sur les flottants ne produisent pas des résultats exacts. En effet, étant donnés deux flottants a et b de FL_p, les réels $a+b$, $a-b$ et $a \times b$ ne sont pas en général dans FL_p. La norme IEEE 754 spécifie que les opérateurs en virgule flottante (notées par la suite $\oplus$, $\ominus$ et $\otimes$) doivent renvoyer le résultat exact arrondi au flottant le plus proche, de mantisse paire s'il y a ambiguïté.

L'écart minimal entre deux flottants d'exposant 0 est égal à $\epsilon = 2^{-p}$. Pour des flottants d'exposant e quelconque, l'écart minimal vaut $2^e \cdot \epsilon$. Si le résultat d'une opération est compris dans l'intervalle $[2^e, 2^{e+1}[$, alors le flottant le plus proche se trouve à une distance au plus $2^e \cdot \epsilon/2$ de ce résultat. On en déduit les deux propriétés suivantes :

$$|(a \oplus b) - (a+b)| \leq (a+b)\epsilon/2 \ , \quad |(a \oplus b) - (a+b)| \leq (a \oplus b)\epsilon/2.$$

Ces propriétés sont vraies si on remplace $+$ et $\oplus$ par $-/\ominus$ ou $\times/\otimes$. De plus, $\oplus$ est croissante par rapport à ses deux variables en toutes circonstances, et $\otimes$ est croissante par rapport à ses

deux variables quand elles sont positives. Pour analyser l'algorithme KFP, seules ces propriétés seront utilisées.

2.2 Orthogonalisation de Gram-Schmidt

Soit $b = (\mathbf{b_1}, \ldots, \mathbf{b_d})$ une famille de vecteurs libres de $\mathbb{R}^n$. Noter que ici et dans toute la suite, l'ordre des vecteurs de la famille est important. L'orthogonalisation de Gram-Schmidt de b est la famille $b^* = (\mathbf{b_1^*}, \ldots, \mathbf{b_d^*})$ définie par :

$$\begin{cases} \mathbf{b_1^*} = \mathbf{b_1} \\ \forall i \in [\![2, d]\!], \mathbf{b_i^*} = \mathbf{b_i} - \sum_{j=1}^{i-1} \mu_{i,j} \mathbf{b_j^*} \\ \forall i \in [\![2, d]\!], \forall j \in [\![1, i-1]\!], \mu_{i,j} = (\mathbf{b_i} | \mathbf{b_j^*}) / \|\mathbf{b_j^*}\|^2. \end{cases} \tag{1}$$

Les vecteurs $\mathbf{b_i^*}$ sont orthogonaux deux à deux et les matrices de passage entre b et b^* sont triangulaires, avec les coefficients diagonaux égaux à 1.

2.3 Réduction de réseau

Une base d'un réseau est dite *réduite* quand ses vecteurs sont relativement orthogonaux et relativement courts par rapport au minimum du réseau. L'algorithme KFP fonctionne d'autant plus efficacement que la base fournie en entrée est réduite. Il existe plusieurs notions de réduction plus ou moins fortes.

2.3.1 LLL-réduction

Soient $\eta \in [1/2, 1[$ et $\delta \in]\eta^2, 1[$. Soit b une base d'un réseau L. On note b^* son orthogonalisation de Gram-Schmidt et $\mu_{i,j}$ les coefficients associés. La base b est (δ, η)-LLL-réduite si :

$$\begin{cases} \forall i \in [\![2, d]\!], \forall j \in [\![1, i-1]\!], |\mu_{i,j}| \leq \eta \\ \forall i \in [\![2, d]\!], \delta \|\mathbf{b_{i-1}^*}\|^2 \leq \|\mathbf{b_i^*} + \mu_{i,i-1} \mathbf{b_{i-1}^*}\|^2. \end{cases}$$

Historiquement les valeurs des paramètres sont $\delta = 3/4$ et $\eta = 1/2$. Plus δ est proche de 1 et plus la base est réduite, on utilisera donc $\delta = 0,99$ dans ce rapport.

On pose $\alpha = \frac{1}{\sqrt{\delta - \eta^2}}$ et on note λ le minimum de L. On peut montrer que si une base b est (δ, η)-LLL-réduite, alors la suite des $\|\mathbf{b_i^*}\|$ ne décroît pas trop vite, au sens où $\|\mathbf{b_i^*}\| \geq \|\mathbf{b_{i-1}^*}\|/\alpha$. Cela implique que $\mathbf{b_1}$ est relativement court par rapport au plus court vecteur : $\|\mathbf{b_1}\|/\lambda \leq \alpha^{d-1}$.

La LLL-réduction est une réduction plutôt faible mais peu coûteuse à obtenir. En effet, l'algorithme LLL [9] (du nom de ses auteurs A. Lenstra, H. Lenstra Jr. et L. Lovász) permet de construire une base LLL-réduite d'un réseau en temps polynomial. Pour un réseau donné par des vecteurs à coefficients entiers, sa complexité exacte est $O(d^5 n \log^3 B)$ où d est la dimension du réseau, n la dimension de de l'espace et B la norme du plus grand des vecteurs de la base.

2.3.2 HKZ-réduction

Une base HKZ-réduite d'un réseau de dimension d est une base telle que :
– $\mathbf{b_1}$ est un plus court vecteur du réseau.
– $\forall i > j, |\mu_{i,j}| \leq 1/2$.

– Si $d > 1$, les projections de $\mathbf{b_2}, \ldots, \mathbf{b_d}$ orthogonalement à $\mathbf{b_1}$ forment une base HKZ-réduite en dimension $d-1$.

La HKZ-réduction est beaucoup plus forte que la LLL-réduction. Trouver une base HKZ-réduite est calculatoirement équivalent à résoudre SVP.

2.3.3 BKZ-réduction

Une hiérarchie de réductions intermédiaires a été introduite par Schnorr [10] entre la LLL-réduction et la HKZ-réduction, ce sont les BKZ-réductions. Une BKZ-réduction est paramétrée par un entier k compris entre 2 et la dimension du réseau d. Prendre $k = 2$ correspond à la LLL-réduction et prendre $k = d$ correspond à la HKZ-réduction. La BKZ-réduction avec un valeur de k intermédiaire nécessite de HKZ-réduire des sous-réseaux de dimension k.

Ainsi, SVP et la réduction de réseaux sont donc deux problèmes intriqués : résoudre SVP nécessite de trouver une base réduite et réduire efficacement une base nécessite de résoudre SVP (en dimension inférieure). Pour appliquer KFP à une base BKZ-réduite, il faut donc l'utiliser récursivement. Néanmoins, on peut appliquer KFP à une base seulement LLL-réduite. Dans ce cas, le problème ne se pose pas mais l'algorithme KFP est plus lent.

3 L'algorithme KFP pour SVP

Il existe plusieurs méthodes pour résoudre SVP :

– L'algorithme AKS [2] est théoriquement le plus efficace. C'est un algorithme probabiliste de complexité $2^{O(d)}$. Il utilise aussi un espace $2^{O(d)}$ et ne semble donc pas utilisable en pratique.
– L'algorithme KFP a une complexité $2^{O(d^2)}$ quand il est appliqué à une base LLL-réduite. Il n'utilise qu'un espace polynomial.
– L'algorithme de Kannan utilise KFP récursivement. La meilleure borne de complexité connue pour l'algorithme de Kannan est $d^{\frac{d}{2e}+o(d)}$ [6] (ce qui est plus efficace que $2^{O(d^2)}$). Cependant, il est moins efficace en pratique que l'algorithme KFP simple pour les dimensions actuellement atteignables.

Les coûts donnés pour ces algorithmes sont exprimés en nombre d'opérations arithmétiques. En fait, chacune de ces opérations a un coût polynomial en $\log B$, où $\log B$ est la taille en bits du plus grand vecteur de la base. Cela multiplie les coûts précédents par $\mathrm{Poly}(\log B)$.

3.1 Principe

Soient $b = (\mathbf{b_1}, \ldots, \mathbf{b_d})$ une base LLL-réduite d'un réseau et $A > 0$. L'algorithme KFP énumère tous les vecteurs du réseau de norme plus petite que $\sqrt{A}$ et renvoie le plus court vecteur non nul. En prenant A assez grand (par exemple $A = \|\mathbf{b_1}\|^2$), on est certain qu'il trouve une solution.

On note b^* l'orthogonalisation de Gram-Schmidt de b et $\mu_{i,j}$ les coefficients associés. Tout vecteur du réseau s'écrit sous la forme $\sum_{i=1}^{d} x_i \mathbf{b_i}$ avec $x_i \in \mathbb{Z}$. Si ce vecteur est plus court que

$\sqrt{A}$, il vérifie :

$$\begin{aligned} \|\sum_{i=1}^{d} x_i \mathbf{b_i}\|^2 &\leq A \\ \|\sum_{i=1}^{d} x_i(\mathbf{b_i}^* + \sum_{j=1}^{i-1} \mu_{i,j}\mathbf{b_j^*})\|^2 &\leq A \qquad \text{(d'après (1))} \\ \sum_{i=1}^{d}(x_i + \sum_{j=i+1}^{d} \mu_{j,i}x_j)^2\|\mathbf{b_i^*}\|^2 &\leq A \qquad (2) \end{aligned}$$

On pose $r_i = \|\mathbf{b_i}\|^2$, $c_i = -\sum_{j=i+1}^{d} \mu_{j,i}x_j$ et $y_i = x_i - c_i$ (c_d est bien défini et vaut 0). En utilisant ces nouvelles notations, l'équation (2) équivaut à :

$$\sum_{i=1}^{d} y_i^2 r_i \leq A.$$

Cette équation implique :

$$\begin{aligned} y_d^2 r_d &\leq A \\ y_{d-1}^2 r_{d-1} &\leq A - y_d^2 r_d \\ &\vdots \\ y_1^2 r_1 &\leq A - \sum_{j=2}^{d} y_j^2 r_j \end{aligned}$$

Le principe de l'algorithme KFP est d'énumérer les $y_d, \ldots, y_1$ qui satisfont ces équations et correspondent à des points du réseau, en fixant, dans l'ordre, $y_d, \ldots, y_1$. Simultanément, on calcule les valeurs des x_i et des c_i. De manière générale, voici comment procéder pour énumérer les valeurs possibles de y_i une fois que $y_d, y_{d-1}, \ldots, y_{i+1}$ sont fixés :

- On calcule $c_i = -\sum_{j=i+1}^{d} \mu_{j,i}x_j$ (les x_j sont déjà connus pour $i < j \leq d$).
- De l'équation $y_i^2 r_i \leq A - \sum_{j=i+1}^{d} y_j^2 r_j$, on déduit

$$|y_i| \leq \sqrt{(A - \textstyle\sum_{j=i+1}^{d} y_j^2 r_j)/r_i}.$$

- On a $x_i = y_i + c_i$, or x_i est entier donc $y_i + c_i \in \mathbb{Z}$.
- La combinaison de ces deux équations permet d'obtenir les valeurs possibles de y_i et les valeurs de x_i correspondantes. Il y en a au plus $2\sqrt{A/r_i} + 1$.

