Rapport de stage :
Réduction de réseaux euclidiens : vers un code fiable, souple et performant

Cadé David

Table des matières

Introduction

L'algorithme LLL de Lenstra, Lenstra et Lovász est utilisé pour résoudre de nombreux problèmes, comme la factorisation de polynômes à coefficients entiers, qui est l'application décrite dans l'article historique [7], ou en algèbre, pour obtenir une approximation rationnelle d'un nombre, et pour essayer de trouver des petites relations entières entre des nombres [3]. Il est aussi utilisé en cryptographie : la cryptanalyse de certains cryptosystèmes, comme certaines variantes de RSA [9]. Il permet de trouver des vecteurs courts dans un réseau, ce qui fournit une première approximation au problème de trouver un vecteur non-nul le plus court dans un réseau. Mais cela n'est pas suffisant pour obtenir un tel vecteur : en particulier certains cryptosystèmes reposent sur la difficulté de cette tâche [5].

Compte tenu de ses nombreuses applications, il est important de rendre la LLL-réduction la plus rapide possible. Beaucoup de chercheurs ont travaillé sur LLL pour tenter d'améliorer la complexité asymptotique de l'algorithme. L'algorithme L^2 de Nguyen et Stehlé [11] utilise des nombres flottants pour arriver à ce but, et d'après leurs auteurs, il s'agit de la meilleure variante prouvée de LLL à ce jour, parce que la complexité de l'algorithme est globalement un polynôme d'ordre 7. Il fait aussi bien que son prédécesseur : l'algorithme de Schnorr [13], qui, pour certaines variantes, est meilleur que L^2 en complexité vis-à-vis de d, mais sont moins utilisables en pratique. La complexité de L^2 est quadratique en $\log B$, alors que toutes les précédentes variantes prouvées sont cubiques en $\log B$ [17].

Il existe beaucoup d'implémentations de LLL, dont :

- `fplll-1.3`[15], implémentation de L^2 par Damien Stehlé. Elle implémente aussi des variantes heuristiques. Ceci est sous licence GPL v2.
- NTL[14], de Victor Shoup, bibliothèque permettant d'effectuer beaucoup d'opérations algébriques, dont LLL. Ceci est sous licence LGPL.
- MAGMA[8], logiciel permettant de faire du calcul formel. Ceci est sous licence propriétaire.

Je me suis intéressé dans ce stage à comment était implémenté `fplll-1.3`, afin de pouvoir faire un «wrapper» : c'est-à-dire essayer de deviner les meilleurs paramètres et la meilleure variante à utiliser afin de réduire une base donnée le plus vite possible.

Pour cela j'ai tout d'abord modifié le code et suis passé au C++ de manière à pouvoir utiliser toutes les variantes de LLL dans le même exécutable. J'ai ensuite ajouté le «wrapper», qui se sert de quelques heuristiques afin de trouver quelles variantes appeler avec quels paramètres. Ensuite j'ai ajouté quelques optimisations, comme utiliser automatiquement le fait que la matrice d'entrée soit triangulaire, ou bien la proprification anticipée.

Nous allons dans un premier temps voir quelques rappels mathématiques requis pour comprendre le rapport. Puis dans la deuxième section, nous allons décrire les algorithmes LLL et L^2. Ensuite, dans la troisième section, nous aborderons l'implémentation des variantes de L^2 et des optimisations faites. Et enfin, dans la quatrième section, nous allons voir comment se compare notre implé-

mentation de LLL avec d'autres implémentations disponibles sur le marché.

1 Notations, prérequis

Nous allons aborder dans cette partie les notations et les différentes notions requises à la compréhension du rapport.

1.1 Notations

On note les vecteurs en gras : le vecteur $\mathbf{x}$ et le scalaire y.

Soit $\langle \cdot, \cdot \rangle$ le produit scalaire canonique. On note $\| \cdot \|$ la norme euclidienne : $\|\mathbf{x}\| = \sqrt{\langle \mathbf{x}, \mathbf{x} \rangle}$.

On notera $\lfloor x \rceil$ la valeur entière la plus proche de x.

Dans le cas où x est aussi proche de sa partie entière supérieure et inférieure, on prendra pour $\lfloor x \rceil$ la partie entière supérieure.

1.2 Orthogonalisation de Gram-Schmidt

L'*orthogonalisation de Gram-Schmidt* d'un ensemble de vecteurs libres $B = \{\mathbf{b}_1, \ldots, \mathbf{b}_d\}$ est l'ensemble des vecteurs $(\mathbf{b}_i^*)_{i \leq d}$, définis récursivement par :

$$\mathbf{b}_i^* = \mathbf{b}_i - \sum_{j=1}^{i-1} \mu_{i,j} \mathbf{b}_j^*$$

avec :

$$\mu_{i,j} = \frac{\langle \mathbf{b}_i, \mathbf{b}_j^* \rangle}{\|\mathbf{b}_j^*\|^2}$$

On prendra dans la suite $\mu_{i,i} = 1$ pour $1 \leq i \leq d$.

Les vecteurs $(\mathbf{b}_i^*)_{i \leq d}$ sont orthogonaux entre eux et l'espace engendré par les vecteurs $\mathbf{b}_1, \ldots, \mathbf{b}_i$ est le même que celui engendré par les vecteurs $\mathbf{b}_1^*, \ldots, \mathbf{b}_i^*$.

On abrègera parfois dans la suite *orthogonalisation de Gram-Schmidt* par GSO.

1.3 Réseau

Un *réseau* R est un sous-groupe discret d'un $\mathbb{R}^n$. Une *base* B de ce réseau est un ensemble de vecteurs *libres* $\mathbf{b}_1, \ldots, \mathbf{b}_d$ tels que R soit l'ensemble des combinaisons linéaires *entières* d'éléments de B :

$$R = \left\{ \mathbf{x} \in \mathbb{R}^n, \exists (a_i)_{i \leq d} \in \mathbb{Z}^d \text{ et } \mathbf{x} = \sum_{i=1}^{d} a_i \cdot \mathbf{b}_i \right\}$$

Pour un réseau donné, le nombre d'éléments dans une base de ce réseau est constant. On l'appelle la *dimension* du réseau. On a toujours $d \leq n$. Il existe une infinité de bases pour un réseau donné dès que $d \geq 2$. Deux bases d'un

réseau B_1 et B_2 sont liées entre elles par une matrice U de déterminant 1 et inversible dans $\mathbb{Z}$. On a $B_1 = U \cdot B_2$. Sur la figure 1, le réseau représenté par les points admet pour bases les deux couples de vecteurs représentés l'un avec des flèches fines et l'autre avec des flèches plus grasses. Ce ne sont évidemment pas les deux seuls réseaux de la base, et on peut déjà voir sur cette figure une notion de qualité d'une base : la base en gras a des vecteurs «assez» courts et ils sont «assez» orthogonaux entre eux ; alors que l'autre base a des vecteurs longs et peu orthogonaux.

