Chapitre 1

Réseaux euclidiens

Ce cours est largement inspiré de divers textes de Phong Q. NGuyen, notamment[12, 13].

On note les vecteurs en **gras** et on les écrit en ligne (souvent, en France, ils sont écrits en colonne ; ce document utilise plutôt la convention anglo-saxonne). Les coordonnées individuelles ne sont pas en gras (ce sont juste des nombres). Donc on a $\mathbf{x} = (x_1, \ldots, x_n)$. OK ?

1.1 Introduction

La *norme euclidienne* d'un vecteur (sa « longueur ») est notée :

$$\|\mathbf{x}\| = \sqrt{{x_1}^2 + {x_2}^2 + \cdots + {x_n}^2}$$

Considérons des vecteurs $\mathbf{b}_1, \ldots, \mathbf{b}_m$ de $\mathbb{R}^n$. À ce stade, tout le monde sait ce qu'est un espace vectoriel engendré par $(\mathbf{b}_1, \ldots, \mathbf{b}_m)$: c'est l'ensemble des combinaisons linéaires des $\mathbf{b}_i$, autrement dit c'est l'ensemble

$$\langle \mathbf{b}_1, \ldots, \mathbf{b}_n \rangle = \left\{ \sum_{i=1}^{m} \lambda_i \mathbf{b}_i : (\lambda_1, \ldots, \lambda_m) \in \mathbb{R}^m \right\}$$

On sait que si deux vecteurs $\mathbf{x}$ et $\mathbf{y}$ appartiennent à un espace vectoriel, alors leur somme $\mathbf{x} + \mathbf{y}$ y appartient aussi, de même que $\lambda\mathbf{x}$, pour tout $\lambda \in \mathbb{R}$. Lorsque les $\mathbf{b}_i$ sont linéairement indépendants, on dit qu'ils forment une *base* de l'espace vectoriel en question (et on a $m = n$).

Un *réseau euclidien* (« Euclidean lattice »), c'est presque la même chose, mais avec des entiers. Il suffit de remplacer $\mathbb{R}$ par $\mathbb{Z}$ dans le paragraphe précédent ! Mais bien sûr, ça change tout.

Définition 1 : Réseau euclidien

> Si $\mathbf{b}_1, \ldots, \mathbf{b}_d$ sont des vecteurs de $\mathbb{Z}^n$ alors ils **engendrent** le **réseau euclidien** $\mathcal{L}(\mathbf{b}_1, \ldots, \mathbf{b}_d)$ —qu'on va abréger $\mathcal{L}$— qui est formé de toutes les combinaisons linéaires à coefficients entiers des $\mathbf{b}_i$:
>
> $$\mathcal{L}(\mathbf{b}_1, \ldots, \mathbf{b}_d) = \left\{ \sum_{i=1}^{d} \mu_i \mathbf{b}_i : \mu_1, \ldots, \mu_d \in \mathbb{Z} \right\}.$$
>
> Si en plus les $\mathbf{b}_i$ sont linéairement indépendants (ce qui sera généralement le cas), on dit qu'ils forment une **base** de $\mathcal{L}$. Dans ce cas, d est la **dimension** du réseau.
>
> Dans le cas particulier important où $d = n$ (la dimension d du réseau est égale à la dimension n de l'espace ambiant $\mathbb{R}^n$), on dit que le réseau est **de rang plein**.

Par exemple, la figure 1.1 montre le réseau de dimension 2 engendré par $\mathbf{b_1} = (1, 5)$ et $\mathbf{b_2} = (2, 2)$. La différence avec les espaces vectoriels saute aux yeux : un réseau est un ensemble *discret* (par opposition à « continu », c.a.d. qu'il y a « du vide » entre les points du réseau). On voit que les points du réseau sont disposés régulièrement dans l'espace. La figure 1.2 montre un réseau en trois dimensions.

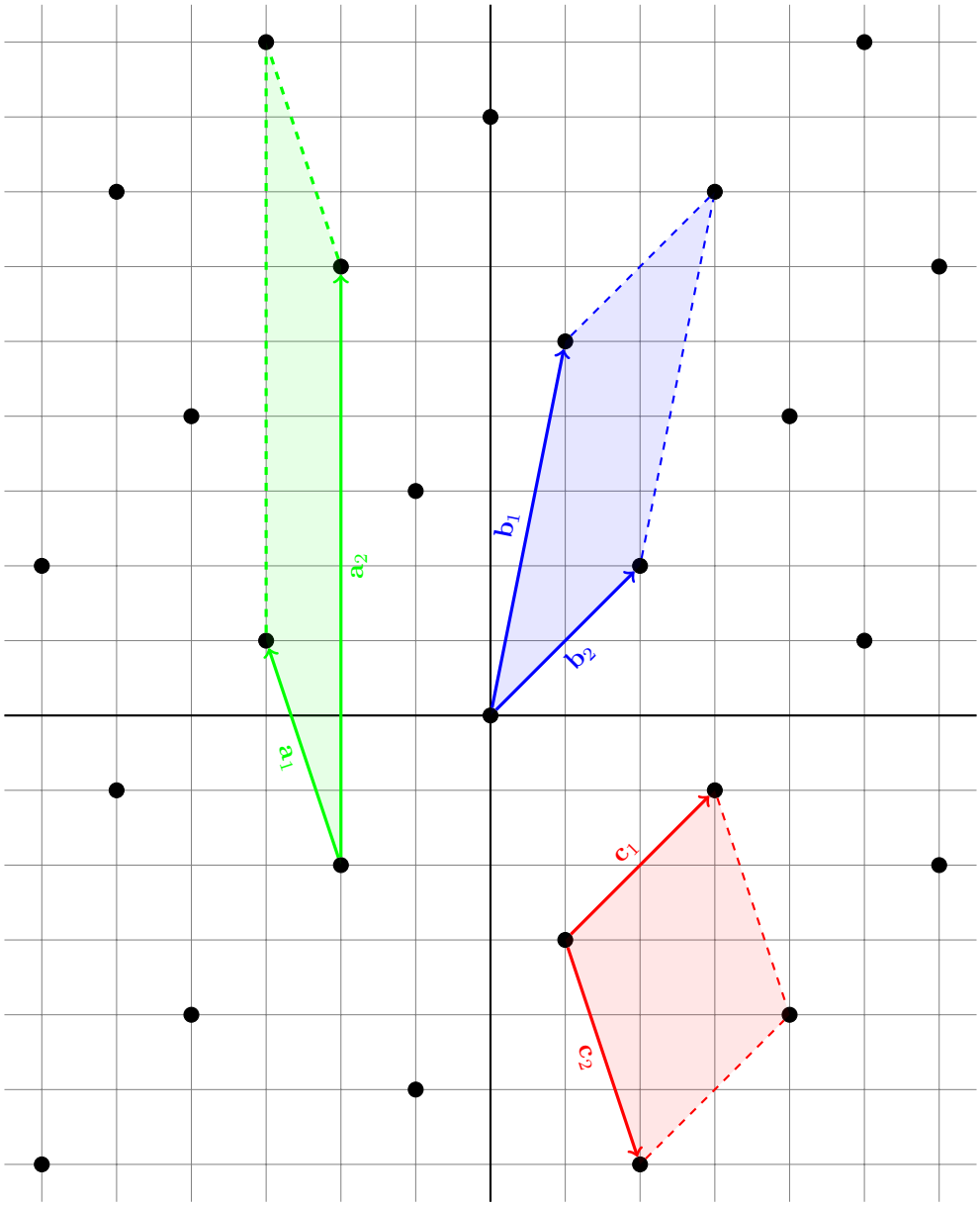

FIGURE 1.1 – Un réseau euclidien en deux dimensions (ce sont les points noirs), avec plusieurs bases possibles et les parallélepipèdes fondamentaux qui correspondent.

1.1.1 Bases d'un réseau

Comme une base est formée d'une collection de vecteurs, il est commode de les représenter par des matrices. Ainsi, la base $(\mathbf{b}_1, \mathbf{b}_2)$ ci-dessus est représentée par (les lignes de) la matrice :

$$B = \begin{pmatrix} 1 & 5 \\ 2 & 2 \end{pmatrix}$$

Tous les points du réseau s'écrivent donc $\mathbf{x}B$, où $\mathbf{x}$ est un vecteur à coordonnées entières.

Le même réseau peut être engendré par plusieurs bases. En effet, celui de la figure 1.1 aurait aussi été engendré par

$$A = \begin{pmatrix} -1 & 3 \\ 0 & 8 \end{pmatrix} \quad \text{et} \quad C = \begin{pmatrix} 2 & 2 \\ 1 & -3 \end{pmatrix}$$

On passe d'une matrice de base à une autre en multipliant (à gauche) par une matrice de « changement de base » à coefficients entiers et dont l'inverse est aussi à coefficients entiers (une telle matrice est dite *unimodulaire* ; il faut et il suffit que son determinant soit ± 1).

Étant donné une base B d'un réseau, il est facile de déterminer si un vecteur $\mathbf{y}$ appartient au réseau ou pas : il suffit de tester si le système linéaire $\mathbf{x}B = \mathbf{y}$ possède une solution $\mathbf{x}$ à coefficients entiers.

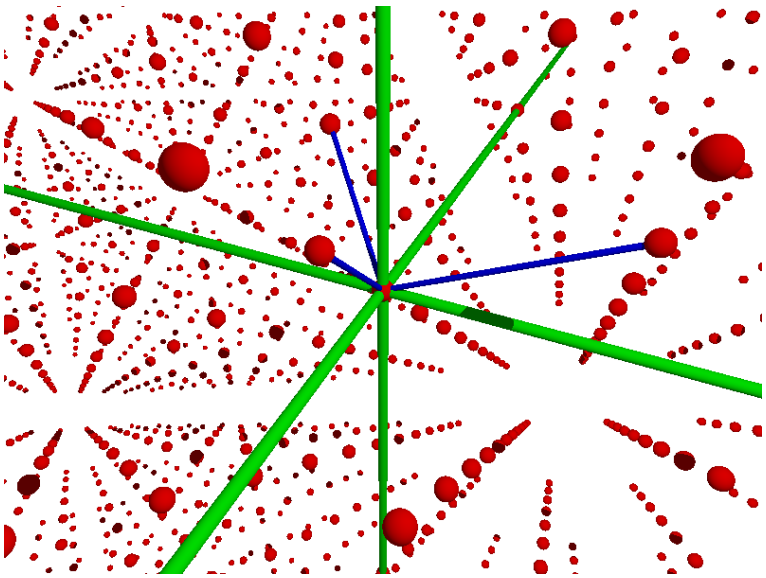

FIGURE 1.2 – Un réseau euclidien en trois dimensions. Les axes sont en vert et les vecteurs de base en bleu.