L'algorithme KFP peut être considéré comme un parcours d'arbre en profondeur. Chaque nœud de l'arbre est identifié par sa hauteur i et sa position $[x_i, \ldots, x_d]$. En fait, l'arbre possède très peu de feuilles au niveau $i = 1$. La plupart des branches s'arrêtent à un niveau intermédiaire. En effet, il arrive fréquemment que pour un nœud $(i, [x_i, \ldots, x_d])$ avec i supérieur à 1, il n'existe aucun y_i tel que $|y_i| \leq \sqrt{(A - \sum_{j=i+1}^{d} y_j^2 r_j)/r_i}$ et $y_i + c_i \in \mathbb{Z}$. Dans ce cas là, ce nœud n'a pas de descendant et l'algorithme doit remonter avant d'avoir fixé toutes les coordonnées. Cela signifie que la complexité de l'algorithme vient principalement du fait qu'on

perd beaucoup de temps à explorer des impasses, et pas du fait qu'il y a beaucoup de vecteurs plus courts que $\sqrt{A}$.

Pour $x_{i+1}, \ldots, x_d$ fixés, l'ordre dans le lequel on considère les y_i possibles n'a pas d'importance pour le fonctionnement de l'algorithme. Mais en pratique, il est intéressant de considérer d'abord les y_i dont la valeur absolue est la plus faible. Cette heuristique, proposée par Schnorr et Euchner dans [11], permet souvent de trouver une première solution plus rapidement. De plus, certains de nos résultats nécessitent que l'ordre d'énumération soit celui fixé par cette heuristique.

Pour que l'algorithme renvoie un résultat correct, tous les calculs doivent être effectués avec une précision infinie. Si la base du réseau est à coefficients rationnels, toutes les variables manipulées sont des rationnels (ou des entiers) donc cela ne pose pas de problème théorique.

3.2 Algorithme KFP

Entrée : une borne A et les coefficients μ_{ij} et r_i.
Sortie : les coordonnées dans la base $(\mathbf{b_1}, \ldots, \mathbf{b_d})$ d'un plus court vecteur de L.

1 $x[1:d] = (1, 0, \ldots, 0), \Delta x[1:d] = (1, 0, \ldots, 0), \Delta^2 x[1:d] = (1, -1, \ldots, -1)$;
2 $c[1:d] = (0, \ldots, 0), \ell[1:d+1] = (0, \ldots, 0), y[1:d] = (0, \ldots, 0)$;
3 $i = 1, \mathbf{sol} = \emptyset$;
4 **repeat**
5 $y_i = |x_i - c_i|$; $\ell_i = \ell_{i+1} + r_i y_i^2$;
6 **if** $\ell_i \leq A$ *and* $i = 1$ **then** $(\mathbf{sol}, A) =$ évaluer$(\mathbf{sol}, A, x, \ell_1)$;
7 **if** $\ell_i \leq A$ *and* $i > 1$ **then**
8 $i = i - 1$;
9 $c_i = -\sum_{j=i+1}^{d} x_j \mu_{ji}$;
10 $x_i = \lfloor c_i \rceil$; $\Delta x_i = 0$;
11 **if** $c_i < x_i$ **then** $\Delta^2 x_i = 1$ **else** $\Delta^2 x_i = -1$;
12 **else**
13 $i = i + 1$;
14 **if** $i > d$ **then return sol**;
15 $\Delta^2 x_i = -\Delta^2 x_i$; $\Delta x_i = -\Delta x_i + \Delta^2 x_i$; $x_i = x_i + \Delta x_i$;
16 **end**
17 **end**

Chaque itération de boucle correspond à l'exploration d'un nœud de l'arbre $(i, [x_i, \ldots, x_d])$. Dans cet algorithme, la fonction `évaluer` n'est pas définie. On en utilise en fait deux versions. La version la plus simple et la plus efficace est :

$$\texttt{évaluer}_1(\mathbf{sol}, A, x, \ell_1) = (\mathbf{x}, \ell_1).$$

À chaque solution trouvée, la borne A est remplacée par la norme de cette solution. Ainsi, on est certain que si la fonction est appelée à nouveau, le vecteur de coordonnées $\mathbf{x}$ est plus court que la solution précédente. On aura besoin par la suite de considérer une seconde variante définie ainsi :

$$\texttt{évaluer}_2(\mathbf{sol}, A, x, \ell_1) = \begin{cases} (\mathbf{x}, A) \text{ si } \| \sum_{i \leq d} x_i \mathbf{b_i} \| \leq \| \sum_{i \leq d} \mathbf{sol}_i \mathbf{b_i} \| \\ (\mathbf{sol}, A) \text{ sinon.} \end{cases}$$

Contrairement à la version précédente, la borne A ne change jamais. Lorsqu'on trouve une solution, sa norme n'est pas forcément inférieure à la solution précédente, mais on compare les normes pour le savoir.

4 Algorithme flottant pour SVP

La complexité théorique de l'algorithme KFP est $2^{O(d^2)} \cdot \text{Poly}(\log B)$. En pratique, la valeur $2^{O(d^2)}$ qui représente le nombre d'itérations de la boucle est relativement faible. Pour donner un ordre d'idée, voici des valeurs moyennes pour 10 bases générées aléatoirement[1] et LLL-réduites.

d	20	25	30	35	40	45
Itérations	$9,6 \cdot 10^2$	$8,0 \cdot 10^3$	$8,4 \cdot 10^4$	$7,1 \cdot 10^5$	$1,5 \cdot 10^7$	$3,4 \cdot 10^8$

Le terme $\text{Poly}(\log B)$, qui représente la taille des nombres manipulés, vaut $O(d \cdot \log B)$. Typiquement, on souhaite réduire des bases où $\log B$ est grand (B est un nombre d'une centaine de bits au moins). Si d est de l'ordre 50, les calculs de l'algorithme effectués en précision arbitraire à l'aide de GMP peuvent être 10^5 fois plus lents que les calculs sur des nombres flottants en double précision.

En pratique, il serait donc bien plus efficace d'utiliser des variables flottantes dans l'algorithme KFP. Toutefois, cela semble risqué : en effet, le but de l'algorithme est de trouver une combinaison linéaire de vecteurs la plus courte possible. Pour une coordonnée donnée, cela revient à rechercher des combinaisons entières de flottants qui s'annulent, ce qui correspond précisément à une situation où on perd beaucoup de précision.

C'est cette idée que nous avons approfondie durant ce stage. L'essentiel de cette section consistera à démontrer et à exploiter le fait que la perte de précision est limitée lorsqu'on applique l'algorithme KFP à une base LLL-réduite.

4.1 Description

Pour accélérer l'algorithme KFP, on remplace toutes les variables rationnelles par des flottants : les coefficients de Gram-Schmidt $\mu_{i,j}$ et r_j, ainsi que c_i, y_i et ℓ_i. Les opérations (c'est-à-dire ici additions, soustractions et multiplications) sont faites en virgule flottante. La valeur renvoyée par l'algorithme est un vecteur d'entiers, cela ne change pas dans la version flottante. Attention, il y a un endroit ou les calculs rationnels ne sont pas remplacés par des calculs flottants : si on utilise la fonction `évaluer`$_2$, la comparaison des normes effectuée dans cette fonction est toujours faite de manière exacte.

Une itération de la boucle est identifiée de manière unique par les valeurs i et $(x_i, \ldots, x_d)$ au début de cette itération. Le couple $(i, [x_i, \ldots, x_d])$ est appelé état de la boucle. Soient $i \leq d$ et $x_i, \ldots, x_d \in \mathbb{Z}$. L'algorithme flottant et l'algorithme exact n'effectuent pas forcément les mêmes itérations, et même si c'est le cas, ils peuvent le faire dans un ordre différent. Il n'est donc pas pertinent de comparer les variables au même tour de boucle dans les deux algorithmes. Cependant, on peut comparer les valeurs des variables dans un état donné de la boucle. En effet, les valeurs de c_i, y_i et ℓ_i sont entièrement déterminées par l'état $(i, [x_i, \ldots, x_d])$. Elles

[1] La méthode de génération utilisée est la même que celle décrite dans la section 6.1.1

sont indépendantes de A, et peuvent être définies même s'il n'existe en réalité aucune itération correspondant à cet état.

Les variables de l'algorithme flottants sont représentées avec une notation particulière : $\bar{c}_i$, $\bar{y}_i$, $\bar{\ell}_i$, etc. L'écart entre les variables de l'algorithme flottant et celles de l'algorithme exact dans le même état est noté par le symbole Δ, par exemple $\Delta c_i = |\bar{c}_i - c_i|$.

A priori, les entrées $\mu_{i,j}$ et r_j ne sont pas représentables exactement en virgule flottante, et doivent être arrondies. L'erreur maximale sur ces entrées est fixée par la constante κ :

$$\kappa = \max\left(\max_{i>j} \frac{\Delta\mu_{i,j}}{\epsilon}, \max_i \frac{\Delta r_i}{r_i \cdot \epsilon}\right).$$

Cette constante est de l'ordre de 1. De manière optimale, si $\bar{\mu}_{i,j}$ et $\bar{r}_j$ sont obtenus en arrondissant les valeurs exactes calculées en rationnels, on peut prendre $\kappa = 1/2$. En pratique, on peut calculer $\bar{\mu}_{i,j}$ et $\bar{r}_j$ plus efficacement si on autorise une valeur légèrement plus grande.

Comme les opérations flottantes ne sont pas associatives, l'ordre des opérations doit être défini :

- À la ligne 5 de l'algorithme, le calcul de ℓ_i est parenthésé $\ell_{i+1} \oplus (r_i \otimes (y_i \otimes y_i))$.
- À la ligne 9, le calcul de c_i est parenthésé $(x_{i+1} \otimes \bar{\mu}_{i+1,i}) \oplus [(x_{i+2} \otimes \bar{\mu}_{i+2,i}) \oplus [\ldots \oplus (x_d \otimes \bar{\mu}_{d,i}) \ldots]]$.