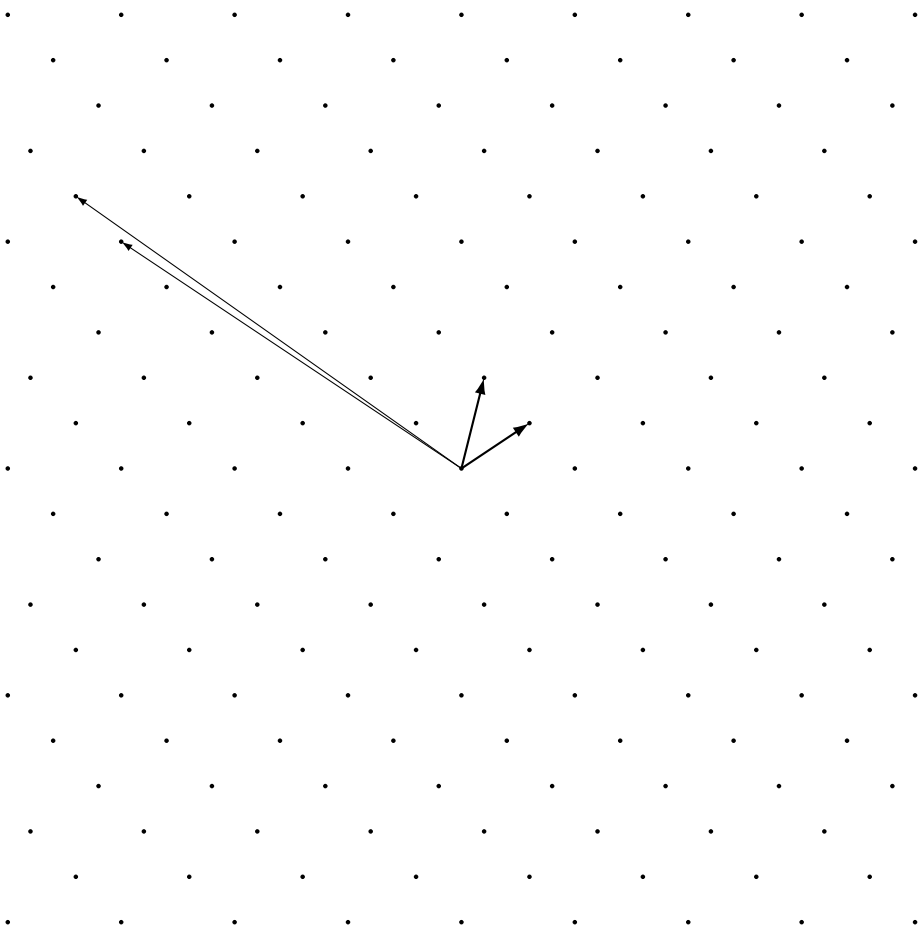

FIG. 1 – Un réseau et deux bases de celui-ci

La matrice de Gram d'une base $B = (\mathbf{b}_1, \ldots, \mathbf{b}_d)$ est la matrice des produits

scalaires de ses vecteurs :

$$G = \begin{pmatrix} \langle \mathbf{b}_1, \mathbf{b}_1 \rangle & \langle \mathbf{b}_1, \mathbf{b}_2 \rangle & \cdots & \langle \mathbf{b}_1, \mathbf{b}_d \rangle \\ \langle \mathbf{b}_2, \mathbf{b}_1 \rangle & \langle \mathbf{b}_2, \mathbf{b}_2 \rangle & \cdots & \langle \mathbf{b}_2, \mathbf{b}_d \rangle \\ \vdots & \ddots & \ddots & \vdots \\ \langle \mathbf{b}_d, \mathbf{b}_1 \rangle & \langle \mathbf{b}_d, \mathbf{b}_2 \rangle & \cdots & \langle \mathbf{b}_d, \mathbf{b}_d \rangle \end{pmatrix}$$

On appelle volume d'un réseau la quantité $V = \prod_{i=1}^{d} \|\mathbf{b}_i^*\|$, avec $\{\mathbf{b}_1, \ldots, \mathbf{b}_d\}$ une base du réseau. Cette quantité ne dépend pas de la base considérée.

On peut aussi remarquer que $\det G = V^2$.

1.4 Arithmétique flottante

Définition Un *nombre flottant* est un triplet (m, e, s), où m est la *mantisse*, nombre entier non signé sur p bits, dont le bit de poids fort est à 1. L'entier e est l'*exposant*, nombre entier signé sur p_{exp} bits, et s est un bit de signe. Le nombre représenté par ce triplet est $(-1)^s \cdot (m2^{-p+1}) \cdot 2^{p_{exp}}$. La quantité $M = (m2^{-p+1})$ est codée par la mantisse, et appartenant à $[1, 2[$. On appelle *précision* l'entier p.

Pour $x \in \mathbb{R}$, soit $\diamond(x)$ le nombre flottant le plus proche. Lorsqu'il y a ambiguïté, on prend le nombre flottant le plus proche et dont la mantisse est paire. Le comportement des `double` par défaut lors d'un appel à une opération arithmétique comme l'addition, la soustraction, la multiplication et la division, ou lors d'un appel à une fonction est d'arrondir le résultat au plus proche. Faire $r := a/b$ avec des nombres flottants revient à faire $r := \diamond(a/b)$.

Implémentations Un ***double*** est un nombre flottant avec $p = 53$, $p_{exp} = 11$. (En fait, pas tout à fait, car il faut enlever les exposants -1023 et 1024 utilisés pour représenter NaN et les infinis.) Ce type est décrit dans le standard IEEE-754 [1]. Il faut 64 bits pour représenter un `double` : 52 bits pour la mantisse (on n'écrit pas le bit de poids fort, toujours égal à 1), 11 bits d'exposant, et un bit de signe.

Prenons par exemple le nombre -2.5. On a $s = 1$. Il faut choisir l'exposant pour que l'on ait $M \in [1, 2[$. On le prend donc égal à $p_{exp} = 2$. On a alors $M = 1.25$, et donc $m = 1.25 \cdot 2^{52} =_2 1010^{50}$.

L'intérêt du type `double` est qu'il est implémenté dans la majeure partie des processeurs actuels, et les opérations faisant appel aux `double` sont donc optimisées. Par contre la taille de l'exposant est très limitée : un `double` ne peut pas dépasser les $(2 - 2^{52}) \cdot 2^{1023}$; ce qui est génant.

On peut passer outre ce problème avec les ***dpe*** de Patrick Pelissier et Paul Zimmermann, qui est une structure contenant un `double` d et un entier e sur 32 bits, servant d'exposant. Le nombre représenté par un `dpe` est $d \cdot 2^e$; et d est gardé dans $[1/2, 1[$. Ce choix semble bizarre, parce qu'un `double` avec un exposant nul et de signe positif est entre $[1, 2[$. Cette bibliothèque est sous licence LGPL.

La bibliothèque `mpfr`[12], elle, permet d'utiliser des nombres flottants avec une précision arbitraire, utilisant la bibliothèque `GNU MP`[2]. Cette bibliothèque

gère les nombres entiers de taille arbitraire et essaye d'optimiser au maximum les opérations pour l'architecture utilisée.

Ces deux bibliothèques sont aussi sous licence LGPL.

Problèmes usuels Deux phénomènes courants se produisent lorsque l'on travaille avec des nombres flottants.

Nous illustrons le phénomène classique d'*annulation* à l'aide d'un exemple. Supposons que nous utilisons des `double`. Soit

$$M = (2^{100} + 2^{40}) - (2^{100} - 2^{40}) = 2^{41}$$

2^{100} et 2^{40} sont représentables avec des `double`, c'est-à-dire $\diamond(2^{100}) = 2^{100}$ et $\diamond(2^{40}) = 2^{40}$. On a une précision de 53 bits, donc $\diamond(2^{100} + 2^{40}) = \diamond(2^{100} - 2^{40}) = 2^{100}$. Calculer M avec des `double` revient à calculer

$$\diamond\left(\diamond(2^{100} + 2^{40}) - \diamond(2^{100} - 2^{40})\right) = 0$$

, qui est bien loin des 2^{41} que l'on voudrait trouver. Additionner des flottants est donc dangereux, et cela nous concerne particulièrement parce que nous utilisons des produits scalaires.

Un autre phénomène qu'on observe est la perte de précision. Soit r_a le résultat attendu, et r_o le résultat obtenu en utilisant des nombres flottants. L'erreur relative sur r_a est la quantité $\frac{|r_a - r_o|}{|r_a|}$. [4] Faire une opération avec des nombres flottants ne permet pas d'assurer que la précision que l'on a sur les nombres de départ soit gardée. Et si on fait des opérations sur des nombres flottants ayant déjà perdu de la précision, on en perd encore plus. L'orthogonalisation de Gram-Schmidt faite avec des nombres flottants est très sujette à ce problème.

2 Présentation de l'algorithme LLL

Dans cette section, nous allons présenter comment fonctionne l'algorithme LLL.