1.1.2 Volume

Si on fixe une base $(\mathbf{b_1}, \ldots, \mathbf{b}_d)$, on peut former le *parallélepipède fondamental* qui correspond (ils sont représentés sur la figure 1.1) c'est-à-dire l'ensemble

$$\left\{ \sum_{i=1}^{d} x_i \mathbf{b}_i : x_1, \ldots, x_d \in [0;1[\right\}.$$

Un tel parallélepipède réalise un pavage de $\mathbb{R}^n$. Des bases différentes en donnent de formes différentes, mais il se trouve qu'ils ont tous le même *volume*[1]. Ce n'est pas évident, mais on admet que c'est vrai.

En fait, c'est même un peu plus fort : une partie D de $\mathbb{R}^n$ (mesurable), telle que les ensembles $\mathbf{b} + D$ (pour $\mathbf{b} \in \mathcal{L}$) recouvrent $\mathbb{R}^n$ et sont d'intérieurs disjoints, s'appelle un domaine fondamental du réseau $\mathcal{L}$. Tous les domaines fondamentaux de $\mathcal{L}$ ont le même volume. Par exemple, les régions du diagramme de Voronoï associées aux points du réseau font aussi l'affaire. Cf. figures 1.3.

Par conséquent, le volume d'un parallélepipède fondamental ne dépend pas du choix de la base, mais c'est une caractéristique du réseau lui-même. On parle donc du *volume* du réseau $\mathcal{L}$ et on le note $\mathrm{Vol}(\mathcal{L})$. On a un moyen simple de calculer le voume :

Théorème 1. *Soit $\mathcal{L}$ un réseau de base B. On a $\mathrm{Vol}(\mathcal{L}) = \sqrt{\det BB^t}$. Dans le cas particulier (important) où $\mathcal{L}$ est de rang plein, alors $\mathrm{Vol}(\mathcal{L}) = |\det B|$.*

Le cas particulier pratique est une conséquence facile du cas général. En effet, si B est une matrice carrée, alors $\det BB^t = (\det B)^2$.

Le coefficient (i, j) de la « matrice de Gram » BB^t est le produit scalaire $\mathbf{b}_i \cdot \mathbf{b}_j$. En conséquence, si la base est orthogonale, tous ces produits scalaires sont nuls saufs ceux sur la diagonale, et le déterminant forme leur produit. Le volume est alors le produit des longueur des $\mathbf{b}_i$, donc c'est bien le volume du parallélepipède qu'ils forment. Et en fait, c'est bien le cas même s'ils ne sont pas orthogonaux.

Le volume du réseau de la figure 1.1 est donc 8. En général, on a souvent affaire à des matrices B qui sont triangulaires, donc le déterminant est très facile à calculer : c'est le produit des entrées sur la diagonales.

Cette notion de volume a plusieurs intérêts. D'abord, plus le volume est faible, plus les points du réseau sont serrés les uns aux autres : la « densité » du réseau est grosso-modo l'inverse du volume. Chaque point du réseau « occupe » un volume $\mathrm{Vol}(\mathcal{L})$.

1. En deux dimensions, le « volume » est simplement la surface (en mètres-carrés). En trois dimension, c'est le volume normal (en mètres-cubes). En $d > 3$ dimension... c'est « l'hypervolume » en m^d.

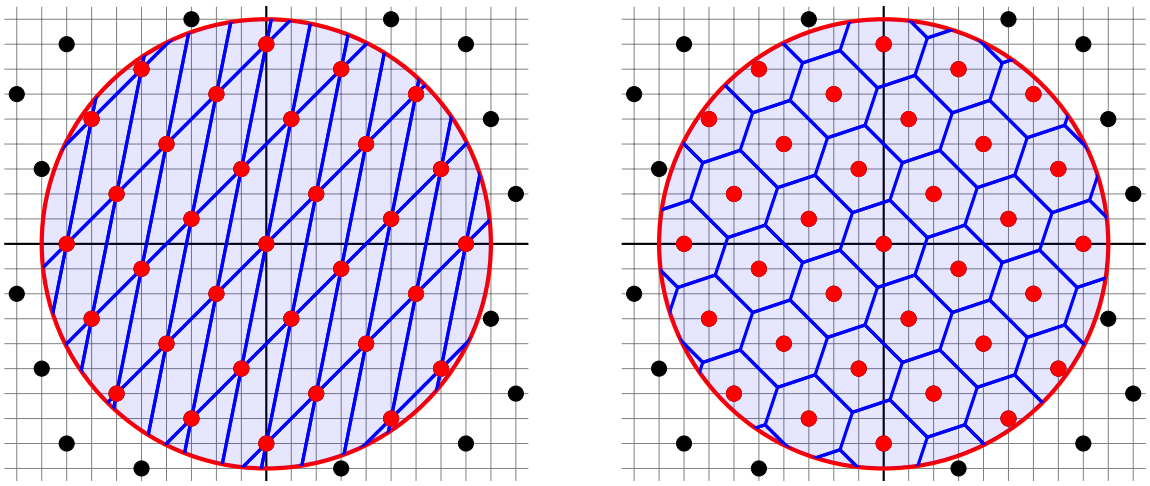

FIGURE 1.3 – Illustration de l'heuristique Gaussienne. À gauche : le parallélepipède fondamental associé à la base B. À droite : le diagramme de Voronoï du réseau. Dans les deux cas, le nombre de points du réseau dans le cercle est à peu près le nombre de copies de la forme répétée contenues dans le cercle, donc à peu près le volume du cercle divisé par le volume du réseau.

Ceci permet de faire le raisonnement suivant, qui se nomme *l'heuristique gaussienne*. Prenons une boule[2] de rayon r ; combien de point du réseau contient-elle ? On peut estimer ceci en disant :

$$\#\text{ points dans la boule} \approx \frac{\text{Volume de la boule}}{\text{Volume du réseau}}.$$

Ceci est une approximation, car à la frontière de la boule il y a des parallélépipèdes fondamentaux tronqués (cf. figure 1.3).

1.1.3 Défaut d'orthogonalité d'une base

Toutes les bases d'un réseau ne sont pas aussi intéressantes les unes que les autres. Certaines ont des vecteurs plus longs que d'autres ; dans certaines, les vecteurs sont plus orthogonaux que dans d'autres (rappel : deux vecteurs sont orthogonaux s'ils sont « perpendiculaires », c.a.d. si leur produit scalaire est nul).

En général, un réseau n'admet pas de base orthogonale (c.a.d. dont les vecteurs sont deux-à-deux orthogonaux). C'est par exemple le cas de celui de la figure 1.1. À défaut, une base « réduite », qui sont formée par des vecteurs relativement courts et quasi-orthogonaux, est intéressante (on verra après pourquoi).

Considérons une base $B = (\mathbf{b}_1, \ldots, \mathbf{b}_d)$ d'un réseau $\mathcal{L}$. Si les vecteurs de la base étaient orthogonaux, alors le volume du parallélépipède fondamental qui leur est associé serait le produit de la longueur des côtés :

$$\mathrm{Vol}(\mathcal{L}) = \|\mathbf{b}_1\| \times \cdots \times \|\mathbf{b}_d\|.$$

Mais en général, quand une base n'est pas orthogonale, le produit de la longueur de ses vecteurs est *plus grand* que le volume du réseau. Pour le réseau de la figure 1.1, dont le volume est 8, on trouve :

$$\begin{array}{rcll} \|\mathbf{a}_1\| \times \|\mathbf{a}_2\| & = & 8\sqrt{10} & \approx & 3.16\,\mathrm{Vol}(\mathcal{L}) \\ \|\mathbf{b}_1\| \times \|\mathbf{b}_2\| & = & 4\sqrt{13} & \approx & 1.8\,\mathrm{Vol}(\mathcal{L}) \\ \|\mathbf{c}_1\| \times \|\mathbf{c}_2\| & = & 4\sqrt{(5)} & \approx & 1.2\,\mathrm{Vol}(\mathcal{L}) \end{array}$$

On voit que moins les vecteurs de la base ont l'air orthogonaux, plus l'écart entre le produit de leur longueur et le volume est important. La figure 1.4 l'illustre d'une autre manière : étant donné une base B, on peut « l'orthogonaliser » (par exemple avec le procédé de Gram-Schmidt, ou bien en calculant une factorisation QR) et obtenir une base B^* qui engendre le même espace vectoriel. On voit que les vecteurs de B^* sont plus courts.

Ceci permet de définir le *défaut d'orthogonalité* d'une base $B = (\mathbf{b}_1, \ldots, \mathbf{b}_d)$: c'est le nombre $\|\mathbf{b}_1\| \times \cdots \times \|\mathbf{b}_d\| / \mathrm{Vol}(\mathcal{L})$. Si B est une base orthogonale, alors ce nombre vaut 1. Plus il est grand, moins la base est orthogonale.

Le mathématicien Charles Hermite (1822–1901), pionnier de l'étude des réseaux euclidiens, a démontré que tout réseau (de rang plein) admet des bases dont le défaut d'orthogonalité est plus petit que $(4/3)^{n(n-1)/4}$ (ce qui est

2. c.a.d. l'ensemble des points qui sont à distance $\leq r$ du centre, quel que soit le nombre de dimensions...

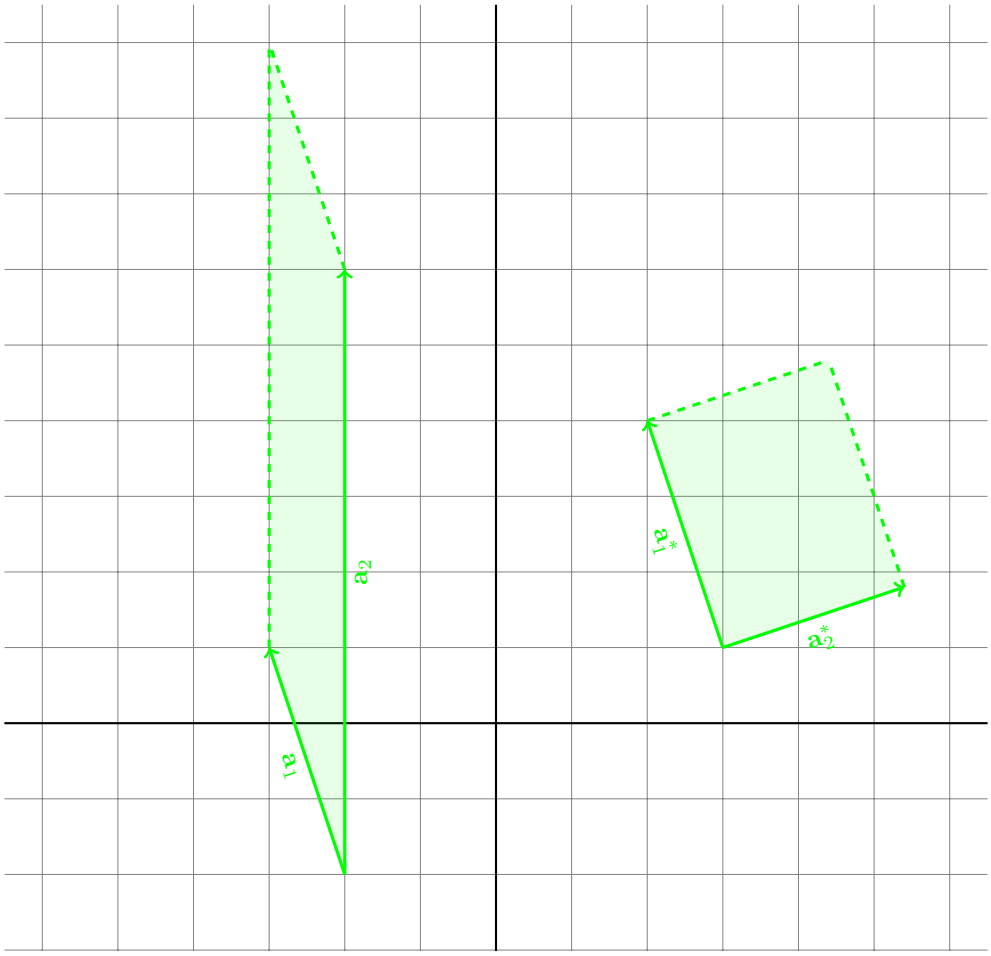

FIGURE 1.4 – Illustration du défaut d'orthogonalité et de l'orthogonalisation d'une base. Les deux zones vertes ont le même volume (8) et $\mathbf{a}_2^*$ est beaucoup plus court que $\mathbf{a}_2$.