4.2 Résultats

Soit $b = (\mathbf{b_1}, \ldots, \mathbf{b_d})$ une base LLL-réduite d'un réseau avec les paramètres $\eta \in [1/2, 1[$ et δ. On suppose $\delta > \eta^2 + \frac{1}{(1+\eta)^2}$ (cette contrainte, plus forte que $\delta > \eta^2$, évite de faire une étude de cas pour des valeurs de (δ, η) qui n'ont pas d'intérêt pratique). On reprend la notation $\alpha = \frac{1}{\sqrt{\delta - \eta^2}}$ et on définit $\rho = \alpha(1+\eta)$. On note λ le minimum du réseau. La constante κ définie au 4.1 détermine l'erreur sur les entrées $\bar{\mu}_{i,j}$ et $\bar{r}_j$. On pose $R = (1 + \kappa\epsilon) \cdot \max r_i$, qui majore les r_i ainsi que les $r_i + \Delta r_i$.

On définit les constantes suivantes, qui servent à fournir des bornes explicites dans les théorèmes. Pour donner un ordre d'idée, C_1, C_2 et C_3 sont de l'ordre de 1 et ϵ' est de l'ordre de $\rho^d\epsilon$ (on verra que c'est aussi l'ordre de grandeur de l'erreur totale).

$$C_1 = \frac{\kappa+2}{\alpha-1} + \frac{\kappa+4}{\rho-1}, \qquad C_2 = \frac{2\alpha}{1+\eta-\alpha}\left(2+\kappa+2C_1\right),$$
$$\epsilon' = 2\frac{R}{r_1}\left[(1+\kappa)\alpha^{2d} + (2C_1+C_2)\rho^d\right]\cdot\epsilon, \qquad C_3 = C_2\frac{K+d\epsilon}{1-\epsilon'}(2+d\epsilon).$$

4.2.1 Algorithme certifié

Théorème 1. *On considère l'algorithme KFP flottant avec la fonction* `évaluer`$_2$ *(qui compare les normes exactement). On suppose que :*

1. $\max(\kappa, C_1\rho^d)\epsilon \leq 0,01$.
2. $A \geq (1+2d\epsilon)\lambda^2 + C_2\rho^d\epsilon \cdot R$.

Alors l'algorithme renvoie un plus court vecteur, autrement dit $\|\sum_{i\leq d} \mathbf{sol}_i\mathbf{b_i}\| = \lambda$

Remarques.

– *Pour satisfaire l'hypothèse 1 du théorème, il suffit de rendre ϵ suffisamment petit, c'est-à-dire d'utiliser des flottants d'assez grande précision.*
– *Le minimum λ est inconnu au départ mais on sait que $\lambda^2 \leq r_1$. Donc pour satisfaire l'hypothèse 2, on peut choisir $A = (1 + 2d\epsilon)r_1 + C_2\rho^d\epsilon \cdot R$. Dans l'algorithme exact, on aurait pris $A = r_1$, ce qui n'est pas beaucoup plus petit.*

Quand une nouvelle solution de norme $\gamma < \sqrt{A}$ est trouvée, la variante $\texttt{évaluer}_1$ remplace A par γ^2 alors que la fonction $\texttt{évaluer}_2$ ne diminue pas la borne A. Cela évite d'écarter des solutions très légèrement plus courtes que γ dont la norme serait surévaluée par l'algorithme flottant. Par contre, l'efficacité de l'algorithme avec la seconde variante est significativement réduite. En fait, il est possible de diminuer A dans l'algorithme certifié à condition de prendre des précautions. Il suffit d'appliquer à nouveau le théorème, c'est-à-dire de prendre $A = (1 + 2d\epsilon)\gamma^2 + C_2\rho^d\epsilon \cdot R$ quand on trouve une nouvelle solution.

L'algorithme certifié par ce théorème ne peut pas être utilisé dans certaines circonstances. En effet, la fonction $\texttt{évaluer}_2$ calcule la norme de $\sum_{i \leq d} x_i \mathbf{b_i}$ en rationnels. Cela n'est possible que si on connaît les vecteurs $\mathbf{b_i}$. Or on ne dispose parfois que des coefficients de Gram-Schmidt approchés $\bar{\mu}_{i,j}$ et $\bar{r}_j$. C'est le cas par exemple lorsqu'on applique l'algorithme KFP à l'intérieur d'un algorithme de BKZ-réduction : la base exacte n'est pas calculée.

4.2.2 Analyse de l'algorithme non certifié

L'algorithme flottant sans vérification de la solution en rationnels produit toujours une solution quasi-optimale. Ce résultat est utile quand on ne peut pas utiliser l'algorithme certifié.

Théorème 2. *On considère l'algorithme KFP flottant avec la fonction $\texttt{évaluer}_1$. On suppose que $\epsilon' \leq 0,01$. Si l'algorithme renvoie une solution* **sol**, *la norme $\gamma = \| \sum_{i \leq d} \mathbf{sol}_i \mathbf{b_i} \|$ de cette solution vérifie :*

$$\lambda^2 \leq \gamma^2 \leq (1 + 4d\epsilon) \cdot \lambda^2 + C_3 \max\left(1, \frac{A}{r_1}\right) \rho^d \epsilon \cdot R.$$

Remarques.
– *Pour satisfaire $\epsilon' \leq 0,01$, il suffit de prendre ϵ assez petit.*
– *Il suffit de prendre $A \geq \bar{r}_1$ pour garantir que l'algorithme renvoie une solution.*

4.2.3 Surcoût des algorithmes flottants

L'algorithme certifié est plus coûteux que l'algorithme non certifié pour deux raisons. Premièrement, la borne A initiale est plus grande. Ce surcoût est quantifiable puisque l'augmentation de A est donnée de façon explicite. Il est négligeable dès que la précision est assez grande. Deuxièmement, les normes des solutions trouvées doivent être recalculées exactement. Il est possible d'estimer ce surcoût : au plus $2^{O(d)}$ normes de vecteurs doivent être calculées en rationnels. Cela est négligeable par rapport à la complexité $2^{O(d^2)}$ totale. De plus, en pratique, la situation est bien meilleure. À condition d'appliquer la remarque de la section 4.2.1 pour diminuer la borne A à chaque nouvelle solution, le nombre total de normes rationnelles à calculer est de l'ordre d'une dizaine. En général, ce surcoût est donc totalement négligeable. Certains types de réseaux pour lesquels le nombre de plus courts vecteurs est exponentiel en la dimension peuvent quand même poser problème.

Un autre point à analyser est que, si les algorithmes flottants sous-estiment systématiquement les normes des vecteurs, ils peuvent effectuer plus d'itérations que l'algorithme exact avec la même borne. Le résultat suivant montre que l'augmentation éventuelle du nombre d'itérations est très limitée.

Théorème 3. *Si $\epsilon' \leq 0,01$, alors l'algorithme en virgule flottante avec la borne A et n'importe quelle fonction d'évaluation effectue moins d'itérations que l'algorithme exact avec la même fonction d'évaluation et en entrée la borne :*

$$A' = (1 + d\epsilon) \cdot A + C_3 \max\left(1, \frac{A}{r_1}\right) \rho^d \epsilon \cdot R.$$

D'un point de vue théorique, ces résultats prouvent que l'algorithme flottant est plus efficace que l'algorithme exact. La complexité de l'algorithme KFP passe de $2^{O(d^2)} \cdot \text{Poly}(\log B)$ pour la version exacte à $2^{O(d^2)} + (2^{O(d)} \cdot \text{Poly}(\log B))$ pour la version exacte ($\log B$ est la taille en bits du plus grand vecteur de la base). On peut également améliorer l'analyse de complexité de l'algorithme de Kannan, qui repose sur KFP. La borne $d^{\frac{d}{2e}+o(d)} \cdot \text{Poly}(\log B)$ devient $d^{\frac{d}{2e}+o(d)} + (d^{o(d)} \cdot \text{Poly}(\log B))$ en utilisant l'algorithme flottant.

4.3 Analyse d'erreur et preuves des théorèmes

Pour obtenir les résultats présentés dans la partie 4.2, la principale difficulté est de borner l'erreur sur la variable $\bar{\ell}_i$ dans l'algorithme flottant dans un état donné.

Cette borne d'erreur est traduite par deux lemmes. La différence essentielle entre ces deux lemmes se situe dans les hypothèses. Dans le premier cas, la borne d'erreur dépend de ℓ_i (par l'hypothèse $n(\mathbf{x}) \leq r_1$) alors que dans le second cas, elle dépend de $\bar{\ell}_i$ (proportionellement).

Pour tout $\mathbf{x} \in \mathbb{Z}^d$, on pose $n(\mathbf{x}) = \| \sum_{i \leq d} x_i \mathbf{b_i} \|^2$. On définit $\sigma = (1, [x_1, \ldots, x_d])$. La quantité $n(\mathbf{x})$ est égale à la valeur de ℓ_1 quand l'algorithme exact de trouve dans l'état σ. On note $\bar{n}(\mathbf{x})$ la valeur de $\bar{\ell}_1$ quand l'algorithme flottant se trouve dans l'état σ. C'est une approximation de $n(\mathbf{x})$.

Lemme 1. *On suppose que* $\max(\kappa, C_1\rho^d) \cdot \epsilon \leq 0,01$. *Soit* $\mathbf{x} \in \mathbb{Z}^d$. *Si* $n(\mathbf{x}) \leq r_1$, *alors :*

$$\bar{n}(\mathbf{x}) \leq (1 + 2d\epsilon) \cdot n(\mathbf{x}) + C_2 \rho^d \epsilon \cdot R.$$

Lemme 2. *On suppose que* $\epsilon' \leq 0,01$. *Soient* $\mathbf{x} \in \mathbb{Z}^d$ *et* $i \leq d$. *On considère les algorithmes flottant et exact dans l'état* $(i, [x_i, \ldots, x_d])$. *Alors :*

$$\ell_i \leq (1 + d\epsilon) \cdot \bar{\ell}_i + C_2 \max\left(1, \frac{\bar{\ell}_i(K + d\epsilon)}{r_1(1 - \epsilon')}\right) \rho^d \epsilon \cdot R.$$

Les preuves de ces deux lemmes sont données dans l'annexe A.

4.3.1 Preuves des théorèmes à partir des lemmes

Avant de prouver les théorèmes, nous avons besoin d'une propriété générale sur l'algorithme flottant.

Lemme 3. *Si on utilise la fonction* `évaluer`$_1$ *dans l'algorithme flottant, alors l'algorithme en virgule flottante appelle cette fonction pour tous les vecteurs* $\mathbf{x} \in \mathbb{Z}^d$ *tels que* $\bar{n}(\mathbf{x}) \leq A$.