2.1 Notations, définitions

On se donne une base $(\mathbf{b}_1, \ldots, \mathbf{b}_d)$ d'un réseau en dimension n.

La base $(\mathbf{b}_1, \ldots, \mathbf{b}_d)$ est dite *η-propre* si un facteur $\eta \geq 1/2$ si pour tous i, j tels que $1 \leq j < i \leq d$, on a $|\mu_{i,j}| \leq \eta$.

Un vecteur $\mathbf{b}_i$ d'une base est dit *η-propre* pour un facteur $\eta \geq 1/2$ si pour tout j tel que $1 \leq j < i$, on a $|\mu_{i,j}| \leq \eta$. Une base est *(δ, η)-LLL-réduite* pour les facteurs $\delta \in]1/4; 1]$ et $\eta \in [1/2, \sqrt{\delta}[$ si elle est η-propre et si elle vérifie la condition de Lovász pour tout i entre 2 et d :

$$\delta \cdot \|\mathbf{b}_{i-1}^*\|^2 \leq \|\mathbf{b}_i^* + \mu_{i,i-1}\mathbf{b}_{i-1}^*\|^2$$

Ces définitions sont illustrées sur la figure 2. Le disque rouge correspond à la condition de Lovász : si $\mathbf{b}_2$ se trouve à l'extérieur de ce disque, alors il vérifie la

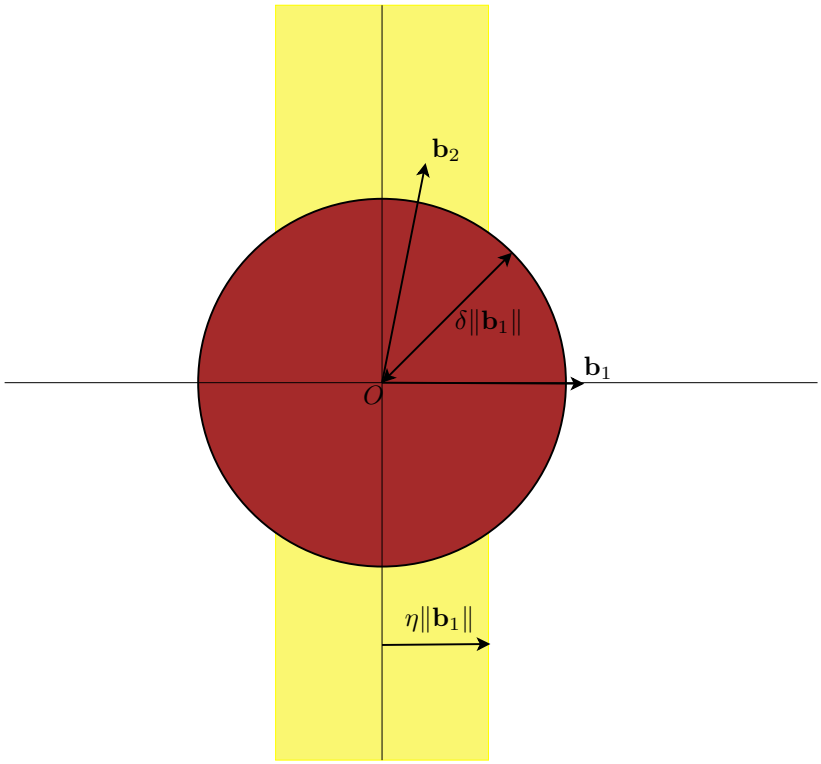

FIG. 2 – Illustration des conditions de la LLL-réduction

condition de Lovász. Le vecteur $\mathbf{b}_2$ doit être «assez grand» par rapport à $\mathbf{b}_1$. La bande jaune correspond à la condition de propreté, c'est-à-dire que la projection de $\mathbf{b}_2$ sur l'espace engendré par $\mathbf{b}_1$ doit être «assez petite» par rapport à $\mathbf{b}_1$.

La partie jaune de la figure correspond donc à l'ensemble des $\mathbf{b}_2$ pour lesquels la base $B = (\mathbf{b}_1, \mathbf{b}_2)$ est LLL-réduite. On peut décrire cela comme l'ensemble des $\mathbf{b}_2$ «assez grands» et «assez orthogonaux» par rapport à $\mathbf{b}_1$.

La figure peut être généralisée avec b_1 devenant b_i^* et b_2 devenant b_{i+1}.

2.2 LLL

2.2.1 Description de l'algorithme

Le but de l'algorithme est, partant d'une base d'un réseau quelconque contenu dans $\mathbb{Z}^n$, d'arriver à une base (δ, η)-LLL-réduite pour des paramètres $\delta \in]1/4, 1[$ et $\eta = 1/2$ fixés, et tout ceci en temps polynomial. L'algorithme terminera aussi pour $\delta = 1$, mais on ne pourra pas assurer qu'il terminera en temps polynomial.

Données : une base $(\mathbf{b}_1, \ldots, \mathbf{b}_\kappa)$, et les $\mu_{i,j}$ pour $1 \leq j \leq i \leq \kappa$.
Résultat : la base avec le vecteur $\mathbf{b}_\kappa$ 1/2-propre, et la GSO mise à jour.

```
1 pour i allant de κ − 1 jusqu'à 1 faire
2     b_κ = b_κ − ⌊μ_{κ,i}⌉ b_i;
3     pour j allant de 1 à i faire
4         μ_{κ,j} = μ_{κ,j} − ⌊μ_{κ,i}⌉ μ_{i,j};
5     fin
6 fin
```

Algorithme 1 : Algorithme de proprification

L'algorithme 1 permet de rendre le vecteur $\mathbf{b}_\kappa$ propre. On peut montrer facilement qu'à la fin de l'algorithme, tous les $\mu_{\kappa,j}$ pour $j < \kappa$ sont entre $[-1/2, 1/2]$.

L'algorithme 2 prend le vecteur courant d'indice κ, le proprifie, puis teste la condition de Lovász. Si cette condition est vérifiée, on passe au vecteur suivant, sinon on échange le vecteur courant avec le précédent, puis on recommence. À chaque début de tour de la boucle principale, on a que la base $(\mathbf{b}_1, \ldots, \mathbf{b}_{\kappa-1})$ est LLL-réduite.

2.2.2 Terminaison de LLL

Nous montrons ici que l'algorithme termine. Il est clair que si l'algorithme termine, il renvoie une base LLL-réduite, du fait de l'invariant de boucle.

On introduit la valeur $D = \prod_{k=1}^{d} v_k$, avec $v_k = \prod_{i=1}^{k-1} \|\mathbf{b}_i^*\|^2$. Cette quantité diminue lorsque un échange est effectué. Il y a au plus un échange par tour de boucle et si il y en a un, c'est un échange entre $\mathbf{b}_{\kappa-1}$ et $\mathbf{b}_\kappa$. Supposons qu'on échange les vecteurs $\kappa - 1$ et κ. Les quantités $v_1, ..., v_{\kappa-1}$ ne changent pas. Le volume d'un réseau $\prod_{k=1}^{d} \|\mathbf{b}_k^*\|$ est constant, or les seuls $\mathbf{b}_i^*$ ayant changé sont $\mathbf{b}_{\kappa-1}^*$ et $\mathbf{b}_\kappa^*$, donc le produit $\|\mathbf{b}_{\kappa-1}^*\| \cdot \|\mathbf{b}_\kappa^*\|$ est constant. Donc les valeurs $v_{\kappa+1}, ..., v_d$ ne changent pas non plus. La seule valeur qui change dans v_κ est

Données : une base $(\mathbf{b}_1, \ldots, \mathbf{b}_\kappa)$, et $1/4 \leq \delta < 1$.
Résultat : la base $(\delta, 1/2)$-LLL-réduite.