fort, c'est que ça ne dépend que de la dimension, pas de la taille des coefficients dans n'importe quelle matrice de base). Mais en général, on ne sait pas comment les calculer en temps polynomial.

1.1.4 Plus courts vecteurs

Dans un réseau euclidien, on peut parler du *plus court vecteur* (non-nul), alors que ça n'aurait pas de sens dans un espace vectoriel. Dans la figure 1.1, on voit bien que c'est $\mathbf{b_2} = (2, 2)$.

La longueur du plus court vecteur d'un réseau $\mathcal{L}$ en est une caractéristique très importante. On l'appelle le *premier minimum* et on la note $\lambda_1(\mathcal{L})$.

On verra plus tard que ce n'est pas facile de trouver explicitement un plus court vecteur dans un réseau quelconque. Cependant, on peut dire des choses sur sa taille. Une première approche consiste à utiliser l'heuristique Gaussienne : on peut estimer le nombre de points du réseau dans une boule de rayon r ; on fait baisser r jusqu'à ce que ce nombre vaille 1 ; alors r est une estimation de la taille du plus court vecteur du réseau. Dit autrement : on s'attend à ce qu'une boule de rayon $\lambda_1(\mathcal{L})$ ait sensiblement le même volume que le réseau.

Notons ν_n le volume de la boule unité (c.a.d. de rayon 1) en n dimensions. Le volume d'une boule de rayon r est $r^n\nu_n$. Le raisonnement ci-dessus dit qu'on devrait avoir $\mathrm{Vol}(\mathcal{L}) \approx \nu_n \lambda_1(\mathcal{L})^n$, autrement dit

$$\lambda_1(\mathcal{L}) \approx \left(\frac{\mathrm{Vol}(\mathcal{L})}{\nu_n}\right)^{\frac{1}{n}}$$

Pour aller plus loin, il faudrait connaître ν_n. Nous savons depuis le collège que $\nu_2 = \pi$ et $\nu_3 = \frac{4}{3}\pi$, mais au-delà on admet que

$$\nu_n \sim \frac{1}{\sqrt{\pi n}} \left(\frac{2e\pi}{n}\right)^{n/2}.$$

En combinant ceci avec le raisonnement précédent, on trouve :

$$\lambda_1(\mathcal{L}) \approx \sqrt{\frac{n}{2e\pi}} \left(\sqrt{\pi n} \operatorname{Vol}(\mathcal{L})\right)^{\frac{1}{n}}.$$

Et si on fait sauter tous les petits facteurs embêtant, on trouve $\lambda_1(\mathcal{L}) \approx \sqrt{n} \operatorname{Vol}(\mathcal{L})^{\frac{1}{n}}$. Cette estimation est très pratique, mais il faut garder à l'esprit qu'il peut y avoir des cas particuliers où elle est complètement fausse.

Heureusement, on peut rendre ceci rigoureux, grâce à un théorème de Hermann Minkowski (1864–1909).

Théorème 2 (conséquence du théorème du corps convexe de Minkowski)**.** *Un réseau $\mathcal{L}$ de dimension d dans $\mathbb{R}^n$ contient un vecteur non-nul $\mathbf{x}$ tel que*

$$\|\mathbf{x}\| \leq 2 \left(\frac{\operatorname{Vol}(\mathcal{L})}{\nu_d}\right)^{\frac{1}{d}}$$

La preuve de ce théorème est non-constructive et ne dit pas *comment* on pourrait trouver le vecteur court dont l'existence est garantie.

Il faut noter que le théorème de Minkowski donne une borne supérieure sur la taille du plus court vecteur d'un réseau. Le résultat (garanti) qu'on obtient est seulement deux fois pire que l'estimation au doigt mouillé qu'on avait tiré de l'heuristique gaussienne. Mais par contre, il n'y a pas de borne inférieure correspondante, et seule l'heuristique gaussienne permet d'estimer (avec précaution) la taille du plus court vecteur.

En gros, un réseau « aléatoire » n'a pas de raison de contenir des vecteurs sensiblement plus courts que ce que ces deux résultats suggèrent.

1.1.5 La constante de Hermite

Le théorème de Minkowski a de nombreuses conséquences importantes. Il implique en particulier que $\lambda_1(\mathcal{L}) / \operatorname{Vol}(\mathcal{L})^{1/d}$ est majoré par une constante qui dépend uniquement de la dimension, et pas de la base du réseau.

On appelle *constante de Hermite* le supremum de $\lambda_1(\mathcal{L})^2 / \operatorname{Vol}(\mathcal{L})^{2/d}$, pris sur tous les réseaux de dimension d. Les valeurs de ces constantes sont mystérieuses et mal connues. On sait que $\gamma_8 = 2$ et $\gamma_{24} = 4$, mais aucune valeur plus grande n'est connue avec certitude. On ne sait même pas si la séquence est croissante ! On sait cependant que $\gamma_d \leq 1 + d/4$.

L'intérêt, c'est que n'importe quel réseau de dimension d contient un vecteur de longueur inférieure à $\lambda_1(\mathcal{L}) \leq \sqrt{\gamma_d} \operatorname{Vol}(\mathcal{L})^{1/d}$.

1.2 Problèmes algorithmiques dans les réseaux

Il y a de nombreux problèmes algorithmiques difficiles qui tournent autour des réseaux, mais les deux principaux sont les plus simples :

Définition 2 : Shortest Vector Problem (SVP)

> Étant donné une base d'un réseau $\mathcal{L}$, calculer un plus court vecteur de $\mathcal{L}$, c.a.d. un vecteur $\mathbf{x}$ tel que $\|\mathbf{x}\| = \lambda_1(\mathcal{L})$.

Définition 3 : Closest Vector Problem (CVP)

> Étant donné une base d'un réseau $\mathcal{L}$ et un vecteur $\mathbf{x}$, calculer un vecteur de $\mathcal{L}$ le plus proche possible de $\mathbf{x}$, c.a.d. un vecteur $\mathbf{y}$ tel que $\|\mathbf{x} - \mathbf{y}\|$ soit minimal.

Il faut noter qu'il s'agit de deux problèmes d'optimisation, et que même si on nous donnait la solution, il n'est pas évident de se convaincre qu'elle est optimale.

Les deux problèmes sont NP-durs (le paramètre principal est la dimension du réseau). Pour SVP, cela a été conjecturé dès 1981, mais démontré seulement en 1998 par Miklós Ajtai (et encore, avec des réductions randomisées) [2]. Démontrer que CVP est NP-dur est assez facile (réduction à Partition), la preuve date de 1981 (par Peter van Emde Boas).

Dans la pratique, les deux problèmes sont *faciles* à résoudre en petite dimension ($n \leq 50$ disons), mais *complètement impossible* à résoudre en grande dimension ($n \geq 200$ disons). Pour SVP, il y a deux familles d'algorithmes qui sont utilisables : les algorithmes *d'énumération* qui fonctionnent en temps $2^{\mathcal{O}(n \log n)}$ et en espace faible

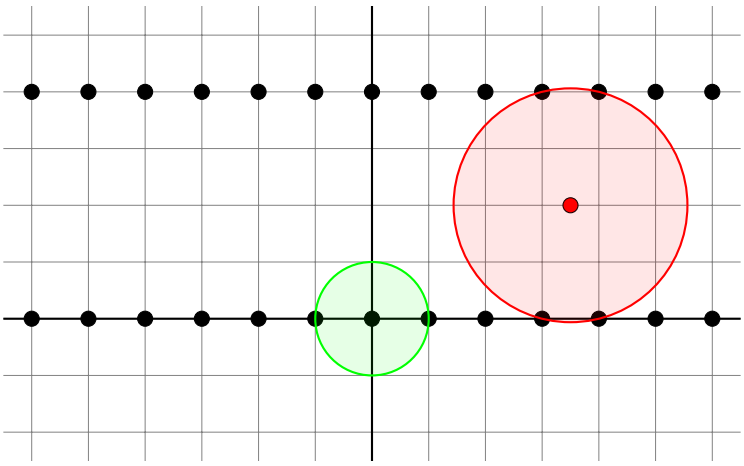

FIGURE 1.5 – SVP vs CVP

d'une part, et les algorithmes de *crible* qui fonctionnent en temps et en espace $2^{\mathcal{O}(n)}$ — en pratique l'espace devient vite un problème.

On peut considérer aussi des versions approchées des deux problèmes en question.

Définition 4 : Approximate Shortest Vector Problem ($\mathbf{SVP}_\gamma$)

> Étant donné une base d'un réseau $\mathcal{L}$ et un paramètre $\gamma \geq 1$, calculer un vecteur $\mathbf{x}$ tel que $\|\mathbf{x}\| \leq \gamma \cdot \lambda_1(\mathcal{L})$.

Définition 5 : Approximate Closest Vector Problem ($\mathbf{CVP}_\gamma$)

> Étant donné une base d'un réseau $\mathcal{L}$, un vecteur $\mathbf{x}$ et un paramètre $\gamma \geq 1$ calculer un vecteur $\mathbf{y} \in \mathcal{L}$ tel que pour tout $\mathbf{z} \in \mathcal{L}$ on a : $\|\mathbf{x} - \mathbf{y}\| \leq \gamma \cdot \|\mathbf{x} - \mathbf{z}\|$ ($\mathbf{y}$ est au pire γ fois trop loin de $\mathbf{x}$).