Démonstration. Soit $\mathbf{x} \in \mathbb{Z}^d$ tel que $\bar{n}(\mathbf{x}) \leq A$. On montre par récurrence sur i, pour i décroissant de d à 1 que l'algorithme flottant effectue une itération dans l'état $(i, [x_i, \ldots, x_d])$ et qu'à cette itération, le test $\bar{\ell}_i \leq A$ est positif.

Soit $i \leq d$. La variable $\bar{c}_i$ peut être définie uniquement en fonction de $(x_{i+1}, \ldots, x_d)$. On considère la suite des états de la forme $\sigma_t = (i, [X^{(t)}, x_{i+1}, \ldots, x_d])$ où $X^{(t)}$ est le t-ième entier le plus proche de $\bar{c}_i$ (en commençant par $X^{(1)} = \lfloor c_i \rceil$). Au moins le premier de ces états, σ_1, correspond à une itération réellement effectuée : c'est immédiat si $i = d$ et c'est vrai grâce à l'hypothèse de récurrence si $i < d$. Si l'algorithme considère d'autres états de cette suite, il les considère dans l'ordre de la suite et sans saut (car l'ordre d'énumération correspond à l'heuristique de Schnorr et Euchner), et s'arrête au premier état σ_t tel que $\bar{\ell}_i(\sigma_t) > A$.

Or par définition de $X^{(t)}$, la suite des $(|X^{(t)} - c_i(\sigma_t)|)_{t \geq 1}$ est croissante. Comme la fonction arrondi est croissante, la suite $(\bar{y}_i(\sigma_t))_{t \geq 1}$ est croissante, ainsi que la suite $(\bar{\ell}_i(\sigma_t))_{t \geq 1}$. Dans la suite des $(X^{(t)})$, il existe un t tel que $X^{(t)} = x_i$. Par hypothèse $\bar{n}(\mathbf{x}) \leq A$, donc $\bar{\ell}(\sigma_t) \leq A$. Cet état est donc examiné par l'algorithme et le test est bien satisfait. □

On peut maintenant démontrer les trois théorèmes.

Démonstration du théorème 1. Soit $\mathbf{x} \in \mathbb{Z}^d$ le vecteur de coordonnées d'un plus court vecteur de L. Il vérifie $n(\mathbf{x}) = \lambda^2$, donc d'après le lemme 1, on a :

$$\bar{n}(\mathbf{x}) \leq (1 + 2d\epsilon) \cdot \lambda^2 + C_2 \rho^d \epsilon \cdot R \leq A.$$

D'après le lemme 3, la fonction $\texttt{évaluer}_1$ est appelée sur $\mathbf{x}$. Comme elle fait des calculs exacts, l'algorithme trouve bien un plus court vecteur. □

Démonstration du théorème 2. Si on exécute l'algorithme avec la fonction $\texttt{évaluer}_2$, la variable A peut diminuer jusqu'à une valeur $A^{(fin)}$. L'algorithme aurait renvoyé le même résultat si on avait pris dès le début $A = A^{(fin)}$. On suppose que c'est le cas. Alors la valeur de A n'est pas modifiée au cours de l'algorithme.

Soit $\mathbf{x} \in \mathbb{Z}^d$ tel que $n(\mathbf{x}) = \lambda^2$, alors d'après le lemme 1 :

$$\bar{n}(\mathbf{x}) \leq (1 + 2d\epsilon) \cdot \lambda^2 + C_2 \rho^d \epsilon \cdot R.$$

Nécessairement, $A \leq (1 + 2d\epsilon) \cdot \lambda^2 + C_2 \rho^d \epsilon \cdot R$. Sinon, l'algorithme diminuerait A en trouvant $\mathbf{x}$. On considère le vecteur $\mathbf{sol} \in \mathbb{Z}^d$ renvoyé par l'algorithme. On a $A = \bar{n}(\mathbf{sol})$. En appliquant le lemme 2 :

$$n(\mathbf{sol}) \leq (1 + d\epsilon) \cdot A + C_2 \max\left(1, \frac{A(K + d\epsilon)}{r_1(1 - \epsilon')}\right) \rho^d \epsilon \cdot R.$$

□

Démonstration du théorème 3. On considère un état de l'algorithme $(i, [x_i, \ldots, x_d])$. On suppose que le résultat du test $\bar{\ell}_i \leq A$ est positif. On applique le lemme 2 et l'inégalité $\bar{\ell}_i \leq A$:

$$\ell_i \leq (1 + d\epsilon) \cdot A + C_2 \max\left(1, \frac{A(K + d\epsilon)}{r_1(1 - \epsilon')}\right) \rho^d \epsilon \cdot R \leq A'.$$

Donc l'algorithme exact avec la borne A' aurait exécuté une itération dans cet état et le test aurait également été positif. Il a exactement autant d'itérations où le test est négatif et $i < d$ que d'itérations où le test est positif et $i > 1$. D'où le résultat sur le nombre total d'itérations. □

4.4 Adaptation en virgule fixe

Les nombres en virgule fixe sont des réels de la forme $\pm m \cdot 2^{-p}$ où m est un entier borné (la mantisse) et p une constante (la précision). La différence avec la virgule flottante est qu'il n'y a pas d'exposant, ce qui restreint l'amplitude des nombres représentables.

La représentation en virgule fixe est peu utilisée sur les ordinateurs classiques, étant donné que la virgule flottante est à la fois plus pratique et plus efficace (car elle est implantée en matériel). Par contre, sur des architectures spécifiques qui ne disposent pas d'unité de calcul flottant, un algorithme en virgule fixe est beaucoup plus rapide. Étant données les applications en télécommunications et en cryptographie de l'algorithme KFP, il est susceptible d'être utilisé sur ce type d'architectures. L'avantage de la virgule fixe apparaît aussi pour une implantation entièrement en matériel de l'algorithme.

Les résultats démontrés sur l'algorithme en virgule flottante peuvent être adaptés à un algorithme en virgule fixe. La seule condition est que le plus grand coefficient r_i doit être normalisé à 1. Cela ne pose aucun problème car diviser tous les r_i par une constante ne change rien au résultat de l'algorithme. Si cette condition est respectée, on obtient des majorations d'erreur du même ordre de grandeur que pour la virgule flottante.

5 Résolution de CVP

Dans cette section, nous étudions la complexité de l'algorithme KFP exact pour CVP. Nous expliquons l'algorithme de Kannan pour CVP et montrons que l'algorithme KFP, plus simple, est en fait aussi efficace.

5.1 L'algorithme KFP pour CVP

Soient $b = (\mathbf{b_1}, \ldots, \mathbf{b_d})$ une base LLL-réduite, $\mathbf{t}$ un vecteur quelconque et $A > 0$. Pour simplifier, on suppose que $d = n$. La base et les coefficients de Gram-Schmidt sont notés respectivement b^* et $\mu_{i,j}$. Dans la variante de l'algorithme KFP pour résoudre CVP, au lieu d'énumérer les vecteurs $\mathbf{x}$ du réseau de norme inférieure à $\sqrt{A}$, on énumère les vecteurs du réseau tels que $\|\mathbf{x} - \mathbf{t}\| \leq \sqrt{A}$. Pour cela, on décompose $\mathbf{x}$ dans la base b sous la forme $\mathbf{x} = \sum_{i \leq d} x_i \mathbf{b_i}$ et $\mathbf{t}$ dans la base b^* sous la forme $\mathbf{t} = \sum_{i \leq d} t_i \mathbf{b_i^*}$. L'objectif est que $\mathbf{x}$ vérifie :

$$\| \sum_{i=1}^{d} x_i \mathbf{b_i} - \sum_{i=1}^{d} t_i \mathbf{b_i^*} \|^2 \leq A. \tag{3}$$

En suivant les même étapes que dans la version pour SVP, on aboutit à :

$$\sum_{i=1}^{d} y_i^2 r_i \leq A$$

avec encore $r_i = \|\mathbf{b_i}\|^2$, $y_i = x_i - c_i$, mais la définition de c_i est différente :

$$c_i = t_i - \sum_{j=i+1}^{d} \mu_{j,i} x_j. \tag{4}$$

Pour trouver les solutions, il suffit alors d'appliquer le même algorithme que pour SVP avec les nouveaux c_i. Dans l'algorithme lui-même, on modifie la ligne 9 où c_i est calculé pour correspondre à l'équation (4). De plus, l'algorithme prend en entrée les coefficients t_i et l'initialisation est différente :

$$\begin{cases} c_d = t_d \ , \quad x_d = \lfloor c_d \rceil \ , \quad \Delta x_d = 0 \\ \Delta^2 x_d = (1 \text{ si } c_d < x_d, -1 \text{ sinon}) \\ i = d \ , \quad \mathbf{sol} = \emptyset. \end{cases}$$

Revenons à SVP. Pour que l'algorithme trouve au moins une solution, il suffit de prendre $A = r_1$ (ou $A = \bar{r}_1$ dans la version flottante). La complexité de l'algorithme dépend du nombre de valeurs possibles pour x_i à chaque niveau i, qui est au plus $2\sqrt{A/r_i} + 1 = 2\sqrt{r_1/r_i} + 1$. Avec une base LLL-réduite en entrée, le rapport r_1/r_i est borné, donc la complexité totale de l'algorithme aussi. Dans le cas de CVP, la borne $A = r_1$ ne convient pas. L'algorithme débute au niveau $i = d$ et peut au mieux trouver une solution après d itérations, si le test $\ell_i \leq A$ est positif à chaque fois. Comme on utilise l'heuristique de Schnorr et Euchner, les premières valeurs de y_i sont bornées par $1/2$, on a :

$$\ell_i \leq \sum_{j=i}^{d} r_j y_j^2 \leq \sum_{j=i}^{d} r_j/4.$$

Pour que l'algorithme trouve au moins un vecteur, il suffit de prendre $A \geq \sum_{i \leq d} r_i/4$. En général, on ne sait pas faire mieux que cette borne. Dans le cas générique, la suite des r_i est décroissante, en particulier r_1 est plus grand que tous les autres coefficients, donc cette borne initiale n'est pas beaucoup plus grande que pour SVP. Cependant, pour des réseaux particuliers, la suite peut être croissante, et un r_j peut même être arbitrairement plus grand que r_1, donc A aussi. Dans ce cas, le rapport le rapport A/r_i n'est pas borné, donc le nombre de vecteurs du réseau qui satisfont l'équation (3) ne l'est pas non plus. On ne peut pas majorer la complexité de l'algorithme de manière satisfaisante de cette façon.