$\kappa = 2$;
tant que $\kappa \leq d$ **faire**
    En se servant de la GSO calculée jusqu'à $\kappa - 1$, calculer les $\mu_{\kappa,i}$ pour $i < \kappa$ et en déduire la GSO jusqu'à κ;
    Proprifier κ en utilisant l'algorithme 1;
    si $\delta\|\mathbf{b}^*_{\kappa-1}\|^2 \leq \|\mathbf{b}^*_{\kappa-1} - \mu_{\kappa,\kappa-1}\mathbf{b}^*_\kappa\|^2$ **alors**
        $\kappa = \kappa + 1$;
    sinon
        Échanger $\mathbf{b}_\kappa$ avec $\mathbf{b}_{\kappa-1}$ dans la base;
        $\kappa = \max(\kappa - 1, 2)$;
    fin
fin

Algorithme 2 : LLL

$\|\mathbf{b}^*_{\kappa-1}\|^2$. Avant l'échange, la condition de Lovász n'est pas vérifiée, donc on a $\delta\|\mathbf{b}^*_{\kappa-1}\|^2 > \|\mathbf{b}^*_{\kappa-1} - \mu_{\kappa,\kappa-1}\mathbf{b}^*_\kappa\|^2$. Cette condition peut se réécrire :

$$(\delta - \mu^2_{\kappa,\kappa-1})\|\mathbf{b}^*_{\kappa-1}\|^2 > \|\mathbf{b}^*_\kappa\|^2$$

et donc $\delta\|\mathbf{b}^*_{\kappa-1}\|^2 > \|\mathbf{b}^*_\kappa\|^2$. D baisse donc d'un facteur au moins $1/\delta$. Les v_i sont les volumes au carré des sous-réseaux engendrés par les bases $(\mathbf{b}_1, \ldots, \mathbf{b}_i)$. Or le volume au carré est le déterminant de la matrice de Gram, les vecteurs $\mathbf{b}_1, \ldots, \mathbf{b}_d$ sont à coordonnées entières, ainsi les v_i sont entiers. Par conséquent D est entier. Au début, on a $D \leq B^{d(d+1)}$, donc on a au maximum $d(d+1)\log_{1/\delta}(B)$ échanges, et si on n'échange pas, κ augmente de 1, et on s'arrête si κ dépasse d. On ne peut augmenter κ au maximum que $d + d(d+1)\log_{1/\delta}(B)$ fois. Le nombre maximal de tours dans la boucle principale est borné par :

$$d + 2d(d+1)\log_{1/\delta}(B)$$

L'algorithme termine, et on a une borne en $O(d^2 \log B)$ sur le nombre total de tours.

Le fait que l'algorithme termine en temps polynomial est plus difficile à montrer, la partie difficile étant d'arriver à montrer que la taille des $\mathbf{b}_i$ et des $\mu_{i,j}$ ne peut pas «trop» augmenter lors de la phase de proprification. On peut trouver une preuve dans [16]. La complexité de l'algorithme est de $O(d^5 n \log^3 B)$. Kaltofen [6] obtient une meilleure borne avec le même algorithme, mais la preuve est encore plus complexe.

2.3 L^2

Cet algorithme utilise des nombres à virgule flottante pour améliorer la complexité de l'algorithme LLL, et prend en entrée aussi uniquement des bases de réseaux inclus dans $\mathbb{Z}^n$.

2.3.1 Description de l'algorithme

Il calcule la matrice de Gram exacte pour éviter les pertes de précision dans le calcul de l'orthogonalisation de Gram-Schmidt et les produits scalaires.

On notera dans la suite $r_{i,i} = \|\mathbf{b}_i^*\|^2$ et $r_{i,j} = \mu_{i,j} r_{j,j} = \langle \mathbf{b}_i, \mathbf{b}_j^* \rangle$.

L'architecture de l'algorithme 3 de L^2 ressemble beaucoup à celle de LLL. On se sert toujours d'un κ, index du vecteur courant. L'algorithme commence aussi par proprifier le vecteur $\mathbf{b}_\kappa$, lignes 8 à 22. Puis il teste la condition de Lovász, lignes 23 à 29, et augmente ou diminue κ selon les cas.

Les mises à jour de G reviennent à modifier G de manière à tenir compte des modifications sur B.

Une amélioration facile qui n'est pas présentée dans le code ci-dessus, et faite dans `fplll-1.3`, est de regrouper plusieurs échecs successifs du test de la condition de Lovász. Si le test échoue, le vecteur $\mathbf{b}_\kappa$ qui était déjà propre par rapport aux vecteurs $(\mathbf{b}_1, \ldots, \mathbf{b}_{\kappa-1})$ l'est aussi par rapport aux vecteurs $(\mathbf{b}_1, \ldots, \mathbf{b}_{\kappa-2})$. Donc la proprification suivante ne modifiera rien. On peut directement réessayer le test de Lovász suivant.

Les variables δ^+ et η^- de l'algorithme sont utilisées parce que les résultats des calculs avec flottants sont intrinsèquement imprécis. Un nombre flottant x peut représenter n'importe quel nombre réel entre $x - 2^{-p}$ et $x + 2^{-p}$. C'est pourquoi, pour pouvoir prouver la (δ, η)-LLL-réduction, on prend une marge suffisante sur les paramètres δ et η.

Pour plus de détails voir la preuve de la correction de L^2 dans [10].

2.3.2 Variantes heuristiques

Dans les variantes heuristiques, ni le fait que l'algorithme finisse, ni le fait que la base éventuellement renvoyée soit bien LLL-réduite n'est garanti.

Précision Pour gagner en efficacité, il est naturel de réduire la base avec moins de précision qu'il ne le faudrait dans le pire cas. Ceci peut amener à des problèmes d'annulation dans la boucle de proprification, donc il se peut que la boucle de proprification ne finisse pas. Par contre cela peut permettre de finir bien plus vite.

Pour cela, on peut utiliser, par exemple, les `double` ou les `dpe`, qui permettent de faire des calculs beaucoup plus vite qu'avec une bibliothèque de nombres flottants multiprécision comme `mpfr`.

Pas de matrice de Gram La variante sans matrice de Gram (appelée «`heuristic`» plus tard) ne tient pas compte de la matrice de Gram, donc on ne se met plus à l'abri d'éventuelles annulations lors des calculs de produits scalaires, mais cela peut accélérer le calcul dans certains cas.

Les deux en même temps On peut, évidemment, à la fois utiliser une variante sans matrice de Gram et avec une précision non optimale.

Données : une base $B = (\mathbf{b}_1, ..., \mathbf{b}_d)$, un couple (δ, η)
Résultat : une base (δ, η)-LLL-réduite

Calculer la matrice de Gram G;
$\eta^- = \frac{\eta+1/2}{2}$;
$\delta^+ = \frac{\delta+1}{2}$;
$\kappa = 2$;
$\bar{r}_{1,1} = \diamond(\langle \mathbf{b}_1, \mathbf{b}_1 \rangle)$;
tant que $\kappa \leq d$ **faire**
    $b = vrai$;
    tant que b **faire**
        Calculer les $\bar{r}_{\kappa,j}$ et les $\bar{\mu}_{\kappa,j}$ pour $1 \leq j \leq \kappa$ à partir de G et des $\bar{\mu}_{i,j}$ et $\bar{r}_{i,j}$ précédents;
        $b = faux$;
        pour $i = \kappa - 1$ *jusqu'à 1* **faire**
            si $|\bar{\mu}_{\kappa,i}| \geq \eta^-$ **alors**
                $b = vrai$;
                $X = \lfloor \bar{\mu}_{\kappa,i} \rceil$;
                $\mathbf{b}_k = \mathbf{b}_k - X \cdot \mathbf{b}_i$;
                Mettre à jour G;
                pour $j = 1$ *jusqu'à* $i - 1$ **faire**
                    $\bar{\mu}_{\kappa,j} = \diamond(\bar{\mu}_{\kappa,j} - \diamond(X \cdot \bar{\mu}_{i,j}))$;
                fin
            fin
        fin
    fin
    si $\delta^+ \cdot \bar{r}_{\kappa-1,\kappa-1} < \bar{r}_{\kappa,\kappa} + \bar{\mu}_{\kappa,\kappa-1}\bar{r}_{\kappa-1,\kappa-1}$ **alors**
        $\kappa = \kappa + 1$;
    sinon
        Échanger $\mathbf{b}_{\kappa-1}$ et $\mathbf{b}_\kappa$;
        Mettre à jour G, $\bar{\mu}$ et $\bar{r}$;
        $\kappa = \max(\kappa - 1, 2)$;
    fin
fin

Algorithme 3 : Algorithme L^2

Variante fast On peut constater que la plupart du temps, pendant la LLL-réduction, toutes les coordonnées d'un vecteur de la base sont d'un même ordre de grandeur. On peut alors factoriser les exposants, c'est-à-dire représenter les vecteurs de B par les couples $(e_i, \mathbf{b}_i^f)$ avec e_i des entiers, et $\mathbf{b}_i^f$ des vecteurs de `double` tels que $\mathbf{b}_i = 2^{e_i}\mathbf{b}_i^f$, et $e_i = \lfloor 1 + \log_2 \max_{j \leq d} |\mathbf{b}_i[j]| \rfloor$. Cela revient à partager un exposant commun à tous les coefficients du vecteur.