Plus le facteur d'approximation est grand, plus les problèmes approchés sont faciles. Si γ est une constante (c.a.d. indépendant de la dimension du réseau), les problèmes approchés restent NP-durs. Plus précisément, Dinur, Kindler Raz et Safra ont montré en 2003 [4] que CVP est NP-dur tant que $\gamma \leq n^{c/\log\log n}$ pour n'importe quelle constante c.

Au contraire, on va voir qu'il existe des algorithmes polynomiaux pour résoudre les problèmes approchés, qui n'atteignent que des facteurs d'approximations exponentiellement grands. Et il y a des cas intermédiaires, comme $\gamma = \sqrt{n/\log n}$, qui n'est probablement pas NP-dur, mais pour lequel on ne connaît pas d'algorithme d'approximation efficace.

Enfin, il y a un dernier problème algorithmique intéressant, dérivé du théorème 2.

Définition 6 : Hermite Shortest Vector Problem ($\mathbf{HSVP}_\gamma$)

> Étant donné une base d'un réseau $\mathcal{L}$ de dimension d et un paramètre $\gamma \geq 1$, calculer un vecteur $\mathbf{x}$ tel que $\|\mathbf{x}\| \leq \gamma \times \mathrm{Vol}(\mathcal{L})^{1/d}$.

L'avantage de ce dernier problème est qu'il est facile de vérifier les solutions... et qu'il est dur quand même. En plus, il a été démontré (par László Lovasz en 1986) que si on sait approximer HSVP avec facteur γ, on peut approximer SVP avec facteur γ^2 [11].

On peut remarquer que CVP est en quelque sorte plus général que SVP : dans SVP, on cherche le vecteur le plus proche du point de coordonnées $(0, \ldots, 0)$, tandis que dans CVP on chercher à partir d'un point arbitraire de l'espace.

Il faut noter que les deux problèmes peuvent avoir des réponses assez différentes : la distance d'un point quelconque de l'espace au reste du réseau peut être sensiblement plus grande que le premier minimum (cf. figure 1.5)

1.3 Algorithme LLL et améliorations

En 1982, Hendrik Lenstra [3] (né en 1949), Arjen Lenstra [4] (né en 1956) et László Lovász (né en 1948) ont inventé le célèbre algorithme qui porte leurs initiales.

L'algorithme LLL prend en entrée une base quelconque d'un réseau ainsi qu'un paramètre $\frac{1}{4} \leq \delta < 1$. Il s'exécute en temps polynomial (en le nombre de bits nécessaire pour écrire la base) et produit en sortie une base « réduite avec un facteur δ ». On ne décrira pas ici l'algorithme.

Par exemple, avec la base suivante, dont tous les vecteurs sont assez grands :

$$B = \begin{pmatrix} 1 & 0 & 0 & 0 & 2121390710 \\ 0 & 1 & 0 & 0 & 365500767 \\ 0 & 0 & 1 & 0 & 2647889201 \\ 0 & 0 & 0 & 1 & 1407355715 \\ 0 & 0 & 0 & 0 & 2585271343 \end{pmatrix},$$

l'exécution de l'algorithme LLL produit la base « réduite » :

$$LLL(B) = \begin{pmatrix} -7 & 28 & -35 & 3 & -36 \\ -4 & -51 & 24 & -2 & -22 \\ 50 & -8 & -29 & -72 & 2 \\ -36 & 33 & 41 & -59 & -48 \\ -70 & -41 & -65 & 7 & 11 \end{pmatrix}$$

dont les vecteurs sont beaucoup plus courts. Les deux matrices engendrent le même réseau, mais la deuxième en est une description beaucoup plus « commode ».

La complexité dans le pire des cas des meilleures implantations de LLL est de $\mathcal{O}\left(d^4 n \log B.(d + \log B)\right)$, où B désigne le nombre de bits des entrées de la matrice représentant la base du réseau fournie en entrée. Autant le dire tout de suite : l'algorithme est polynomial, mais ça peut coûter cher quand même dans le pire des cas. Souvent, en pratique, la complexité empirique est meilleure que ce que cette borne suggère.

Sur le plan pratique, l'algorithme LLL est disponible dans plusieurs logiciels open-source. Le plus facile d'utilisation est certainement `SageMath` (avec une syntaxe qui est quasiment du `python`). Il existe aussi une librairie `C` nommée `fplll` avec un utilitaire en ligne de commande, et surtout un paquet `python fpylll` qui permet de l'utiliser dans `python`. En fait, `fpylll` est inclus dans `SageMath`. Outre l'algorithme LLL, il contient aussi des algorithmes pour résoudre exactement CVP et SVP (en petite dimension bien sûr).

L'intérêt de l'algorithme LLL est donné tout entier dans le théorème suivant :

Théorème 3. *Soit $\alpha = 1/(\delta - \frac{1}{4})$ et soit $(\mathbf{b}_1, \ldots, \mathbf{b}_d)$ une base LLL-reduite avec un facteur δ d'un réseau euclidien $\mathcal{L}$. Alors :*

1. $\|\mathbf{b}_1\| \leq \alpha^{(d-1)/4} \left(\mathrm{Vol}(\mathcal{L})\right)^{1/d}$.
2. $\|\mathbf{b}_1\| \leq \alpha^{(d-1)/2} \lambda_1(\mathcal{L})$.
3. $\|\mathbf{b}_1\| \times \cdots \times \|\mathbf{b}_d\| \leq \alpha^{d(d-1)/4} \mathrm{Vol}(\mathcal{L})$.

Voici quelques commentaires.

Historiquement, l'algorithme LLL a été présenté avec $\delta = \frac{3}{4}$, ce qui donne $\alpha = 2$. On ne sait pas si l'algorithme est toujours polynomial avec $\delta = 1$ (qui donnerait $\alpha = \frac{4}{3}$). En pratique, $\delta = 0.99$ fonctionne assez bien, et c'est généralement le réglage par défaut dans les implantations efficaces de LLL.

Le premier vecteur d'une base LLL-reduite est relativement court. Il est au pire $\left(\alpha^{1/4}\right)^d$ fois plus long que la « borne de Minkowski » donnée par le théorème 2. Ceci implique que l'algorithme LLL permet de résoudre HSVP avec « facteur de Hermite » $\approx (4/3)^{d/4} \approx 1.0745^d$ (il est exponentiel en la dimension).

Mais le plus intéressant est le deuxième point : le premier vecteur d'une base LLL-réduite est $\sqrt{\alpha}^d$ fois plus long que le plus court vecteur du réseau. Ceci implique que l'algorithme LLL permet de résoudre SVP avec facteur d'approximation $\approx \sqrt{4/3}^d \approx 1.1547^d$ (il est encore exponentiel en la dimension).

Par conséquent, l'algorithme LLL nous donne accès à un moyen simple de calculer des vecteurs « relativement courts » d'un réseau en temps polynomial : il suffit de prendre le premier vecteur d'une base LLL-réduite.

3. Il a aussi mis au point l'algorithme de factorisation basé sur les courbes elliptiques, qui est le meilleur pour trouver de petits facteurs premiers...

4. Il a aussi co-inventé le meilleur algorithme de factorisation des grands entiers, le crible algébrique...

Au passage, si un réseau contient *un* vecteur très court, et que tous les autres sont au moins $2^{d/2}$ fois plus longs, alors le deuxième point du théorème affirme que l'algorithme LLL va trouver ce vecteur très court en temps polynomial (car c'est le seul qui peut satisfaire le 2ème point du théorème).

Le troisième point du théorème montre que le défaut d'orthogonalité des bases LLL-reduites est majoré par $\alpha^{d^2/4}$. Autrement dit, l'algorithme LLL, à partir d'une base quelconque en entrée, produit une base dont le défaut d'orthogonalité est contrôlé (et donc, dont les vecteurs sont plutôt courts).

Il faut noter que ces résultats sont valables dans le pire des cas. En pratique, l'algorithme LLL se comporte souvent mieux que prévu, et en particulier il peut renvoyer une base composée de vecteurs sensiblement plus courts que ce que le théorème annonce. En pratique, on observe plutôt un facteur de Hermite de l'ordre de 1.0219^n [6]. Pour des réseaux ayant des structures spéciales, la situation peut être encore plus favorable.

En deux dimensions ($d = 2$), LLL renvoie prouvablement le plus court vecteur du réseau.

Le facteur d'approximation pour SVP est généralement le carré de celui pour HSVP : de toute façon c'est compatible avec la réduction entre les deux.

1.3.1 Algorithme BKZ

Il existe un autre algorithme qui produit des bases améliorées : il s'agit de la BKZ-réduction (pour Block-Korkine-Zolotarev), dûe à Schnorr et Euchner en 1994 [14].

Il prend en argument un paramètre supplémentaire β (la taille du « bloc »), et son temps d'exécution est fortement exponentiel en β. En pratique, il produit des bases plus réduites. L'algorithme BKZ, s'il termine, atteint prouvablement un « facteur de Hermite » (approximation pour HSVP) de $\sqrt{\gamma_\beta}^{1+(n-1)/(\beta-1)}$. Par exemple, avec $\beta = 24$, cela donne $\approx 1.0305^n$. Encore une foi, la pratique est meilleure que la théorie : le facteur observé empiriquement est plutôt de l'ordre de 1.0125^n.

De même, si BKZ termine, il permet d'approximer SVP avec facteur $\gamma_\beta^{(n-1)/(\beta-1)}$. Avec $\beta = 24$, cela donne $\approx 1.0621^n$. Ces résultats sont décrits dans [6].

En pratique, jusqu'à quelle dimension peut-on résoudre HSVP avec un facteur d'approximation linéaire ? Avec LLL on doit pouvoir monter jusqu'à la dimension 250. Avec BKZ-24, on doit pouvoir monter jusqu'à la dimension 500.

1.4 Algorithmes polynomiaux pour approcher CVP

Dans cette section, on considère un réseau $\mathcal{L}$, de rang plein, dont on possède une base $B = (\mathbf{b_1}, \ldots, \mathbf{b_n})$, et on cherche à résoudre CVP dedans de manière approchée, pour un vecteur « cible » $\mathbf{y}$.

Une première remarque c'est que la situation est beaucoup plus facile si la base B qu'on nous fournit est orthogonale. Par exemple, si c'est le cas, alors le plus court vecteur du réseau est forcément déjà dedans ! De même, CVP est facile aussi à résoudre dans ce cas-là.

Poursuivant cette idée-là, László Babai (né en 1950) a observé en 1986 [3] que même si on ne dispose pas d'une base orthogonale, mais d'une base de « bonne qualité » (défaut d'orthogonalité faible), alors on peut espérer s'en tirer assez bien en pratique, au moins tant que la dimension n'est pas trop grande. Ceci lui a permis de proposer deux algorithmes simples pour approcher CVP : la méthode de l'arrondi et la méthode de l'hyperplan le plus proche.