5.2 L'algorithme de Kannan pour CVP

Dans [8], Kannan donne une méthode pour résoudre CVP qui repose sur KFP mais permet d'éviter ces cas particuliers. Soit r_i le plus grand élément de la suite $r_1, \ldots, r_d$. Si $i = 1$, alors on applique l'algorithme KFP normalement en prenant $A = \sum_{j \leq d} r_j/4 \leq dr_1/4$. Dans ce cas, la complexité de l'algorithme ne pose pas de problème. Si $i > 1$, considérons le comportement de l'algorithme KFP. On prend $A = \sum_{j \leq d} r_j/4 \leq dr_i/4$. Le nombre de choix pour x_j pour $j \geq i$ est borné par $2\sqrt{A/r_j} + 1 \leq 2\sqrt{dr_i/r_j} + 1$, et r_i/r_j est borné par α^{j-i} car la base est LLL-réduite. Ce n'est que pour $j < i$ qu'il y aurait un problème. Plaçons nous à un instant où l'algorithme a fixé la valeur de $x_d, x_{d-1}, \ldots, x_i$. À ce stade, il reste à trouver $x_{i-1}, \ldots, x_1$ tels que :

$$\|(\textstyle\sum_{j=1}^{i-1} x_j \mathbf{b_j}) + (\sum_{j=i}^{d} x_j \mathbf{b_j}) - t\| \leq A.$$

Cela revient à résoudre une nouvelle instance de CVP en dimension $i-1$ avec $t' = t - \sum_{j=i}^{d} x_j \mathbf{b_j}$.

L'idée de l'algorithme de Kannan est d'exécuter l'algorithme KFP et d'interrompre l'exécution dès qu'on atteint le niveau i. À ce moment, l'algorithme de Kannan est appelé récursivement sur la nouvelle instance de CVP. Une fois que l'appel récursif est terminé, l'algorithme KFP reprend mais sans descendre en-dessous du niveau i. Ainsi, la complexité de l'algorithme KFP

partiel est bornée, et il y a au plus d niveaux de récusivité. Il est donc possible d'estimer la complexité totale. La meilleure borne connue est $d^{\frac{d}{2}+o(d)}\text{Poly}(\log B)$, elle est démontrée dans [6].

5.3 Nouvelle analyse de l'algorithme KFP

Nous avons prouvé grâce à une analyse plus précise de l'algorithme KFP que celui-ci a une complexité au moins aussi bonne que l'algorithme de Kannan. Cette analyse ne fonctionne que si on utilise l'heuristique de Schnorr et Euchner décrite à la section 3.1.

Supposons que le r_i est le maximum de la suite $r_1, \ldots, r_d$, avec $i \neq 1$. Pour simplifier, on suppose également que r_1 est le deuxième maximum de cette suite (donc dans l'algorithme de Kannan, il n'y a qu'un niveau de récursivité). On démarre l'exécution de KFP et on s'arrête lorsque les valeurs de $(x_d, \ldots, x_i)$ sont fixées. On pose $A' = A - \sum_{j=1}^{d} y_j^2 r_j$. À cet instant, continuer l'algorithme revient à résoudre CVP en dimension $i-1$ sur $(\mathbf{b_1}, \ldots, \mathbf{b_{i-1}})$ avec la cible $t' = t - \sum_{j=i}^{d} x_j \mathbf{b_j}$ et la borne A'. Mais A' peut être du même ordre de grandeur que A, c'est-à-dire beaucoup plus grand que r_1. Ce en quoi l'algorithme de Kannan se différentie de l'algorithme KFP, c'est qu'en faisant un appel récursif, on va essayer de résoudre le même problème CVP mais avec la borne $A'' = \sum_{j=1}^{i-1} y_j^2 r_j$ qui est du même ordre de grandeur que r_1.

Il y a alors deux cas. Ou bien $A' < A''$. Alors l'algorithme KFP énumère les valeurs de $x_{i-1}, \ldots, x_1$ plus vite que l'algorithme de Kannan. Ou bien $A'' \geq A'$. Mais en fait, la borne A'' est choisie de telle sorte qu'on trouve un premier vecteur dont la distance γ à $\mathbf{t}$ est inférieure à A'' après seulement d itérations. Comme $A' \geq A''$, l'algorithme KFP va trouver cette même solution en d itérations. Dans les deux cas, les bornes A' et A'' sont remplacées par γ dès le début de l'algorithme, et les deux algorithmes font ensuite exactement les mêmes itérations pour trouver les valeurs possibles $x_{i-1}, \ldots, x_1$. Ce qui conclut l'analyse.

C'est pourquoi contrairement à ce qui avait été prévu initialement, la résolution de CVP dans l'implémentation n'utilise pas l'algorithme de Kannan mais directement l'algorithme KFP.

6 Implémentations

Nous décrivons ici l'implémentation de KFP réalisée pendant ce stage, la méthode qui a été utilisée pour créer une version parallèle de cet algorithme et ce qui a été commencé sur la version FPGA.

6.1 LatEnum

Il existe peu d'implémentations de l'algorithme KFP. À ma connaissance, la seule disponible librement est celle de la bibliothèque NTL [12]. Elle est relativement efficace car elle fonctionne en virgule flottante, mais n'est pas certifiée et n'est pas utilisable directement (elle est intégrée à un algorithme de BKZ-réduction).

Pendant ce stage, j'ai réalisé un programme de résolution de SVP et CVP en C++, nommé LatEnum (pour LATtice ENUMeration). Le code source est disponible sous licence GPL à l'adresse `http://perso.ens-lyon.fr/xavier.pujol/latenum/`. Il contient environ 3000

lignes de code. Le programme repose sur une implémentation en virgule flottante de l'algorithme KFP. Pour SVP, le programme peut renvoyer une solution certifiée en utilisant les principes de la section précédente.

Le programme prend en entrée une base LLL-réduite. Pour résoudre SVP dans une base quelconque, il faut donc le combiner à un programme de LLL-réduction. Le code de LatEnum a été conçu pour être compatible avec fplll [4], une implémentation certifiée de l'algorithme LLL.

6.1.1 Amélioration des bornes d'erreur

Pour être le plus efficace possible, l'algorithme flottant doit pouvoir fonctionner avec une précision de flottants implantée sur le processeur. La plus grande précision standardisée est la double précision, ce qui correspond à 53 bits de mantisse. Or en utilisant les résultats précédents, la double précision ne suffirait que jusqu'à $d = 45$ environ. Mais en fait, pour une base générée aléatoirement, dans un sens particulier défini plus bas, le pire cas est loin d'être atteint dans les bornes d'erreur. Il est possible d'évaluer plus finement l'erreur en utilisant l'idée suivante : dans les preuves, les quantités $|\mu_{i,j}|$ et r_j sont systématiquement majorées par η et R, mais on peut ne pas faire ce remplacement si on considère une base fixée.

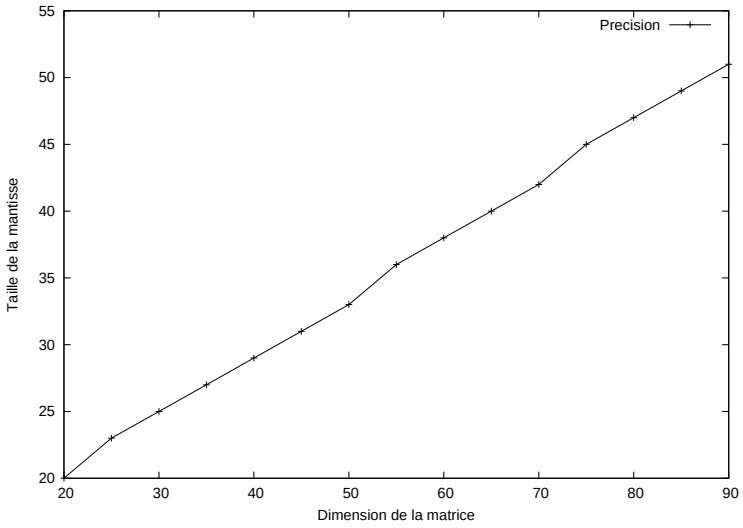
C'est pourquoi LatEnum n'utilise pas directement les résultats précédents, mais contient un module d'évaluation d'erreur en fonction de la base en entrée. La majoration d'erreur récursive faite dans les lemmes 1 et 2 correspond dans le programme à une boucle de d étapes. À chaque étape i, on calcule des majorations pour c_i, Δc_i, y_i, Δy_i, $\Delta \ell_i$ et x_i en utilisant les valeurs exactes de $\mu_{i,j}$ et r_j. Cette approche algorithmique permet aussi d'éviter certaines majorations de la preuve dont le seul but est de simplifier le résultat.

L'algorithme d'évaluation d'erreur est appelé au début et à chaque fois qu'une solution est trouvée dans l'algorithme. Il n'utilise que $O(d^2)$ opérations et son coût est négligeable par rapport au coût total de l'énumération.

Des estimations de la précision requise ont été réalisées à partir de bases générées aléatoirement. Pour générer des bases aléatoires, on produit des bases dites "sac-à-dos" dont la matrice est de la forme :

$$\begin{pmatrix} x_1 & 1 & 0 & \dots & 0 \\ x_2 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_d & 0 & 0 & \dots & 1 \end{pmatrix}$$

puis on applique l'algorithme LLL. Cela produit une meilleure distribution qu'en fixant directement chaque coefficient de la matrice aléatoirement (les instances de SVP générées par cette dernière méthode seraient en fait significativement plus faciles à résoudre). Le graphe suivant indique la précision maximale requise sur un échantillon de dix bases générées aléatoirement.



On constate que la double précision est suffisante jusqu'en dimension 90, alors que le temps de calcul devient prohibitif au-delà de la dimension 70 avec les ordinateurs actuels.