La variante `fast` est basée sur cette représentation et est calculée sans matrice de Gram.

Cette méthode est plus rapide que la méthode avec `dpe` et sans matrice de Gram, parce que les produits scalaires sont effectués beaucoup plus vite.

Prenons $\mathbf{b}_1$ et $\mathbf{b}_2$ et calculons le produit scalaire avec des `dpe` et `fast`.

Pour `dpe`, soit $\mathbf{b}_1 = ((f_{1,1}, e_{1,1}), \ldots, (f_{1,n}, e_{1,n}))$ et $\mathbf{b}_2 = ((f_{2,1}, e_{2,1}), \ldots, (f_{2,n}, e_{2,n}))$.

Un produit de deux `dpe` (f_1, e_1) et (f_2, e_2) est calculé par $dpenorm(f_1 f_2, e_1 + e_2)$, avec *dpenorm* qui fait en sorte que le premier argument soit entre $[1/2, 1[$ en divisant par la puissance de 2 qu'il faut, et l'ajoute dans le deuxième argument.

Une somme de deux `dpe` (f_1, e_1) et (f_2, e_2) est calculé en prenant celui qui a l'exposant maximal, et en ajoutant à son flottant le flottant de l'autre divisé par $2^{e_1 - e_2}$. Il faut ensuite vérifier que le flottant obtenu ne dépasse pas 1, et si c'est le cas, diviser par 2 le flottant et ajouter 1 à l'exposant. Le produit scalaire est obtenu en utilisant ces deux opérations sur les membres des vecteurs.

Pour `fast`, soit $\mathbf{b}_1 = (e_1, (f_{1,1}, \ldots, f_{1,n}))$ et $\mathbf{b}_2 = (e_2, (f_{2,1}, \ldots, f_{2,n}))$. Le produit scalaire est tout simplement $2^{e_1+e_2}\langle f_1, f_2 \rangle$.

Donc les produits scalaires avec `dpe` coûtent bien plus cher. Les opérations faisant intervenir $\bar{\mu}$ et $\bar{r}$ sont aussi plus rapides avec `fast`.

3 Implémentation

J'ai basé mon code sur celui de `fplll-1.3`. Ce programme est écrit en C, il permet l'utilisation de plusieurs types d'entiers :

- `int`, le type entier de base du C ;
- `mpz`, les nombres entiers de taille arbitraire de la bibliothèque `GMP` ;

de plusieurs types flottants :

- `double`, le type de flottants du C respectant le standard IEEE 754 pour 64 bits ;
- `dpe`, qui est une structure contenant un `double` et un `int` qui sert d'exposant au `double` ;
- `mpfr`, le type de flottants à précision arbitraire de la bibliothèque MPFR ;

et de «méthodes» de calcul de LLL :

- `proved`, qui est une implémentation de L^2 ;
- `heuristic`, qui est une implémentation de la variante heuristique discutée précédemment ;
- `fast`, qui est une implémentation de la variante «fast».

Toutes ces possibilités sont mises en place avec une utilisation intensive dans le code de blocs `#ifdef`...`#endif`, et pour pouvoir utiliser LLL avec des types

et méthodes particulières, il faut recompiler avec les bons switchs `-D` du compilateur. Par exemple, pour utiliser `heuristic` avec `int` et `mpfr`, il faut compiler avec :

```
make heuristic "CC=gcc -DUSE_MPFR -DZUSE_INT"
```

Ce que je devais faire était un programme qui choisisse la meilleure méthode suivant la base donnée en entrée, donc je ne pouvais pas me permettre de recompiler à chaque fois qu'il fallait changer de méthode. C'est pourquoi il fallait que je puisse compiler toutes ces possibilités dans le même exécutable.

3.1 Gestion de l'arithmétique

La solution qui m'a paru la plus naturelle était de faire appel aux templates du C++, pour deux raisons.

Tout d'abord, le langage C++ est quasiment un sur-ensemble du langage C, à part quelques modifications, le code de `fplll-1.3` compilait avec un compilateur C++. Ceci m'a permis de ne pas avoir à tout réécrire.

Ensuite, les templates du C++ permettent justement d'écrire du code générique. Voici un exemple simple de ce concept :

```
template<class T> T sqr(T x)
{
   return x*x;
}
```

Cette fonction met au carré l'argument `x` de type générique T et le renvoie.

```
int x=3;
int y=sqr(x);
double z=4.;
double t=sqr(z);
```

À la fin `y` vaudra 9 et `t` vaudra 16.

Les fonctions génériques peuvent être *spécialisées*, c'est-à-dire que son code peut être donné pour un type particulier. Lorsque le compilateur remarque qu'il doit appeller une fonction générique pour un type donné U, si il n'existe pas de spécialisation pour ce type, il créé la fonction en remplacant tous les T par des U.

J'ai architecturé le code autour de `Z_NR`, classe d'entiers générique, et `FP_NR`, classe de nombres flottants générique. Cela revient à mettre en argument un type à la classe.

Les types `Z_NR` et `FP_NR` contiennent leurs opérations. Voici un bout de code pour expliquer comment cela s'utilise.

```
Z_NR<mpz_t> a;//Créer la variable a
a.set(10);    //L'initialiser à 10
Z_NR<mpz_t> b;//Créer la variable b
```

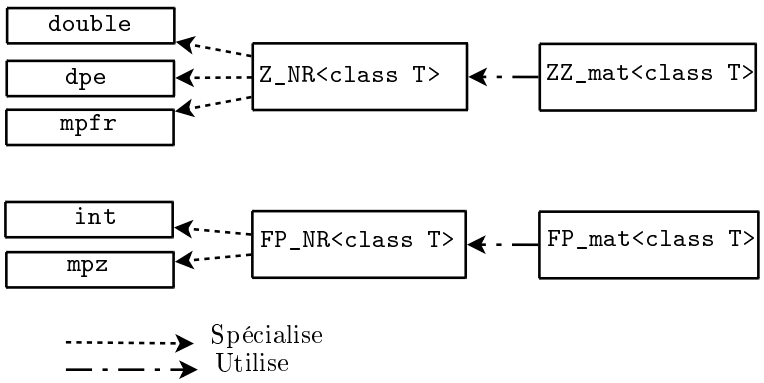

FIG. 3 – Les types entiers et flottants

```
b.set(15);    //L'initialiser à 15

Z_NR<mpz_t> c;//Créer la variable c
c.mul(a,b);   //Lui affecter la valeur a+b
```

On peut, bien entendu, changer tous les `mpz_t` en `long int` dans le code qui précède, et dans ce cas-là il fera les mêmes opérations, mais en utilisant le type entier `long int`.

Par ailleurs, certaines opérations font intervenir un type flottant générique et un type entier générique en même temps. Par exemple la conversion d'un entier en flottant. Ceci est implémenté avec une fonction template `set_fz`.

Chaque variante de LLL est une classe template prenant en argument les types qui doivent être utilisés.