1.4.1 Méthode du plongement

Il s'agit d'une technique heuristique pour résoudre CVP en le ramenant à SVP. On forme le réseau $\mathcal{L}'$ de dimension $n + 1$ engendré par

$$B' = \begin{pmatrix} \mathbf{b}_1 & 0 \\ \mathbf{b}_2 & 0 \\ \vdots & \\ \mathbf{b}_n & 0 \\ \mathbf{y} & 1 \end{pmatrix}$$

Si $\mathbf{z}$ est le vecteur le plus court de $\mathcal{L}$, alors $(\mathbf{z}, 0)$ appartient à $\mathcal{L}'$ et il a la même longueur. Les deux réseaux ont par ailleurs le même volume (car $\det B' = \det B$) et (à un près) ils ont la même dimension. On pourrait donc s'attendre à ce que leurs plus courts vecteurs soient à peu près de la même taille.

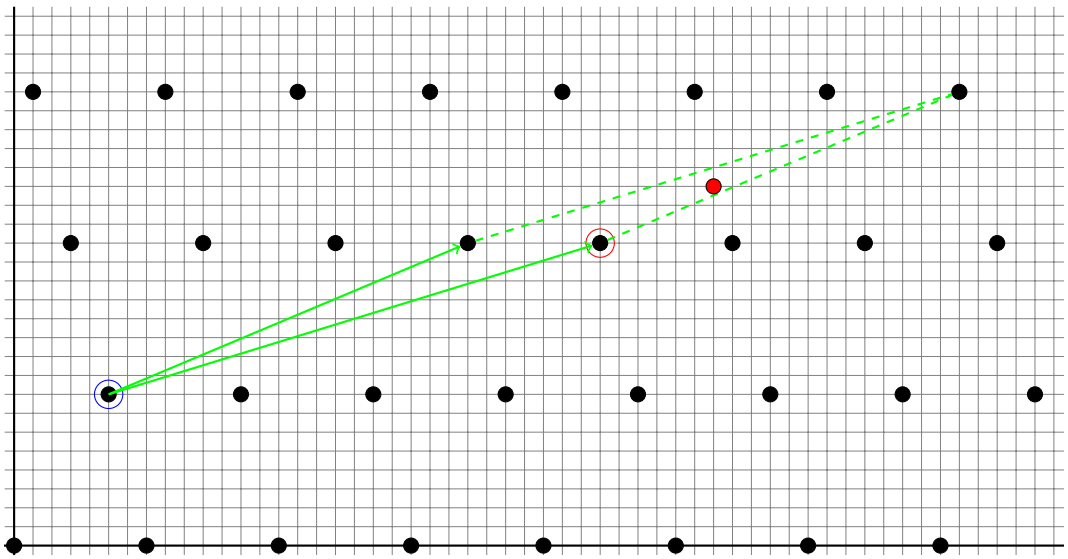

FIGURE 1.6 – Tentative de résolution de CVP avec la méthode de l'arrondi. La cible est le point rouge. Comme la base est très mauvaise, le sommet du parallélépipède le plus proche de la cible... en est assez éloigné, et ce n'est pas le CVP.

Considérons la solution $\mathbf{x} \in L$ de notre instance de CVP. Le vecteur $(\mathbf{y} - \mathbf{x}, 1)$ appartient à $\mathcal{L}'$, et il est « court » (sa longueur est la distance entre $\mathbf{y}$ et le point du réseau le plus proche).

Ainsi, si la distance de $\mathbf{y}$ au réseau $\mathcal{L}$ est bien plus petite que $\lambda_1(\mathcal{L})$, on pourrait espérer que ce vecteur $(\mathbf{y} - \mathbf{x}, 1)$ est un plus court vecteur de $\mathcal{L}'$. Si tel était le cas, alors on pourrait résoudre notre instance de CVP en résolvant SVP dans $\mathcal{L}'$.

Et si $\mathbf{y}$ était *vraiment* très près d'un point du réseau $\mathcal{L}$, alors même un algorithme qui résout SVP approximativement (comme LLL) permettrait de révéler la solution exacte.

Il faut noter que cette méthode est heuristique, et il y a des cas où elle échoue (de manière déterministe).

1.4.2 Méthode de l'arrondi

Puisque $\mathcal{B}$ est une base de $\mathbb{R}^n$ (en tant qu'espace vectoriel), on peut écrire

$$\mathbf{y} = \sum_{i=1}^{n} \lambda_i \mathbf{b}_i,$$

où les λ_i sont des *réels* (et pas forcément des entiers). Maintenant, si on *arrondit* les λ_i, alors on obtient un point du réseau !

Si on arrondit vers zéro, alors on obtient un point $\mathbf{b}$ (entouré en bleu sur la figure 1.6) tel que la cible $\mathbf{y}$ est contenue dans le parallélépipède fondamental ancré en $\mathbf{b}$. En arrondissant au plus près, on obtient un sommet du parallélépipède fondamental (entouré en rouge) qui est plus proche de la cible.

La figure 1.6 montre que si la base est « mauvaise » (vecteurs longs et peu orthogonaux), ça ne risque pas de bien marcher. En effet, si le parallélépipède est très étiré, son sommet le plus proche de la cible peut en être assez éloigné (plus en tout cas que d'autres points en dehors du parallélépipède).

Notons $\lfloor x \rceil$ l'entier le plus proche de x. Il est clair que

$$\mathbf{x} = \sum_{i=1}^{n} \lfloor \lambda_i \rceil \, \mathbf{b}_i$$

est un point du réseau $\mathcal{L}$. Le problème consiste à voir s'il est loin ou près de $\mathbf{y}$. Pour cela on écrit :

$$\|\mathbf{x} - \mathbf{u}\| = \left\| \sum_{i=1}^{n} (\lambda_i - \lfloor \lambda_i \rceil) \, \mathbf{b}_i \right\| \leq \sum_{i=1}^{n} \|\lambda_i - \lfloor \lambda_i \rceil \, \mathbf{b}_i\| \leq \frac{1}{2} \sum_{i=1}^{n} \|\mathbf{b}_i\|$$

(Le passage à la première inégalité utilise simplement l'inégalité triangulaire ; pour passer à la second on utilise le fait que l'entier le plus proche est forcément à distance $\leq 1/2$). On voit que plus les vecteurs de la base $\mathcal{B}$ sont courts, plus cette technique va fournir un vecteur proche.

Ce qui n'est pas facile, c'est de se convaincre que le facteur d'approximation est garanti. En fait, il est possible (mais difficile) de démontrer que :

Théorème 4 (Babai, 1986)**.** *La méthode de l'arrondi utilisée avec des bases LLL-réduites avec un facteur* $\delta = 3/4$ *approxime CVP avec un facteur* $1 + 2d(9/2)^{d/2}$.

On ne décrira pas ici l'autre algorithme simple (et polynomial) dû à Babai, la méthode de l'hyperplan le plus proche. Le résultat est en fait meilleur, car avec une base LLL-réduite avec facteur δ, on peut atteindre en temps polynomial un facteur d'approximation pour CVP $\gamma \leq \sqrt{n}\alpha^{-n/2}$, où $\alpha = 1/(\delta - 1/4)$, comme dans LLL.

Grosso-modo, tout ce que LLL donne pour SVP, on peut l'obtenir aussi pour CVP.

Chapitre 2

Applications à la cryptanalyse

On considère ici des attaques à base de réseaux euclidiens contre diverses constructions : générateurs pseudo-aléatoires, fonctions de hachage et chiffrement.

2.1 Générateurs Pseudo-aléatoires

Informellement, un PRG (Pseudo-Random Generator) est un algorithme *déterministe* (paradoxalement) qui, à partir d'une courte chaine de bits aléatoire, sa *graine* (« seed »), est capable de produire un nombre plus grand de bits pseudo-aléatoires, c.a.d. impossibles à distinguer efficacement de bits « vraiment » aléatoires.

Les PRG sont des outils indispensables en cryptographie. Pour ne donner qu'un exemple, la génération d'une paire de clefs RSA de n bits (en trouvant les deux nombres premiers par rejection sampling) nécessite environ $3n^2/2$ bits aléatoire, ce qui donne plus de 6 millions pour une clef de 2048 bits. En fait, on utilise systématiquement un PRG initialisé à partir d'une graine de 128 ou 256 bits.

2.1.1 Notes historiques

Les PRG ont été étudiés depuis les débuts de l'informatique. En 1946, Von Neumann et Metropolis, qui travaillaient sur l'ENIAC, ont proposé la méthode du « *middle-square* » : si u_n est un nombre de k chiffres, former u_{n+1} en prenant les $\frac{k}{2}$ chiffres du milieu de u_n et en élevant ce nombre au carré. Le problème de cette méthode, c'est que la séquence fournie entre rapidement dans un cycle court.

Un peu plus tard, Lehmer a proposé les *générateurs congruentiels linéaires*. Il a proposé la séquence définie par $X_0 = 47594118, X_{n+1} = 23X_n \bmod 10^8 + 1$, et démontré que sa période est de 5'882'353.

Ce genre de PRG est répandu. La plupart des langages de programmation comportent des fonctions simples qui permettent d'obtenir des séquences de nombre réputés pseudo-aléatoires. En C, c'est un générateur congruentiel linéaire (tronqué) qu'il y a dans la fonction `rand()`. Python et PHP en utilisent un autre, le « Mersenne Twister ».

Malheureusement, aucun de ces PRGs ne sont cryptographiquement sûrs. Par exemple, après avoir observé plusieurs fois la sortie de `rand()`, il est aisé de produire *soi-même* toutes les prochaines valeurs. Il y a donc un algorithme qui, à partir d'un préfixe de la séquence de bits pseudo-aléatoires, est capable de *prédire* la suite de la séquence — avec probabilité 1.

Un tel phénomène de prédiction est impossible avec une séquence de bits « vraiment aléatoires ». Par définition, un phénomène aléatoire est imprédictible.

On verra en TD que tous les générateurs linéaires congruentiels sont faibles. Plus précisément, si on nous donne $X_0, X_1, \ldots, X_n$ dans la séquence $X_{i+1} = aX_i + b \bmod m$, où a, b et m sont *inconnus*, alors il est facile de tout reconstituer et ainsi de prédire la suite de la séquence.

2.1.2 Définition et notion de sécurité

Généralement, lorsqu'on parle *d'ensembles probabilistes*, on parle en fait d'ensembles de variables aléatoires $\{X_n\}_{n\in\mathbb{N}}$ à valeur dans $\{0,1\}^*$ et telles que $|X_n| = \text{poly}(n)$.

Définition 7 : Indistinguabilité calculatoire

Deux ensembles probabilistes $\{X_n\}_{n\in\mathbb{N}}$ et $\{Y_n\}_{n\in\mathbb{N}}$ sont **calculatoirement indistinguables** si pour tout algorithme polynomial randomisé, tout polynôme positif $p(\cdot)$ et tous n suffisamment grand, on a :

$$\left|\mathbb{P}\left[\mathcal{A}(1^n, X_n) = 1\right] - \mathbb{P}\left[\mathcal{A}(1^n, Y_n) = 1\right]\right| < \frac{1}{p(n)}.$$

La grandeur dans le membre gauche est l'**avantage** du distingueur $\mathcal{A}$.