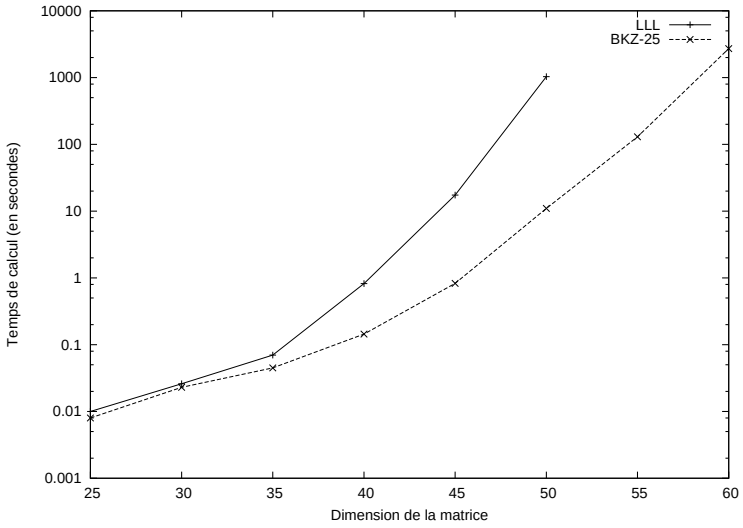
6.1.2 Algorithme certifié n'utilisant que les coefficients de Gram-Schmidt

Si on ne connaît que des valeurs approchées des coefficients de Gram-Schmidt, il est en général impossible de certifier le résultat. En effet, si deux vecteurs ont deux normes très proches, étant donnée l'incertitude sur les coefficients, il est impossible de comparer leurs normes.

Cependant, cette situation est rare. Pour la plupart des bases, il y a un écart significatif entre le plus court vecteur et le deuxième plus court vecteur (sans compter l'opposé du premier). Le programme détecte cette situation en calculant l'écart entre les normes des deux plus courts vecteurs trouvés et en utilisant les résultats des lemmes 1 et 2. En cas de réussite, il indique que le résultat est certifié. En cas d'échec, le pourcentage maximal d'erreur sur le plus court vecteur est affiché.

6.1.3 Résultats

La courbe suivante donne le temps d'exécution de l'algorithme sur des instances de SVP de dimensions variables. Chaque point du graphe représente un temps d'exécution moyen sur 10 bases générées aléatoirement. Cette courbe met en évidence la croissance superexponentielle de la complexité (l'échelle est logarithmique) et montre que l'algorithme est beaucoup plus rapide si la base en entrée est mieux réduite (c'est-à-dire BKZ-réduite au lieu de LLL-réduite).



Le temps d'exécution de l'algorithme a été comparé à celui de la version utilisée dans la bibliothèque NTL sur des matrices LLL-réduites de dimension 50. En moyenne, LatEnum est 1,8 fois plus rapide alors que les deux programmes implémentent le même algorithme. La différence est sans doute due au fait que la version de LatEnum minimise le nombre de tests effectués, et que la version de NTL fait un appel de fonction à chaque tour de boucle, ce qui empêche le compilateur de réaliser certaines optimisations. De plus, LatEnum fournit un résultat garanti, mais le temps d'exécution du code supplémentaire pour certifier la solution est en fait négligeable.

6.2 Version parallèle de l'algorithme KFP

L'algorithme KFP est un parcours d'arbre. L'exploration de sous-arbres distincts de cet arbre peut être faite en parallèle.

Par exemple, si les valeurs possibles pour x_d et x_{d-1} sont (0,0), (0,1), (0, 2), (1, -3) et (1, -4), la première machine recherche toutes les solutions telles que $x_d = 0$ et $x_{d-1} = 0$, la seconde recherche les solutions telles que $x_d = 0$ et $x_{d-1} = 1$, etc. Dans la suite, attribuer une tâche $(x_i, \ldots, x_d)$ à une machine signifie que la machine calcule les solutions de SVP dont les dernières coordonnées sont $x_i, \ldots, x_d$.

Le principe est simple. La difficulté consiste à diviser équitablement le travail entre les machines. Il n'est pas indispensable d'obtenir une division parfaite. Il suffit de préparer beaucoup plus de tâches que de machines, et de distribuer les tâches dynamiquement : dès qu'une machine termine une tâche, elle en reçoit une nouvelle. Si le temps maximum d'exécution d'une tâche est faible comparé au temps total d'exécution, alors la parallélisation est efficace.

Malgré cette souplesse, la division en tâches pose problème. La solution la plus simple, qui consiste à choisir un niveau i et à prendre comme tâches tous les $(x_i, \ldots, x_d)$ possibles ne fonctionne pas : si i est proche de d, le temps de traitement de la plus grosse tâche n'est pas assez petit par rapport au coût total. Si i est plus petit, le nombre de tâches augmente exponentiellement et le coût des communications devient trop grand. Il n'y a pas de compromis acceptable. Cela est dû au fait que les tâches sont de tailles très déséquilibrées.

Plutôt que de fixer par avance la répartition, on utilise un algorithme récursif pour diviser l'énumération en tâches. Cet algorithme considère toutes les valeurs de x_d possibles. Pour

chaque valeur de x_d, le coût de la tâche (x_d) est estimé : l'idée est de calculer approximativement le volume de l'hypersphère de rayon $A - \ell_d$ et de dimension $d - 1$ (celle dans laquelle l'algorithme énumère les points) multiplié par la densité du réseau, ce qui donne $\frac{\pi^{(d-1)/2} A^{d-1}}{\Gamma((d-1)/2+1) \cdot r_1 \ldots r_{d-1}}$. Plus précisément, il faut prendre le volume de la plus grande hypersphère de rayon $A - \ell_d$ et de dimension $d - i$ multipliée par la densité du réseau engendré par les projetés de $(\mathbf{b_i}, \ldots, \mathbf{b_{d-1}})$ orthogonalement à $(\mathbf{b_1}, \ldots, \mathbf{b_{i-1}})$. Les tâches dont le coût est inférieur au seuil critique sont envoyées aux machines, les autres sont divisées récursivement selon la valeur de x_{d-1}, x_{d-2}, etc.

En dehors de la distribution des tâches, le seul aspect centralisé de l'algorithme est la gestion de la borne A. Quand une machine trouve une nouvelle solution, la borne A globale est diminuée et toutes les autres machines en sont informées dès qu'elles commencent une nouvelle tâche.

6.2.1 Implémentation

L'algorithme parallèle a été implémenté en utilisant MPI. Le code de l'énumération est partagé avec celui de la version non parallèle. Le code pour la parallélisation elle-même fait environ 600 lignes. Une option permet de produire des résultats certifiés en utilisant le même mécanisme que dans la version non parallèle. Deux optimisations seraient possibles : d'une part, informer immédiatement les machines quand la borne est mise à jour au lieu d'attendre qu'elles finissent leur tâche et d'autre part utiliser des communications non bloquantes et les recouvrir par des calculs. Mais en choisissant convenablement le seuil critique pour la taille des tâches, le gain apporté serait marginal.

Des tests du programme ont été effectués avec sur des machines Intel Core 2 Duo à 1,86 GHz. L'implémentation de MPI utilisée est MPICH 1.2.7. La taille critique des tâches a été fixée à 10^7 itérations. Pour un ensemble de matrices de dimension 50 à 60, en utilisant 10 machines (plus une machine maître qui ne calcule pas), le facteur d'accélération moyen est de 9,7 (soit une efficacité 0,97). De plus, la majeure partie du temps perdu provient de l'initialisation de MPI, qui devient négligeable si on résout plusieurs problèmes à la suite. L'efficacité du programme est donc quasiment optimale.

6.3 Implantation sur FPGA

Le développement d'une implantation matérielle de l'algorithme KFP a été entamé. Actuellement, le code pour l'algorithme lui-même est écrit, mais la gestion des entrées/sorties n'est pas faite. D'autres implantations matérielles ont déjà été réalisées, en particulier [3], mais elles sont destinées seulement à des applications en télécommunications qui n'utilisent que des réseaux de très petite dimension. Le but de cet implantation est de créer un circuit résolvant SVP ou CVP au-delà des dimensions actuellement accessibles, c'est-à-dire plus que 70.

L'architecture cible est un FPGA, ce qui signifie *Field-Programmable Gate Array*. Un FPGA est un composant électronique que l'on peut configurer pour qu'il réalise n'importe quel circuit. Il fonctionne grâce à un ensemble de LUT (Look-Up Table), qui se comportent comme des portes logiques programmables, le réseau de routage entre les LUT étant également programmable. Cette architecture fait qu'un circuit sur FPGA est plus lent qu'un circuit intégré, mais l'avantage du FPGA est qu'il est reprogrammable à volonté, donc la réalisation d'un circuit sur FPGA revient beaucoup moins cher.

La version matérielle de l'algorithme utilise l'algorithme KFP en virgule fixe. Le langage utilisé pour la description du circuit est VHDL, mais le code VHDL est généré par un programme C++ pour une dimension de réseau spécifiée. Il ne semble pas possible pour l'instant de réaliser une implantation certifiée car un circuit ayant la précision requise serait trop gros pour tenir sur un FPGA.

Bien que les FPGA soient relativement lents, la version matérielle de KFP peut être rendue beaucoup plus efficace que la version logicielle en exploitant le parallélisme du matériel. Le bloc principal du circuit permet de calculer une itération de boucle de l'algorithme. Ce bloc est pipeliné sur n niveaux, n dépendant de la dimension du réseau. En divisant l'énumération en tâches comme pour la version parallèle, le bloc peut traiter n tâches en parallèle. À chaque cycle d'horloge, une itération de boucle est terminée pour une des tâches.

7 Conclusion

Pendant ce stage, nous avons prouvé plusieurs résultats sur l'algorithme KFP en virgule flottante, ce qui a permis notamment de garantir le résultat renvoyé par l'algorithme KFP pour SVP. Nous avons mis en pratique ces résultats dans une implémentation qui est ainsi certifiée et dont la complexité est connue. En pratique, le surcoût dû à la certification de la solution est négligeable.

En associant ce code avec fplll, une version certifiée de LLL reposant sur l'arithmétique flottante, de nombreux algorithmes sur les réseaux peuvent être rendus rigoureux. Cela s'applique en particulier aux algorithmes de HKZ-réduction et de BKZ-réduction, qui utilisent à la fois LLL et KFP.

Pour finir, une méthode de parallélisation a été mise au point et implémentée. Le facteur d'accélération obtenu est quasiment optimal.

Il reste plusieurs travaux à terminer. Le premier est l'implantation de l'algorithme sur FPGA. Le second est la certification de CVP dans le programme. Il est possible d'appliquer la même analyse que pour SVP, mais il faut tenir compte de certains réseaux particuliers, pour lequels la borne d'erreur obtenue peut être beaucoup plus mauvaise.

Enfin, des tests effectués sur l'algorithme en utilisant la virgule fixe avec une faible précision tendent à montrer que même la borne d'erreur adaptative du programme est trop grande. En dimension 40, utiliser seulement 16 bits de précision suffit pour obtenir un résultat correct avec toutes les bases testées. Améliorer la borne d'erreur aurait peu d'intérêt pour l'implémentation logicielle, puisqu'il est inutile de descendre en-dessous de la double précision. Par contre, cela pourrait servir à certifier une implantation matérielle de l'algorithme.