3.2 Gestion automatique des différentes variantes

Maintenant qu'il est possible d'utiliser toutes les variantes dans le même exécutable, on va rendre le choix des paramètres et des variantes transparent à l'utilisateur.

3.2.1 Savoir si une variante a échoué

Rappelons que toutes les variantes à part `proved` avec `mpz` et `mpfr` avec une précision suffisante ne permettent pas de garantir qu'elles vont s'arrêter, ni qu'elles vont renvoyer un résultat LLL-réduit. On doit donc trouver des moyens de savoir si la méthode qu'on a lancée a échoué.

Pour cela, il faut garantir que l'algorithme ne parte pas en boucle infinie. Il y a deux boucles `tant que` dans l'algorithme, les seules boucles ayant des chances de ne pas finir.

La boucle `tant que` globale a un nombre d'itérations maximal, il suffit de vérifier qu'on ne dépasse pas ce nombre. Nous avons déterminé un nombre d'itérations maximal dans la section 2.2.2.

Lors de la proprification, on remarque que si, pendant une itération, les $|\mu_{i,j}|$ ne baissent pas assez, (sauf pour la dernière itération, bien entendu), alors il y a des risques de boucle infinie. On peut appuyer ce raisonnement sur la variante prouvée de L^2, où $M = \max |\mu_{i,j}|$ baisse au moins d'une valeur déterminée. Le théorème 5 dans [11] montre que M baisse pendant une itération de l'algorithme de proprification. Avec $\delta = 0.998$ et $\eta = 0.501$, et une constante $c > 12$, la quantité M_1 représentant M après le tour de boucle vérifie l'inégalité :

$$M_1 \leq 0.5007 + 2^{1-c} M$$

Intuitivement, la diminution de M d'un certain rapport entraîne la diminution de la taille du vecteur proprifié d'environ le même rapport : les $\mathbf{b}_1, \ldots, \mathbf{b}_{\kappa-1}$ sont quasi orthogonaux entre eux, et on fait baisser la taille des projections sur ces vecteurs d'environ ce rapport.

Pour vérifier que les $|\mu_{i,j}|$ diminuent bien sauf pour la dernière itération, je garde en mémoire les 3 valeurs précédentes de M, et je teste si la valeur de M a assez diminué entre l'avant-dernière et l'antépénultième. Pour l'instant «baisser assez» est défini par la contrainte qu'il y ait au moins un rapport 2^{10} entre les M consécutifs considérés. Ce rapport semble suffisant, parce qu'il évite déjà les boucles infinies de proprification en réduisant assez vite la taille des vecteurs.

3.2.2 Savoir pourquoi une variante a échoué

Il y a essentiellement deux causes possibles à l'échec d'une variante :
- soit des annulations dans les produits scalaires,
- soit un manque de précision pour le calcul de la GSO.

En moyenne, le premier problème peut apparaître à n'importe-quelle dimension, alors que le deuxième apparaît à des dimensions plus élevées [17].

Pour savoir pourquoi la variante a échouée, on peut regarder le κ final et voir si il est grand ou pas. Si il est grand, il y a plus de chances que le problème vienne d'un manque de précision ; si il est petit, il y a plus de chances que le problème vienne d'annulations.

J'ai effectué des tests pour trouver la dimension où apparaît en moyenne le deuxième problème, pour un certain nombre de couples (δ, η), avec la précision fixée, et j'utilise ces résultats pour savoir si κ est «grand» ou «petit».

Les matrices que j'ai utilisées sont dix matrices de type «sac-à-dos» de la forme :

$$\begin{pmatrix} random(2^e) & 1 & 0 & 0 & \cdots & 0 \\ random(2^e) & 0 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ random(2^e) & 0 & 0 & \cdots & \cdots & 1 \end{pmatrix}$$

avec $9995 \leq e \leq 10004$ et de dimension 300×301.

La fonction $random(x)$ renvoie un nombre pseudo-aléatoire uniformément distribué entre 0 et $x-1$. J'ai utilisé les fonctions de la bibliothèque GMP pour la génération de nombres pseudo-aléatoires. On considère le réseau engendré par les lignes de la matrice.

Les phénomènes d'annulation sont plutôt rares avec les matrices de type «sac-à-dos».

J'ai regardé pour quel κ la variante `fast` échoue sur tous les couples (δ, η) valides tels que δ soit multiple de 0.01 et η soit multiple de 0.05 et pour $\eta = 0.501$. Au vu des résultats, on a l'impression que cette dimension critique pourrait s'écrire comme le produit de deux fonctions, l'une dépendant uniquement du paramètre η, et l'autre uniquement du paramètre δ : $d(\eta, \delta) = f(\eta) \cdot g(\delta)$. Les fonctions f et g que j'ai obtenues expérimentalement sont représentés sur les figures 4 et 5. Par ailleurs, cette dimension critique dépend linéairement de la précision utilisée.

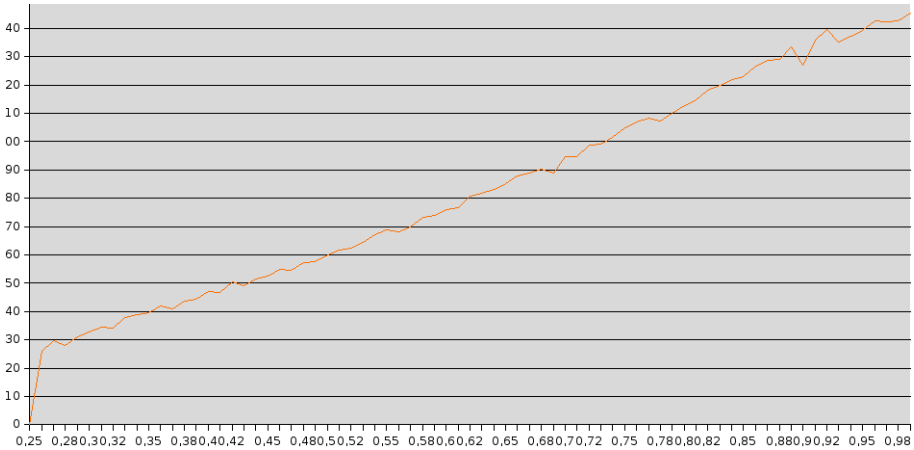

FIG. 4 – La fonction g expérimentale obtenue

3.2.3 Architecture générale du wrapper

Tout d'abord, il faut regarder si la taille des éléments de la base donnée en entrée sont «grands» ou pas. En effet, si leur taille dépasse les 512 bits, il y a beaucoup de chances que les produits scalaires calculés dépassent l'exposant maximum d'un `double` qui est de 1024.

Si l'exposant du double ne suffit pas, l'algorithme produira des infinis et des NaN (Not a Number) dans la boucle de proprification, ce qui sera détecté.

Une fois la taille des entrées matricielles considérée, on lance d'abord la méthode la plus rapide possible selon le cas : `heuristic` avec `double` si la base est petite, sinon `fast`.

Ensuite, si il y a échec et si κ est petit, on suppose que le problème vienne

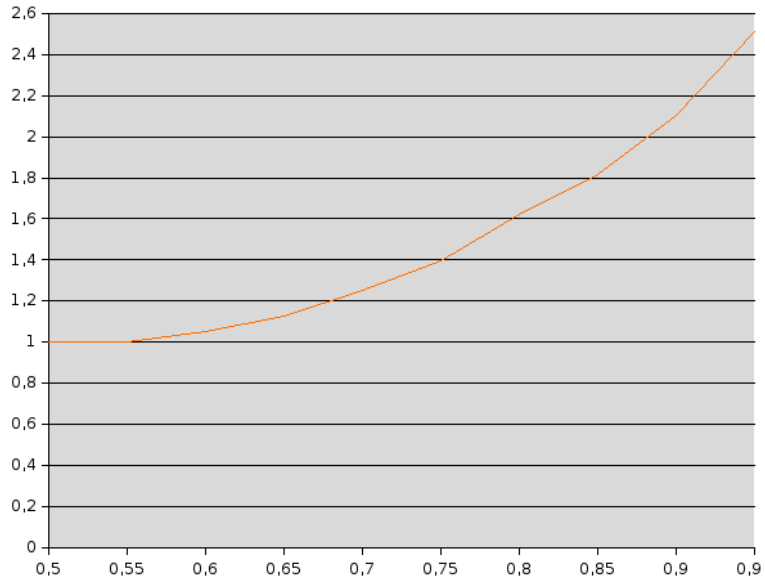

FIG. 5 – La fonction f expérimentale obtenue

d'annulations, et donc que la variante utilisée n'est pas d'assez bonne qualité. De `fast`, on passe à `heuristic`, et de `heuristic`, on passe à `proved`.