Cette définition stipule que si $\{X_n\}_{n\in\mathbb{N}}$ et $\{Y_n\}_{n\in\mathbb{N}}$ sont calculatoirement indistinguables, alors aucun algorithme efficace ne peut avoir un comportement très différent selon qu'on lui donne X_n ou Y_n. Et ici, « avoir un comportement différent » signifie que la probabilité que l'algorithme $\mathcal{A}$ renvoie 1 doit être presque la même dans les deux cas. A contrario, si $\mathcal{A}$ était capable, la plupart du temps, de renvoyer « 0 » quand on lui donne X_n mais « 1 » quand on lui donne Y_n, alors il *distinguerait* les deux ensembles probabilistes.

La sécurité est définie de manière asymptotique lorsque n peut devenir arbitrairement grand. L'algorithme $\mathcal{A}$ reçoit la chaine de bits 1^n, ce qui lui permet de fonctionner en temps polynomial en n, même si X_n est plus petit.

Par exemple, l'hypothèse Diffie-Hellman Décisionnelle (DDH) pour un groupe $\langle g\rangle$ d'ordre $q_n > 2^n$, affirme que les ensembles $\left\{\left(g^{A_n}, g^{B_n}, g^{A_nB_n}\right)\right\}_n$ et $\left\{\left(g^{A_n}, g^{B_n}, g^{C_n}\right)\right\}$ sont calculatoirement indistinguables (A_n, B_n et C_n sont trois variables aléatoires uniformément distribuées dans $\mathbb{Z}_{q_n}$).

Notons U_n une variable aléatoire uniformément distribuée dans $\{0,1\}^n$.

Définition 8 : Ensemble pseudo-aléatoire

Un ensemble probabiliste (de chaines de bits) $\{X_n\}_{n\in\mathbb{N}}$ est **pseudo-aléatoire** s'il est calculatoirement indistinguable de $\{U_{|X_n|}\}_{n\in\mathbb{N}}$.

« Pseudo-aléatoire » signifie donc : « aucun algorithme efficace ne peut faire la différence avec des bits *vraiment* aléatoires ».

Définition 9 : Générateur pseudo-aléatoire avec taille arbitraire

Un **générateur pseudo-aléatoire** est un algorithme déterministe polynomial G qui satisfait les deux conditions suivantes :

1. *Sortie arbitrairement grande* : pour toute graine $s \in \{0,1\}^*$ et tout $t \in \mathbb{N}$, on a $|G(s, 1^t)| = t$ et la chaine de bits $G(s, 1^t)$ est un préfixe de $G(s, 1^{t+1})$.
2. *Pseudo-aléatoire* : pour tout polynôme p, l'ensemble $\{G(U_n, 1^{p(n)})\}_{n\in\mathbb{N}}$ est pseudo-aléatoire.

Il faut noter que les PRGs les plus courants dans les langages de programmation, et en particulier la fonction `rand()` du langage C, ne satisfont *PAS* cette définition.

Il y a d'abord un obstacle mineur. La définition exige que la sortie pseudo-aléatoire soit produite d'un seul coup, alors que la plupart du temps on peut l'obtenir au fur-et-à-mesure. Ceci n'est pas grave.

Plus problématique, dans les PRGs « concrets », la taille de la graine est fixée, ce qui ne rentre pas dans le cadre (où la sécurité est définie asymptotiquement lorsque la taille de la graine augmente). Ensuite, il y a le problème de la sécurité : il existe des algorithmes efficaces qui distinguent la sortie de ces PRGs de bits uniformément aléatoires.

Il est possible de construire un PRG sûr sous l'hypothèse que le problème du logarithme discret est difficile ou que la factorisation des grands entiers est difficile. Ces construction sont (extrêmement) rarement utilisées en pratique. On verra plus tard des exemples de PRG cryptographiques raisonnables.

2.2 Générateurs Linéaires Congruentiels Tronqués

2.2.1 Description

Au début des années 1980, Don Knuth (né en 1938) a suggéré la technique suivante pour produire des générateurs pseudo-aléatoires de qualité cryptographique [8]. On considère la séquence :

$$X_{i+1} \leftarrow (aX_i + b) \bmod m \qquad \text{et} \qquad Y_i \leftarrow \lfloor X_i/k \rfloor$$

On suppose que a, b, k et m sont connus. La graine du PRNG est X_0, et la séquence de ses sorties est $Y_0, Y_1, Y_2, \ldots$

C'est un générateur congruentiel linéaire *tronqué* : lors de chaque sortie, une partie de X_i nous est dissimulée, puisqu'on ne récupère que le quotient de la division euclidienne de X_i par k. Si $k = 2^\ell$, cela revient à cacher les ℓ bits de poids faible. Écrivons donc $X_i = kY_i + Z_i$ (où $0 \leq Z_i < k$ est la partie tronquée).

2.2.2 Instances

Plusieurs PRNG très connus utilisent ce principe, notamment dans la librairie C standard (les fonctions `rand()` et `mrand48()`). La norme POSIX spécifie que `mrand48` doit être implanté de la façon suivante :

```
uint64_t rand48_state;

void srand48(uint32_t seed) {
  rand48_state = seed;
  rand48_state = 0x330e + (rand48_state << 16);
}

uint32_t mrand48() {
  rand48_state = (0x00000005deece66d * rand48_state + 11) & 0x0000ffffffffffff;
  return (rand48_state >> 16);
}
```

Avec nos notations ci-dessus, cela revient à :

$$a = 25214903917, \quad b = 11, \quad m = 2^{48}, \quad k = 2^{16}$$

L'avantage, c'est que `mrand48` renvoie des entiers 32 bits. La spécification du langage C prévoit qu'une fonction `rand` soit également disponible. L'implantation suivante est suggérée dans la spécification

```
static unsigned long int next = 1;

int rand(void) { /* RAND_MAX assumed to be 32767 */
    next = next * 1103515245 + 12345;
    return ((unsigned)(next/65536) % 32768);
}

void srand(unsigned int seed) {
    next = seed;
}
```

Avec nos notations ci-dessus, cela revient à :

$$a = 1103515245, \quad b = 12345, \quad m = 2^{31}, \quad k = 2^{16}$$

Ces générateurs pseudo-aléatoires sont faibles, et il est possible de récupérer la graine en observant très peu de flux pseudo-aléatoire. Ceci permet de prédire la suite du flux pseudo-aléatoire, et même de connaître le flux pseudo-aléatoire généré avant l'observation.

2.2.3 Attaque lorsque $b = 0$

Le raisonnement ci-dessous est une version simplifiée de l'analyse due à Frieze, Håstad, Kannan, Lagarias et Shamir en 1988 [5].

Notre objectif consiste à récupérer l'un des X_i, donc en fait l'un des Z_i. On ne connaît pas entièrement X_i, mais on en connaît une *approximation*, qui est $k \times Y_i$. En effet, $|X_i - kY_i| < k$, donc ces deux nombres sont « proches ».

Pour exploiter cette observation, le plan de bataille est le suivant :

- Fabriquer un réseau euclidien $\mathcal{L}$ qui contient le vecteur $\mathbf{x} = (X_0, X_1, X_2, \ldots, X_n)$ — qu'on ne connaît pas.
- Remarquer que le vecteur $\mathbf{y} = k(Y_0, Y_1, \ldots, Y_n)$ —qu'on connaît— est *proche* de $\mathbf{x}$.
- Faire le pari que $\mathbf{x}$ est le point du réseau le plus proche de $\mathbf{y}$, donc qu'on pourrait récupérer les bits manquants en résolvant une instance de CVP dans $\mathcal{L}$.

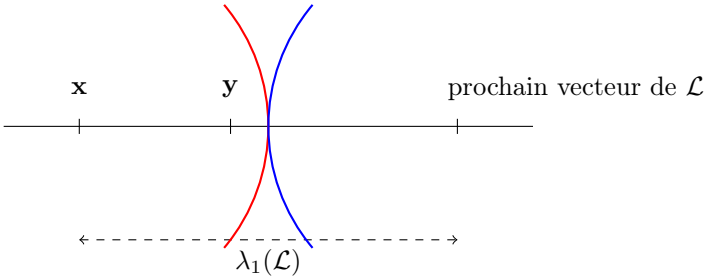

FIGURE 2.1 – Comment s'assurer que $\mathbf{x}$ est bien le plus proche vecteur de $\mathbf{y}$.

Supposons dans un premier temps que $b = 0$. On a alors $X_{i+1} \equiv aX_i \bmod m$, donc $X_i \equiv a^i X_0 \bmod m$. Comme les X_i sont des restes modulo m, il existe donc des entiers $q_1, q_2, q_3, \ldots$ (qu'on ne connaît pas) tels que :

$$\begin{aligned} X_1 &= aX_0 + q_1 m \\ X_2 &= a^2 X_0 + q_2 m \\ X_3 &= a^3 X_0 + q_3 m \\ &\vdots \end{aligned}$$

Par conséquent, on a l'égalité :

$$(X_0, q_1, q_2, \ldots, q_{n-1}) \begin{pmatrix} 1 & a & a^2 & \ldots & a^{n-1} \\ & m & & & \\ & & m & & \\ & & & \ddots & \\ & & & & m \end{pmatrix} = (X_0, X_1, X_2, \ldots, X_{n-1})$$

L'idée générale consiste à regarder le réseau $\mathcal{L}$ engendré par cette matrice. On sait que $\mathbf{x} = (X_0, X_1, X_2, \ldots, X_{n-1})$ appartient à $\mathcal{L}$.

La distance entre $\mathbf{y} = k(Y_0, Y_1, \ldots, Y_{n-1})$ et $\mathbf{x}$ est majorée par $\sqrt{k^2 + \cdots + k^2} = k\sqrt{n}$.

Le volume du réseau est m^{n-1} (le déterminant de la matrice est très facile à calculer car elle est triangulaire supérieure : c'est juste le produit des entrées sur la diagonale). On « prédit » avec l'heuristique gaussienne que le plus court vecteur de ce réseau est de taille $\lambda_1(\mathcal{L}) \approx \sqrt{n} m^{(n-1)/n}$. Par conséquent, deux points du réseau doivent être séparés d'au moins cette distance-là.