Remerciements

Je tiens à remercier Damien Stehlé pour m'avoir guidé et aidé tout au long de ce stage, ainsi que tous les membres de l'équipe Arénaire.

Références

[1] IEEE Standards Committee 754. ANSI/IEEE standard 754-1985 for binary floating-point arithmetic. Reprinted in SIGPLAN Notices, 22(2):9–25, 1987.

[2] M. Ajtai, R. Kumar, and D. Sivakumar. A sieve algorithm for the shortest lattice vector problem. In *Proceedings of the 33rd Symposium on the Theory of Computing (STOC 2001)*, pages 601–610. ACM Press, 2001.

[3] A. Burg, M. Borgmann, M. Wenk, M. Zellweger, W. Fichtner, and H. Bölcskei. VLSI implementation of mimo detection using the sphere decoding algorithm. *IEEE Journal of Solid State Circuits*, 40(7):1566–1577, 2005.

[4] D. Cadé and D. Stehlé. fplll-2.1, a floating-point LLL implementation. Available at `http://perso.ens-lyon.fr/damien.stehle`.

[5] U. Fincke and M. Pohst. A procedure for determining algebraic integers of given norm. In *Proceedings of EUROCAL*, volume 162 of *Lecture Notes in Computer Science*, pages 194–202. Springer-Verlag, 1983.

[6] G. Hanrot and D. Stehlé. Improved analysis of Kannan's shortest lattice vector algorithm (extended abstract). In *Proceedings of Crypto 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 170–186. Springer-Verlag, 2007.

[7] J. Hoffstein, J. Pipher, and J. H. Silverman. NTRU: a ring based public key cryptosystem. In *Proceedings of the 3rd Algorithmic Number Theory Symposium (ANTS III)*, volume 1423 of *Lecture Notes in Computer Science*, pages 267–288. Springer-Verlag, 1998.

[8] R. Kannan. Improved algorithms for integer programming and related lattice problems. In *Proceedings of the 15th Symposium on the Theory of Computing (STOC 1983)*, pages 99–108. ACM Press, 1983.

[9] A. K. Lenstra, H. W. Lenstra, Jr., and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:513–534, 1982.

[10] C. P. Schnorr. A hierarchy of polynomial lattice basis reduction algorithms. *Theoretical Computer Science*, 53:201–224, 1987.

[11] C. P. Schnorr and M. Euchner. Lattice basis reduction: improved practical algorithms and solving subset sum problems. *Mathematics of Programming*, 66:181–199, 1994.

[12] V. Shoup. NTL, Number Theory C++ Library. Available at `http://www.shoup.net/ntl/`, 1990.

Annexe A - Preuves des lemmes 1 et 2

La première partie de la démonstration des lemmes 1 et 2 repose sur des techniques classiques d'analyse d'erreur. Cette analyse est découpée en trois lemmes techniques. En utilisant cette analyse, la démonstration du lemme 1 est quasi-immédiate et celle du lemme 2 nécessite quelques idées supplémentaires.

7.1 Lemmes techniques intermédiaires

Lemme 4. *On suppose que* $\max(\kappa, C_1\rho^d)\cdot\epsilon \leq 0,01$. *On considère une itération de l'algorithme flottant, on se place après la ligne 5. Si* $\forall j \in [\![i+1, d]\!], y_i \leq \nu\alpha^{i-1}$ *pour une constante* $\nu \geq 1$, *alors :*

$$\begin{aligned} \Delta c_i &\leq C_1\nu\alpha^d(1+\eta)^{d-i}\epsilon \\ \Delta y_i &\leq y_i\epsilon/2 + KC_1\nu\alpha^d(1+\eta)^{d-i}\epsilon. \end{aligned}$$

Lemme 5. *À la ligne 5 de l'algorithme flottant, on a :*

$$|(\bar{y}_i \otimes \bar{y}_i) \otimes \bar{r}_i - r_i y_i^2| \leq RK^2[(\kappa+1)y_i^2\epsilon + (2y_i + \Delta y_i)\Delta y_i].$$

Lemme 6. *On suppose que* $\max(\kappa, C_1\rho^d)\cdot\epsilon \leq 0,01$. *On considère une itération de l'algorithme flottant, on se place après la ligne 5. Si* $\forall j \in [\![i, d]\!], y_i \leq \nu\alpha^{i-1}$ *pour une constante* $\nu \geq 1$, *alors :*

$$\begin{cases} \Delta\ell_i \leq d\epsilon \cdot \ell_i + C_2\nu^2\rho^d\epsilon \cdot R \\ \Delta\ell_i \leq d\epsilon \cdot \bar{\ell}_i + C_2\nu^2\rho^d\epsilon \cdot R. \end{cases}$$

Preuve du lemme 4

Soient $i \leq d$ et $x_i, \ldots, x_d \in \mathbb{Z}$. On se place à l'itération qui correspond à un état $(i, [x_i, \ldots, x_d])$, et on suppose qu'il existe $\nu \geq 1$ tel que pour tout $j > i$, $y_j < \nu\alpha^{j-1}$. La preuve se déroule en trois étapes : on majore successivement $\sum_j |x_j|$, Δc_i et finalement Δy_i.

1. *Borne sur* $\sum |x_k|$. Pour tout $j \geq i$, on pose $S_j = \sum_{k=j+1}^{d} |x_k|$ (en particulier $S_d = 0$). Soit $j > i$. Par hypothèse, on a $y_j \leq \nu\alpha^{j-1}$. En remplaçant y_j, cela donne $|x_j + \sum_{k>j} x_k\mu_{k,j}| \leq \nu\alpha^{j-1}$. Grâce à l'inégalité triangulaire et l'hypothèse de LLL-réduction, on a :

$$|x_j| \leq \nu\alpha^{j-1} + \sum_{k>j} |x_k||\mu_{k,j}| \leq \nu\alpha^{j-1} + \eta S_j.$$

On peut maintenant borner S_j pour tout $j \geq i$.

$$\begin{aligned} S_j &= |x_{j+1}| + S_{j+1} \leq \nu\alpha^j + (1+\eta)S_{j+1} \\ \frac{S_j}{\nu\alpha^j} &\leq 1 + (1+\eta)\alpha + \ldots + [(1+\eta)\alpha]^{d-j-1} \leq \frac{[(1+\eta)\alpha]^{d-j}}{\rho - 1} \\ S_j &\leq \frac{\nu\alpha^d(1+\eta)^{d-j}}{\rho-1}. \end{aligned}$$

2. *Borne sur* Δc_i. On définit récursivement $u_d = 0$ et $u_j = u_{j+1} \oplus (x_{j+1} \otimes \bar{\mu}_{j+1,i})$ pour $j \in \{i, \ldots, d-1\}$. Soit $j \geq i$. En utilisant l'inégalité $\eta + \kappa\epsilon \leq 1$, on obtient :

$$\begin{aligned}
|u_j| &\leq K(|x_{j+1} \otimes \bar{\mu}_{j+1,i}| + |u_{j+1}|) \\
&\leq K(K(\eta + \kappa\epsilon)|x_{j+1}| + |u_{j+1}|) \\
&\leq K^2|x_{j+1}| + K|u_{j+1}| \\
&\leq K^2|x_{j+1}| + K^3|x_{j+2}| + \ldots + K^{d-j+1}|x_d| \\
&\leq K^d S_j \\
|x_j \otimes \bar{\mu}_{j,i} - x_j\mu_{j,i}| &\leq |x_j \otimes \bar{\mu}_{j,i} - x_j\bar{\mu}_{j,i}| + |x_j(\bar{\mu}_{j,i} - \mu_{j,i})| \\
&\leq |x_j\bar{\mu}_{j,i}|\epsilon/2 + |x_j|\kappa\epsilon \\
&\leq |x_j|(1+\kappa)\epsilon
\end{aligned}$$

En utilisant le fait que $K^d \leq 2$, on obtient :

$$\begin{aligned}
\left|u_j - \sum_{k=j+1}^{d} x_k\mu_{k,i}\right| &\leq \left|u_{j+1} - \sum_{k=j+2}^{d} x_k\mu_{k,i}\right| + |x_{j+1} \otimes \bar{\mu}_{j+1,i} - x_j\mu_{j+1,i}| + |u_j|\epsilon/2 \\
&\leq \sum_{k=j+1}^{d} (|x_k \otimes \bar{\mu}_{k,i} - x_k\mu_{k,i}| + u_{k-1}\epsilon/2) \\
&\leq \sum_{k=j+1}^{d} \left(|x_k|(\kappa+1) + \frac{K^d}{2}S_{k-1}\right)\epsilon \\
&\leq \sum_{k=j+1}^{d} (|x_k|(\kappa+1) + S_{k-1})\,\epsilon \\
&\leq \sum_{k=j+1}^{d} (|x_k|(\kappa+2) + S_k)\,\epsilon \\
&\leq \sum_{k=j+1}^{d} (y_k(\kappa+2) + (\eta(2+\kappa)+1)S_k)\,\epsilon \\
&\leq \nu \sum_{k=j+1}^{d} \left((\kappa+2)\alpha^{k-1} + \frac{\eta(\kappa+4)\alpha^d(1+\eta)^{d-k}}{\rho-1}\right)\epsilon \\
&\leq C_1\nu\alpha^d(1+\eta)^{d-j}\epsilon
\end{aligned}$$

en posant $C_1 = \frac{\kappa+2}{\alpha-1} + \frac{\kappa+4}{\rho-1}$. Comme $c_i = -u_i$, on a $\Delta c_i \leq C_1\nu\alpha^d(1+\eta)^{d-i}\epsilon$.

3. *Borne sur* Δy_i. On a $y_i = |x_i - c_i|$ et $\bar{y}_i = |x_i \ominus \bar{c}_i|$. Par conséquent :

$$\Delta y_i \leq |x_i - \bar{c}_i|\frac{\epsilon}{2} + \Delta c_i \leq |x_i - c_i|\frac{\epsilon}{2} + K\Delta c_i,$$

ce qui conclue la preuve.