Si il y a échec et κ est grand, alors vraisemblablement la précision n'est pas suffisante. Il faut alors augmenter la précision du calcul. La précision maxprec correspond à la précision qu'il faut pour garantir un résultat LLL-réduit. On a à peu près maxprec $= 1.6 \cdot d$ pour η proche de 0.5 et δ proche de 1, donc pour des dimensions utilisées de l'ordre de 200, cette précision devient très grande. Faire une réduction utilisant des nombres flottants avec une grande précision prend beaucoup plus de temps qu'en faire une avec une précision plus petite. C'est pourquoi je double la précision dans ce cas-là. Cette augmentation paresseuse permet d'être plus rapide dans la plupart des cas.

Sur la figure 3.2.3, les flèches représentent quelle méthode choisir après un échec pour continuer. On y voit bien le fait que lorsque κ est petit, on change de méthode pour une méthode plus rigoureuse. Et lorsque κ est grand, on augmente la précision.

Une fois qu'une variante a réussi, on regarde si le résultat est bien LLL-réduit en utilisant la variante proved avec `mpfr` et avec la précision maxprec. Si la base est réduite, cela se fait très vite par rapport à la réduction.

3.3 Matrices triangulaires

Il arrive souvent que les matrices considérées dans LLL soient quasi triangulaires, par exemple les matrices de type «sac-à-dos» que l'on a vues plus haut.

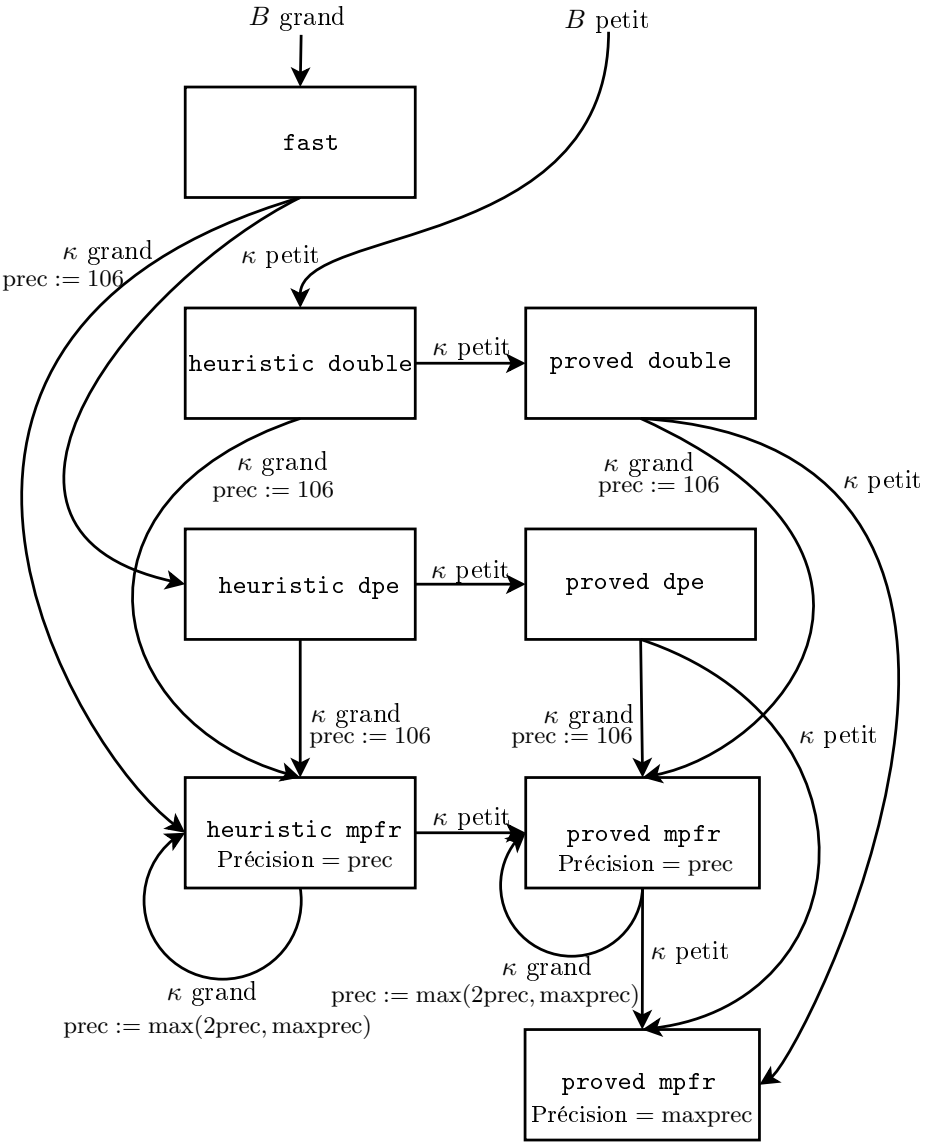

FIG. 6 – Architecture du wrapper

Dans l'algorithme, on garde en mémoire la valeur κ_m, qui indique le κ maximum déjà rencontré.

On définit le *décalage* s d'une matrice $B = (b_{i,j})$ comme le nombre $\max_i\{\min\{j-i, j \geq i \text{ et } b_{i,j} = 0\}\}$. Cela représente le nombre de colonnes qu'il faudrait enlever au début de la matrice pour qu'on aie une matrice triangulaire.

On garde en mémoire dans l'algorithme κ_m, le rang κ maximum déjà atteint. À une étape κ donnée de l'algorithme, la matrice a la forme :

$$B = \begin{pmatrix} b_{1,1} & \cdots & b_{1,\kappa_m+s} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{\kappa_m,1} & \cdots & b_{\kappa_m,\kappa_m+s} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \vdots \\ b_{n-s-1,1} & \cdots & \cdots & \cdots & \ddots & 0 \\ b_{n-s,1} & \cdots & \cdots & \cdots & \cdots & b_{n-s,n} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{d,1} & \cdots & \cdots & \cdots & \cdots & b_{d,n} \end{pmatrix}$$

Tous les vecteurs $(\mathbf{b}_1, \ldots, \mathbf{b}_\kappa)$ ont des zéros sur les coordonnées $i > m$ avec $m = \min(\kappa_m + s, n)$. Donc pendant le calcul de la proprification, on peut se limiter aux m premières coordonnées, les suivantes ne seront pas affectées.

Pour des matrices triangulaires ou presque, on peut remarquer un gain de temps important en faible dimension. Avec une matrice de type «sac-à-dos» de dimension 10 avec $e = 1000$, on passe de 60ms à 40ms de temps de calcul environ. On observe un gain bien moindre dans les dimensions plus élevées. Pour des matrices non triangulaires, on ne remarque pas de différences de temps de calcul.

3.4 Proprification anticipée

Une proprification anticipée (early reduction) revient à proprifier tous les vecteurs qui suivent κ et non pas seulement κ dans l'algorithme. En faisant cela, on baisse la taille des vecteurs, ce qui rendra les proprifications suivantes plus rapides.

On en fait une lorsqu'on proprifie pour la première fois les vecteurs d'indice $(2^i)_{i\geq 1}$. Faire cette proprification anticipée trop souvent n'est pas utile et ne fera que ralentir la réduction. Mais en faire une de temps en temps permet dans certains cas de rendre la réduction beaucoup plus rapide.