Tant que $k\sqrt{n} < \frac{1}{2}\lambda_1(\mathcal{L})$, alors il est rigoureusement certain que $\mathbf{x}$ est bien le vecteur de $\mathcal{L}$ le plus proche de $\mathbf{y}$ (cf. figure 2.1). Donc, si on fait confiance à notre estimation de $\lambda_1(\mathcal{L})$ par l'heuristique gaussienne, on s'attend à ce que $CVP(\mathcal{L}, \mathbf{y})$ renvoie vraiment $\mathbf{x}$ lorsque :

$$\sqrt{n}k < \sqrt{n} m^{(n-1)/n}$$

(remarque : on a laissé tomber le facteur $\frac{1}{2}$, mais de toute façon on avait déjà largué le $1/\sqrt{2e\pi}$ de l'heuristique gaussienne...). Ceci se simplifie en

$$\frac{\log m}{\log m - \log k} < n.$$

Ceci est logique : on veut reconstituer X_0 qui fait $\log m$ bits. Observer l'un des Y_i nous apprend $\log m - \log k$ bits. Donc l'inéquation ci-dessus montre que l'attaque (qui consiste à résoudre CVP) fonctionne dès qu'on a observé le nombre minimal de Y_i qui permet de reconstituer X_0 de manière unique.

Il faut donc résoudre une instance de CVP. Si la dimension est petite, ce sera possible avec les algorithmes exacts.

Si on ne nous donne que le bit de poids fort et que la graine fait 128 bits, par contre, on va souffrir.

2.2.4 (*) Attaque lorsque $b \neq 0$

L'idée générale consiste à se ramener au cas précédent. Pour cela, il faut faire sauter le b. Si on ne le connaît pas, on peut poser $Y_i = X_{i+1} - X_i$, car alors on a $Y_{i+1} = aY_i \bmod m$. Mais on parvient alors à reconstituer la différence entre deux états successifs et on n'est pas complètement tiré d'affaire.

Pat contre, si on connaît b, la situation est plus simple. En déroulant la récurrence, on trouve :

$$\begin{aligned} X_1 &\equiv aX_0 + b \mod m \\ X_2 &\equiv a^2 X_0 + (a+1)b \\ X_3 &\equiv a^3 X_0 + (a^2 + a + 1)b \\ &\vdots \\ X_i &\equiv a^i X_0 + b\sum_{i=0}^{i-1} a^i \end{aligned}$$

Donc on pose $X'_i = X_i - b\sum_{i=0}^{i-1} a^i$ (si on retrouve l'un des X'_i, on retrouve automatiquement le X_i correspondant). Le but de la manoeuvre, c'est qu'alors $X'_0 = X_0$ et $X'_i = a^i X'_0$.

En fait, X' est la séquence d'états internes à laquelle on aurait affaire si $b = 0$. Comme $Y_i = \lfloor X_i/k \rfloor$, il semble logique de poser $Y'_i = \lfloor X'_i/k \rfloor$. On peut calculer les Y'_i avec l'observation :

$$Y'_i = \left\lfloor \frac{X_i}{k} - \frac{b}{k}\sum_{i=0}^{i-1} a^i \right\rfloor = Y_i - \left\lfloor \frac{b}{k}\sum_{i=0}^{i-1} a^i \right\rfloor \pm 1$$

Le ± 1 vient du fait que la partie entière peut introduire une « erreur d'arrondi ». Il n'empêche pas que $\mathbf{y}' = k(Y'_0, \ldots, Y'_{n-1})$ est proche d'un vecteur du réseau $\mathcal{L}$ qui révèle les X'_i.

2.2.5 (*) Cas des paramètres inconnus

Antoine Joux (né en 1967) et Jacques Stern (né en 1949) ont montré en 1998 [7] que même si on ne connaît pas a, b et m, on peut tout retrouver la plupart du temps. Les techniques utilisées sont plus sophistiquées... mais font aussi appel aux réseaux euclidiens. En deux mots, il s'agit de produire une liste de polynômes P_i à coefficients entiers tels que $P_i(a) \equiv 0 \bmod m$. On les obtient en cherchant de « petites relations linéaires » entre les Y_i avec LLL.

Une fois qu'on les a, on peut en faire des combinaisons linéaires pour éliminer toutes les puissances de X, et on se retrouve avec un (petit) multiple de m.

2.3 Attaque de Wiener sur RSA avec petit exposant secret

Dans le chiffrement ou la signature RSA, il faut produire une paire de clefs. Dans ce processus, on choisit généralement une valeur standard de l'exposant public e, typiquement $e = 3$ ou $e = 2^{16} + 1$. Ensuite, on calcule l'exposant secret d en résolvant la congruence $ed \equiv 1 \bmod \phi(N)$.

Du coup, d a typiquement la même taille que $\phi(N)$, or ce dernier est de la même taille que N. Il y a donc une asymétrique entre les deux exposants : e est petit et d est grand, donc le chiffrement/vérification est plus rapide que le déchiffrement/signature.

Pour éviter ceci, on pourrait faire l'inverse : choisir d aléatoirement (suffisamment grand pour qu'une recherche exhaustive soit impossible, disons 128 bits), puis calculer e en résolvant l'équation. Cela revient à dire que d est « petit », c'est-à-dire qu'il a été choisi aléatoirement dans l'ensemble $\{1, \ldots, N^\alpha\}$.

En 1989, Michael J. Wiener [16] a montré que ceci est en fait une assez mauvaise idée, car alors on peut trouver la clef secrète instantanément, si $\alpha \leq 1/4$! Son attaque est généralement présentée via la méthode des fractions continues. Ici, on adopte un autre point de vue, basé sur les réseaux euclidiens.

En tant qu'adversaire, on connaît la clef publique (N, e) ainsi que la borne α. On cherche à retrouver d. On sait que $ed \equiv 1 \bmod \phi$, donc il existe un entier k tel que $ed = 1 + k\phi(N)$. De plus, on a forcément $k < d$ [1] Ensuite, $\phi(N) = (p-1)(q-1) = N - (p+q) + 1$. Comme $p, q \approx N^{1/2}$, alors $\phi(N) \approx N$.

1. En effet, $k = (ed-1)/\phi(N)$, donc $k < ed/\phi(N)$; le résultat annoncé découle de ce que $e < \phi(N)$.

On considère le réseau $\mathcal{L}$ engendré par la matrice :

$$B = \begin{pmatrix} \sqrt{N} & e \\ 0 & N \end{pmatrix}$$

Il contient le vecteur $\mathbf{x} = (d, -k)B = (d\sqrt{N}, ed - kN) = (d\sqrt{N}, k(p+q-1))$, qui est de norme $\leq \sqrt{N^{\alpha+1/2}}$.

Le volume du réseau est facile à calculer car B est triangulaire, et donc $\mathrm{Vol}(\mathcal{L}) = N^{3/2}$. Par conséquent, l'heuristique gaussienne suggère que la taille du plus court vecteur « accidentel » de $\mathcal{L}$ devrait être de l'ordre de $N^{3/4}$.

Par conséquent, si $\mathbf{x}$ est sensiblement plus court que ceci, alors on peut raisonnablement faire le pari que $\mathbf{x}$ est bien le plus court vecteur de $\mathcal{L}$. Comme on est en dimension deux, on est sûr que l'algorithme LLL va bien révéler le plus court vecteur.

Si $\sqrt{N^{\alpha+1/2}} \ll N^{3/4}$, donc si $\alpha < 1/4$, alors on peut donc certainement retrouver d en temps très faible avec la procédure suivante

1. Assembler la matrice B ci-dessus.
2. Réduire B avec LLL.
3. Supposer que le premier vecteur de la matrice LLL-réduite est $(d\sqrt{N}, de - kN)$.
4. Extraire d de la première coordonnée de ce vecteur.

2.3.1 (*) D'où sort le réseau magique ?

Par définition de e et d, on a : $ed = 1 + k(N - p - q + 1)$. Ceci implique en particulier que les valeurs d, k secrètes satisfont le système linéaire :

$$|ed - kN| \leq N^{\alpha+1/2}, \qquad \text{avec } |k|, |d| \leq N^{\alpha}$$

(ce ne sont pas les seules solutions a priori, mais regarder ça est déjà un début).

Il s'agit d'un problème de « programmation linéaire entière ». C'est NP-dur en général, mais si le nombre de variable est fixé, alors c'est polynomial en la taille des coefficients [2].

Autrement dit, il faut trouver une combinaison linéaire entière de e et N de petite valeur. Pour cela, il y a une solution simple : l'algorithme d'Euclide étendu. Il produit une relation de Bezout, c'est-à-dire qu'il calcule deux entiers u, v tels que $ue + vN = 1$ (en effet, e est certainement premier avec N, sinon on factorise N en calculant son PGCD avec e...). Le problème c'est que les coefficients u et v peuvent être plus grands que N^{α}. Ils risquent d'être de taille $\approx N$.

En fait, il nous faudrait une relation linéaire à coefficients entiers entre e et N dont la valeur *et* les coefficients soient petits. Ceci est typiquement un problème de réseaux euclidiens !

Par exemple, on pourrait regarder le réseau engendré par :

$$B = \begin{pmatrix} 1 & 0 & e \\ 0 & 1 & N \end{pmatrix}$$

Un vecteur court dans ce réseau donne ce qu'on cherche. Si $(u, v)B = \mathbf{y}$ alors $\|\mathbf{y}\|^2 = u^2 + v^2 + (eu + vN)^2$. Par conséquent, $|u|, |v|$ et $|eu + vN|$ sont tous les trois plus petits que la norme de $\mathbf{y}$. Donc, si $\mathbf{y}$ est « court », alors on trouve une petite combinaison linéaire de e et N à petits coefficients.

Le théorème de Minkowski, associé au calcul du volume du réseau engendré par B, permettrait de donner une borne supérieure sur la taille de $\mathbf{y}$. C'est un peu casse-pied car le réseau n'est pas de rang plein.

On trouve en fait que le le volume est $\sqrt{N^2 + e^2 + 1} \approx N$. Donc le théorème de Minkowski nous dit qu'il y a un vecteur de longueur $\approx \sqrt{(N)}$ dans le réseau, ce qui implique une solution avec $|u|, |v| \approx \sqrt{N}$ et $|eu - vN| \approx \sqrt{N}$.

Le problème, c'est que ceci va fournir une relation linéaire entre e et N qui sera « équilibrée » : la taille des coefficients sera la même que la taille du résultat. Or, dans notre cas, les coefficients doivent être de taille N^{α}, mais le résultat peut être plus grand, de taille N^{α}. Pour refléter ceci, on modifie un peu le réseau :

$$B = \begin{pmatrix} \sqrt{N} & 0 & e \\ 0 & \sqrt{N} & N \end{pmatrix}$$

2. En fait, c'est pour démontrer *cela* que LLL a été inventé

Cette fois, si $(u, v)B = \mathbf{y}$ alors $\|\mathbf{y}\|^2 = u^2N + v^2N + (ue + vN)^2$ — on voit que le « poids » de u et v dans la norme a augmenté. Et alors on a que $|u, v| \leq \|\mathbf{y}\|/N^{1/2}$ tandis que $|ue + vN| \leq \|\mathbf{y}\|$, donc les coefficients doivent être $N^{1/2}$ fois plus petits que le résultat, ce qui est bien ce qu'on cherche.