□

Preuve du lemme 5 En appliquant l'inégalité trianguaire, on obtient :

$$\begin{aligned}
|\bar{y}_i \otimes \bar{y}_i - y_i^2| &\leq |\bar{y}_i \otimes \bar{y}_i - \bar{y}_i^2| + |\bar{y}_i^2 - y_i^2| \\
&\leq \bar{y}_i^2\epsilon/2 + |\bar{y}_i^2 - y_i^2| \\
&\leq y_i^2\epsilon/2 + K|\bar{y}_i^2 - y_i^2| \\
&\leq y_i^2\epsilon/2 + K(2y_i + \Delta y_i)\Delta y_i \\
|(\bar{y}_i \otimes \bar{y}_i) \times \bar{r}_i - r_i y_i^2| &= |(y_i^2 + \bar{y}_i \otimes \bar{y}_i - y_i^2)(r_i + \bar{r}_i - r_i) - r_i y_i^2| \\
&\leq Ry_i^2\kappa\epsilon + R|\bar{y}_i \otimes \bar{y}_i - y_i^2| \\
&\leq R[y_i^2(\kappa + 1/2)\epsilon + K(2y_i + \Delta y_i)\Delta y_i] \\
|(\bar{y}_i \otimes \bar{y}_i) \otimes \bar{r}_i - r_i y_i^2| &\leq |(\bar{y}_i \otimes \bar{y}_i) \otimes \bar{r}_i - (\bar{y}_i \otimes \bar{y}_i) \times \bar{r}_i| + |(\bar{y}_i \otimes \bar{y}_i) \times \bar{r}_i - r_i y_i^2| \\
&\leq r_i y_i^2\epsilon/2 + K|(\bar{y}_i \otimes \bar{y}_i) \times \bar{r}_i - r_i y_i^2| \\
&\leq RK^2[(\kappa + 1)y_i^2\epsilon + (2y_i + \Delta y_i)\Delta y_i]
\end{aligned}$$

□

Preuve du lemme 6 Soient $i \leq d$ et $x_i, \ldots, x_d \in \mathbb{Z}$. On se place à l'itération qui correspond à un état $(i, [x_i, \ldots, x_d])$, et on suppose qu'il existe $\nu \geq 1$ tel que pour tout $j \geq i$, $y_j < \nu\alpha^{j-1}$. Soit $j \geq i$. En appliquant le lemme 4, on obtient :

$$\Delta y_j \leq y_j\epsilon/2 + KC_1\nu\alpha^d(1+\eta)^{d-j}\epsilon \leq \nu\epsilon(\alpha^{j-1}/2 + KC_1\alpha^d(1+\eta)^{d-j}\epsilon) \leq D_3\nu\alpha^d(1+\eta)^{d-j},$$

en posant $D_3 = 1/2 + KC_1$. En utilisant l'hypothèse sur la précision, on obtient :

$$y_j + \Delta y_j \leq K\nu(\alpha^{j-1} + 0,01) \leq 1.01K\nu\alpha^{j-1}.$$

On combine ces deux majorations avec le résultat du lemme 5.

$$\begin{aligned}
|(\bar{y}_i \otimes \bar{y}_i) \otimes \bar{r}_i - r_i y_i^2| &\leq RK^2[(\kappa + 1)\nu^2\alpha^{2(j-1)}\epsilon + (\nu\alpha^{j-1} + 1.01K\nu\alpha^{j-1})D_3\nu\alpha^d(1+\eta)^{d-j}\epsilon] \\
&\leq D_4\nu^2\alpha^{d+j}(1+\eta)^{d-j}R\epsilon,
\end{aligned}$$

en posant $D_4 = K^3[(\kappa + 1) + 2.01D_3]$. Il y a maintenant deux variantes de la majoration de $\Delta\ell_i$ à démontrer.

$$\begin{aligned}
\Delta\ell_i &\leq \ell_i\frac{\epsilon}{2} + K(\Delta\ell_{i+1} + |(\bar{y}_i \otimes \bar{y}_i) \otimes \bar{r}_i - y_i^2 r_i|) \\
&\leq K^d \sum_{j=i}^{d} \left(\ell_j\frac{\epsilon}{2} + |(\bar{y}_j \otimes \bar{y}_j) \otimes \bar{r}_j - y_j^2 r_j|\right) \\
&\leq K^d \sum_{j=i}^{d} \left(\ell_j\frac{\epsilon}{2} + D_4\nu^2\alpha^{d+j}(1+\eta)^{d-j}R\epsilon\right) \\
&\leq d\ell_i\epsilon + D_5\nu^2\alpha^d(1+\eta)^d R\epsilon,
\end{aligned}$$

en posant $D_5 = \dfrac{K^d D_4 \alpha}{1+\eta-\alpha}$. À noter que la denière ligne repose sur le fait que $1+\eta \geq \alpha$. Pour la seconde majoration, on procède de façon semblable :

$$\begin{aligned}
\Delta\ell_i &\leq \bar{\ell}_i \frac{\epsilon}{2} + (\Delta\ell_{i+1} + |(\bar{y}_i \otimes \bar{y}_i) \otimes \bar{r}_i - y_i^2 r_i|) \\
&\leq \sum_{j=i}^{d} \left(\bar{\ell}_j \frac{\epsilon}{2} + D_4 \nu^2 \alpha^{d+j} (1+\eta)^{d-j} R\epsilon \right) \\
&\leq d\bar{\ell}_i \epsilon + D_5 \nu^2 \alpha^d (1+\eta)^d R\epsilon.
\end{aligned}$$

Au final, on a $D_5 = \frac{K^{d+3}\alpha}{1+\eta-\alpha}[\kappa + 1 + 2.01(1/2 + KC_1)] \leq \frac{2\alpha}{1+\eta-\alpha}(2 + \kappa + 2C_1) = C_2$. □

7.2 Lemmes 1 et 2

Lemme 1. Soit $\mathbf{x} \in \mathbb{Z}^d$. On suppose que $n(\mathbf{x}) \leq r_1$. On se place dans l'état $(1, [x_1, \ldots, x_d])$. Comme la base est LLL-réduite, on a :

$$y_i \leq \sqrt{\frac{n(\mathbf{x})}{r_i}} \leq \frac{r_1}{r_i} \leq \alpha^{i-1}.$$

En appliquant le lemme 6 avec $\nu = 1$, on obtient directement le résultat. □

Lemme 2. Soit $\mathbf{x} \in \mathbb{Z}^d$ et $i \leq d$. On pose $\nu = \max\left(1, \sqrt{\frac{\bar{\ell}_i(K+d\epsilon)}{r_1(1-\epsilon')}}\right)$. On montre par récurrence sur j, pour j décroissant de d à i, que la borne sur Δc_j du lemme 4 s'applique et que $y_j \leq \nu\alpha^{j-1}$.

Soit $j \in [\![i, d]\!]$. D'après l'hypothèse de récurrence, pour tout $k > j$, on a $y_k \leq \nu\alpha^{k-1}$ donc le lemme 4 s'applique. L'étape principale consiste à borner y_j.

En utilisant les lemmes 5 et 6, on obtient :

$$\begin{aligned}
r_j y_j^2 &\leq \ell_j \leq \bar{\ell}_j + \Delta\ell_j \leq K\bar{\ell}_j + \Delta\ell_{j+1} + \left|(\bar{y}_j \otimes \bar{y}_j) \otimes \bar{r}_j - y_j^2 r_j\right| \\
&\leq K\bar{\ell}_j + \left(d\epsilon \cdot \bar{\ell}_j + C_2 \nu^2 \rho^d \epsilon \cdot R \right) + RK^2 \left[(\kappa+1) y_j^2 \epsilon + (2y_j + \Delta y_j)\Delta y_j \right].
\end{aligned}$$

On applique maintenant le lemme 4 pour borner Δy_j dans l'équation ci-dessus. On obtient alors une équation de la forme $P(y_j) \leq 0$, où P est un polynôme de degré 2 dont les coefficients sont :

$$\begin{aligned}
P_2 &= r_j - RK^3(\kappa+1)\epsilon \\
P_1 &= -RK^3 C_1 \nu \alpha^{d-j} (1+\eta)^d \epsilon(2+\epsilon) \\
P_0 &= -\bar{\ell}_i(K + d\epsilon) - C_2 \nu^2 R \rho^d \epsilon - RK^4 (C_1 \nu \rho^d \epsilon)^2.
\end{aligned}$$

Le fait que $\epsilon' \leq 0,01$ implique que $P_2 > 0$, donc y_j est inférieur à la racine positive de P. De plus, ϵ' a été défini de telle sorte que $\nu\alpha^{j-1}$ soit supérieur à la racine positive de P. On a donc $y_j \leq \nu\alpha^{j-1}$. Ceci termine la récurrence.

Pour conclure, on utilise la seconde partie du lemme 6, que l'on peut appliquer grâce aux majorations de y_j. □

Annexe B - Notations par ordre alphabétique

Notation	Section	Signification
A	3.1	Borne de la norme des vecteurs énumérés dans KFP.
α	2.3.1	Constante dépendant des paramètres de LLL ($\simeq 1,16$).
b		Base d'un réseau, toujours LLL-réduite à partir de 3.1.
b^*	2.2	Orthogonalisation de Gram-Schmidt de b.
c_i	3.1	Écart entre x_i et y_i.
C_1, C_2, C_3	4.2	Constantes de l'ordre de 1.
d		Dimension du réseau étudié ($d \leq n$).
δ	2.3.1	Paramètre de LLL (usuellement 3/4 ou $0,99$).
ϵ	2.1	ULP des flottants (2^{-52} pour la double précision).
ϵ'	4.2	Constante de l'ordre de $\rho^d\epsilon$.
η	2.3.1	Paramètre de LLL (usuellement 1/2).
FL_p	2.1	Ensemble des nombres flottants de précision p.
K	4.2	$1 + \epsilon/2 \simeq 1$
κ	4.1	Borne l'erreur sur $\bar{\mu}_{i,j}$ et $\bar{r}_j$.
L		Réseau engendré par la base b.
ℓ_i	3.1	Carré de la norme du vecteur de coordonnées $(0, \ldots, 0, y_i, \ldots, y_d)$ dans la base de Gram-Schmidt.
$\mu_{i,j}$	2.2	Coefficient utilisé pour définir la base de Gram-Schmidt.
n		Dimension de l'espace auquel appartiennent les vecteurs de b.
n (fonction)	4.3	Carré de la norme d'un vecteur donné par ses coordonnées dans L.
p	2.1	Précision des flottants (53 pour la double précision).
R	4.2	Majorant des $r_j + \Delta r_j$.
r_j	3.1	$\|\mathbf{b}_\mathbf{j}^*\|^2$
ρ	4.2	Constante dépendant des paramètres de LLL ($\simeq 3$).
sol	3.2	Solution renvoyée, exprimée par ses coordonnées dans b.
x_i		Coordonnée (entière) d'un vecteur du réseau $L(b)$ dans la base b.
y_i	3.1	Coordonnée d'un vecteur de $L(b)$ dans la base b^*.