4 Comparaison avec les autres implémentations de LLL

`fplll-2.0` est légèrement plus lent que `fplll-1.3` à cause de l'abstraction en C++, mais est plus facile d'utilisation. Dans le graphique qui suit, j'ai testé

LLL de plusieurs bibliothèques avec plusieurs matrices de type «sac-à-dos» de la forme :

$$\begin{pmatrix} random(2^e) & 1 & 0 & 0 & \cdots & 0 \\ random(2^e) & 0 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ random(2^e) & 0 & 0 & \cdots & \cdots & 1 \end{pmatrix}$$

J'ai pris $e = 1000$ et des matrices de dimension multiples de 10 entre 10 et 200.

Sur la figure 7, j'ai mis le temps pris par différentes bibliothèques à comparer avec les temps pris par les bibliothèques `fplll-2.0, wrapper` et `fplll-2.0, fast early`.

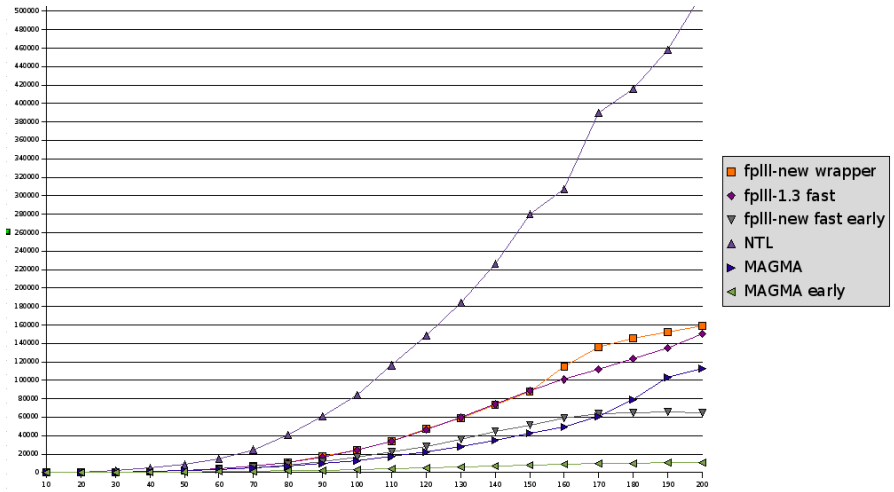

FIG. 7 – Résultats obtenus

On peut remarquer qu'à partir de la dimension 160, la méthode `fplll-2.0 wrapper` devient plus lente que `fplll-1.3`. Ceci est dû à l'échec de la méthode `fast`, première méthode considérée par le wrapper : lors de la boucle de proprification, les $|\mu_{i,j}|$ ne décroissaient pas assez vite, donc il y avait des risques de boucle infinie, et donc la méthode a été arrêtée pour continuer avec la méthode suivante dans le graphe des variantes de la figure 3.2.3, même si cela n'aurait pas été le cas vu que le `fast` de `fplll-1.3` a fini. On est peut-être un peu trop protectionniste

On peut remarquer aussi que sur les grandes dimensions, `MAGMA` est bien meilleur ; cela est dû à sa gestion des entiers qui sont les entiers natifs de l'architecture lorsque le nombre est assez petit, et des entiers GMP sinon. Or avec e petit par rapport à la dimension, les calculs seront souvent faits avec des entiers. La variante de `MAGMA` avec l'early-reduction est encore plus efficace parce qu'elle

permet de changer de méthode pour une réduction moins sûre mais plus efficace lors de l'exécution.

Lorsque e est grand par rapport à la dimension, `MAGMA` et `fplll-2.0` vont à peu près aussi vite l'un que l'autre. (voir les dimensions<50 sur le graphe).

L'avantage de l'implémentation `fplll-2.0` est tout de même sa licence libre GPLv2 qui permet de modifier son code et de rajouter ce qu'il manque.

Conclusion et perspectives

En quelques points, voici les avantages de `fplll-2.0` sur les autres implémentations.

- `fplll-2.0` est aussi competitif que `MAGMA` dans la plupart des cas, voire plus rapide dans certains cas. Surtout quand e est très grand, l'arithmétique sur les grands entiers de GMP domine le temps de calcul, et est utilisé de manière plus fine dans `fplll-2.0`.
- `fplll-2.0` est libre et donc peut être modifié et modulé. Ceci facilite le développement de nouvelles stratégies ou variantes de LLL.
- `fplll-2.0` est intégrable dans d'autres librairies. En particulier il est techniquement faisable de l'integrer dans Pari/GP et NTL.

On peut encore ajouter beaucoup de choses encore à l'implémentation actuelle, comme

- La possibilité de changer de variante vers une variante plus faible lorsque la base est assez réduite, comme dans `MAGMA`.
- Un type entier hybride combinant entier et GMP, comme dans `MAGMA`.
- Changer l'orthogonalisation de Gram-Schmidt par une version numériquement plus stable comme Householder [4].
- L'algorithme de certification de Gilles Villard [18] permet de vérifier qu'une base est bien LLL-réduite, de manière plus efficace qu'exécuter la variante prouvée. Un thésard travaille actuellement sur cet algorithme.

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. 6

[2] T. Granlund. The GNU MP Bignum Library. Available at `http://gmplib.org/`. 6

[3] G. Hanrot. Lll : a tool for effective diophantine approximation. In *Proceedings of the LLL+25 Conference*, pages 81–118, 2007. 3

[4] N. Higham. *Accuracy and Stability of Numerical Algorithms.* Society for Industrial and Applied Mathematics, 2002. 7, 22

[5] J. Hoffstein, N. Howgrave-Graham, J. Pipher, and W. Whyte. Practical lattice-based cryptography : NTRUEncrypt and NTRUsign. In *Proceedings of the LLL+25 Conference*, pages 199–236, 2007. 3

[6] E. Kaltofen. On the complexity of finding short vectors in integer lattices. In *Proceedings of EUROCAL'83*, volume 162 of *Lecture Notes in Computer Science*, pages 236–244. Springer-Verlag, 1983. 10

[7] A. K. Lenstra, H. W. Lenstra, Jr., and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261 :513–534, 1982. 3

[8] Magma. The Magma computational algebra system for algebra, number theory and geometry. Available at `http://magma.maths.usyd.edu.au/magma/`. 3

[9] A. May. Using LLL-reduction for solving RSA and factorization problems : A survey. In *Proceedings of the LLL+25 Conference*, pages 170–198, 2007. 3

[10] P. Nguyen and D. Stehlé. Low-dimensional lattice basis reduction revisited (extended abstract). In *Proceedings of the 6th Algorithmic Number Theory Symposium (ANTS VI)*, volume 3076 of *Lecture Notes in Computer Science*, pages 338–357. Springer-Verlag, 2004. 11

[11] P. Nguyen and D. Stehlé. Floating-point LLL revisited. In *Proceedings of Eurocrypt 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 215–233. Springer-Verlag, 2005. 3, 16

[12] The SPACES Project. MPFR, a LGPL-library for multiple-precision floating-point computations with exact rounding. Available at `http://www.mpfr.org/`. 6

[13] C. P. Schnorr. A hierarchy of polynomial lattice basis reduction algorithms. *Theoretical Computer Science*, 53 :201–224, 1987. 3

[14] V. Shoup. NTL, Number Theory C++ Library. Available at `http://www.shoup.net/ntl/`. 3

[15] D. Stehlé. fplll-1.3, a floating-point LLL implementation. Available at `http://perso.ens-lyon.fr/damien.stehle`. 3

[16] D. Stehlé. *Algorithmique de la réduction de réseaux et application à la recherche de pires cas pour l'arrondi de fonctions mathématiques.* PhD thesis, Université Henri Poincaré Nancy 1, 2005. 10

[17] D. Stehlé. Floating-point LLL : Theoretical and practical aspects. In *Proceedings of the LLL+25 Conference*, pages 34–63, 2007. 3, 16

[18] G. Villard. Certification of the QR factor R, and of lattice basis reducedness. In *Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation (ISSAC'07)*. ACM Press, 2007. 22