En fait, ces deux réseaux sont de dimension deux mais les points du réseau sont à coordonnées dans $\mathbb{R}^3$ — en fait ils sont enfermés dans un plan d'équation $ex + yN - z = 0$.

Il serait possible de faire un changement de base et de toute exprimer en coordonnées à l'intérieur de ce plan [3]... Mais c'est plus beaucoup plus simple de larguer la coordonnée y, en projetant sur le plan xz ! C'est ainsi qu'on aboutit au réseau :

$$B = \begin{pmatrix} \sqrt{N} & e \\ & N \end{pmatrix}$$

Cela revient un peu à dire qu'on fait comme si on ne cherchait pas à contrôler la taille de k mais qu'on imposait seulement une limite à celle de d. En fait, comme e et N ont sensiblement la même taille, la valeur de k est contrainte à avoir à eu près la même taille que celle de d par le fait que $|ed - kN|$ doit être petit aussi. Donc au final, on ne perd rien.

2.4 Cryptanalyse d'une fonction à sens unique « post-quantique »

Le problème Subset Sum est un problème NP-complet classique [4]. Étant donné k entiers $a_1, \ldots, a_k$ et un entier « cible » b, il s'agit de décider s'il existe des coefficients $x_i \in \{0, 1\}$ tels que $v = \sum x_i a_i$ — autrement dit, la cible b est-elle la somme d'un sous-ensemble des a_i.

La version décisionnelle (qui est NP-dure) consiste à déterminer si les x_i existent ; la version calculatoire consiste à les trouver. Aucun algorithme capable de s'exécuter en temps polynomial sur des ordinateurs quantiques n'est connu à ce jour. Par conséquent, il est tentant de faire de la cryptographie en se basant sur la difficulté supposée de ce problème.

Par exemple, on peut construire la famille de fonctions suivante. La « clef » est la donnée de n entiers $a_1, \ldots, a_n$ assez grands, dont la somme s'écrit sur m bits (choisir une clef est facile : il suffit de tirer les a_i aléatoirement). La fonction s'évalue tout simplement :

$$\begin{array}{rccc} f : & \{0,1\}^n & \longrightarrow & \{0,1\}^m \\ & (x_0, \ldots, x_n) & \mapsto & \sum x_i a_i \end{array}$$

L'idée générale, c'est que cette fonction est à *sens unique* tant que le problème Subset Sum est difficile (en effet trouver une préimage est exactement la version calculatoire du problème). On verra qu'on peut construire presque toute la cryptographie symétrique à partir d'une simple fonction à sens unique — en particulier, on peut faire du chiffrement à clef secrète.

Il se trouve que dans ce cas-précis, trouver une collision dans la fonction est également NP-dur (comme l'a montré Peter van Emde Boas en 1981 ; c'était l'un des ingrédients pour montrer que CVP est NP-dur). Donc elle peut également faire office de fonction de hachage.

La sécurité de cette fonction à sens unique dépend de manière critique de la *densité* du sac-à-dos, qui est définie par :

$$d = \frac{n}{\max \log_2 a_i} \leq \frac{n}{m}.$$

Si $d \gg 1$ (la sortie est plus petite que l'entrée), alors des techniques simples de programmation dynamique peuvent trouver une préimage. Mais si $d < 1/2$ (la sortie est $2\times$ plus grosse que l'entrée), alors des techniques de réseau peuvent inverser la fonction efficacement.

Il semble qu'avec $d \approx 1$, la fonction soit vraiment difficile à inverser.

3. Les coordonnées ne seraient alors pas entières, et probablement même pas rationnelles...
4. Dans le contexte de la cryptographie, on l'appelle parfois le problème du sac-à-dos (Knapsack).

2.4.1 Attaque avec des réseaux en faible densité

Supposons qu'une clef ait été tirée aléatoirement (et soit connue). On dispose de $b = f(x)$, et on cherche x. Pour cela, on considère le réseau $\mathcal{L}$ (de dimension n) engendré par les lignes de la matrice

$$B = \begin{pmatrix} a_1 & 1 & & \\ a_2 & & 1 & \\ \vdots & & & \ddots \\ a_n & & & & 1 \end{pmatrix},$$

Le vecteur $\mathbf{x} = (x_1, \ldots, x_n)B = (b, x_1, \ldots, x_{256})$ appartient au réseau, et il est « proche » de $\mathbf{y} = (b, 0, \ldots, 0)$. Plus précisément, la distance entre les deux est inférieure à $\sqrt{n}$.

Le réseau n'est pas de rang plein (la matrice a $n+1$ colonnes et n lignes), mais il est facile de démontrer par récurrence que

$$\mathrm{Vol}(\mathcal{L}) = \sqrt{\det\left(B \times B^t\right)} = \sqrt{1 + \sum_{i=1}^{n} {a_i}^2} \approx 2^m \sqrt{n}.$$

Par conséquent, avec l'heuristique gaussienne, on s'attend à ce qu'une distance typique dans le réseau soit $\approx \mathrm{Vol}(\mathcal{L})^{1/n} \approx \sqrt{n} 2^{m/n}$. Ceci revient à dire que si on cherche une combinaison linéaire des a_i qui s'annule, alors il faut que ses coefficients fassent $2^{m/n}$ bits. On voit que le vecteur « cible » $\mathbf{y}$ est (heuristiquement) le plus proche de $\mathbf{x}$ lorsque $m \gg n$.


Lagarias et Odlyzko ont démontré en 1983 que lorsque la densité du sac-à-dos est inférieure à 0.645, alors résoudre cette instance de CVP avec la méthode du plongement permet bien de résoudre le problème en trouvant le plus court vecteur du réseau [10]. Ceci est plus précis que le raisonnement « à la pelleteuse » décrit ici, qui en est adapté.

Par contre, le trouver n'est pas facile : il faut résoudre une instance de CVP en assez grande dimension.. or le problème est NP-dur lui aussi.

Mais on peut s'en tirer avec un algorithme polynomial qui résout CVP avec un facteur d'approximation exponentiel, tels que ceux de la section 1.4. L'idée générale est la suivante (cf. figure 2.2) : si d désigne la distance entre la « cible » et le point du réseau le plus proche, alors un algorithme qui résout CVP avec facteur approximation γ va renvoyer un point du réseau à distance $\leq \gamma \cdot d$ de la cible. Si jamais le point du réseau le plus proche de la cible est *le seul* point du réseau qui satisfait cette condition, c'est forcément celui-là qu'on va récupérer.

Autrement dit, plus la cible est proche du réseau, plus un mauvais algorithme d'approximation suffit. Dans notre cas, la cible est à distance $\sqrt{n}$ tandis que la distance entre deux points du réseau est de l'ordre de $\sqrt{m/n} 2^{m/n}$. Donc lorsque m augmente, la situation qu'on recherche (décrite par la figure 2.2) se créée effectivement.

Si on utilise un algorithme polynomial qui approche CVP avec un facteur $2^{n/2}$ (comme la méthode du plongement ou la méthode de l'hyperplan le plus proche), alors on va pouvoir résoudre exactement l'instance de CVP lorsque $2^{n/2}\sqrt{n} < \sqrt{n} 2^{m/n}$. Ceci se simplifie en $n^2/2 < m$.

Autrement dit, lorsque $m > n^2/2$, alors on peut inverser la fonction en temps polynomial en résolvant une instance de approx-CVP. Pour une entrée de 128 bits, ça donnerait une sortie d'au moins 8192 bits pour que l'attaque fonctionne. En pratique, le raisonnement ci-dessus est pessimiste, et l'attaque marche bien mieux que cela : dès $m = 600$ bits, ça fonctionne très bien.

2.5 Attaque contre le chiffrement partiellement homomorphe sur les entiers

En 2010, van Dijk, Gentry, Halevi et Vaikuntanathan ont proposé un mécanisme de chiffrement (à clef secrète) simple et étonnant [15].

- La clef est simplement un nombre aléatoire (impair) $-2^{n-1} \leq x < 2^n$.
- Pour chiffrer un bit $m \in \{0, 1\}$ avec la clef x, choisir au hasard k, r tels que $|k| < x^\alpha$ et $|r| < x^\beta$ puis calculer $c \leftarrow m + 2r + kx$.
- Pour déchiffrer, faire $m \leftarrow (c \bmod x) \bmod 2$.

Les auteurs suggèrent de choisir $r \approx 2^{\sqrt{n}}$ et $k \approx 2^{n^3}$ pour que le système soit sûr. Intuitivement, le terme en $2r$ est du « bruit », tandis que le multiple de x joue le rôle d'un masque jetable (et comme x est impair, ça

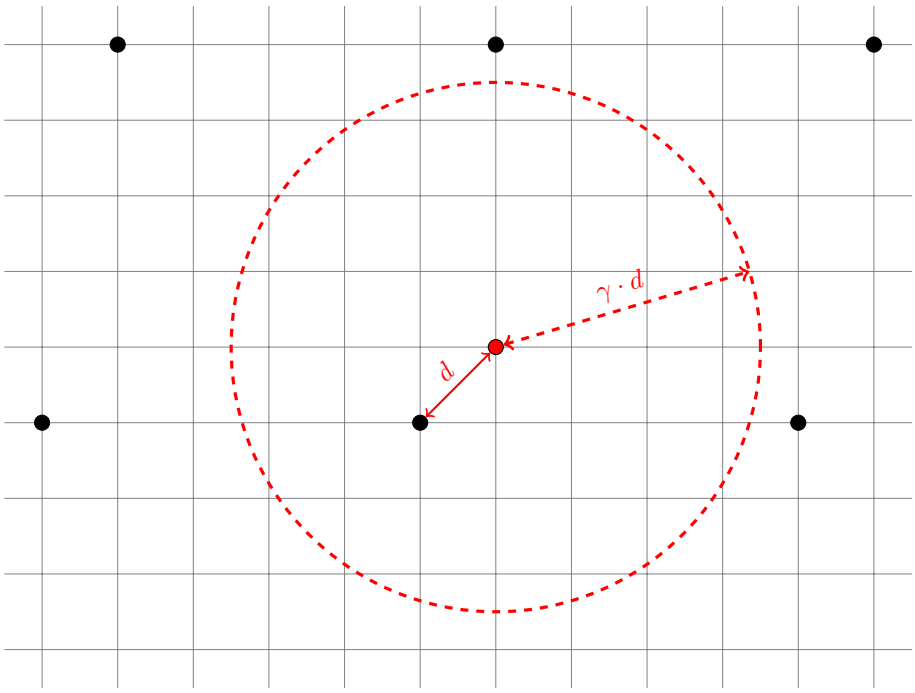

FIGURE 2.2 – Résolution exacte de CVP avec un algorithme d'approximation, même très mauvais, dans le cas où la « cible » est très proche d'un point du réseau. Pour que ça renvoie le plus proche point du réseau, il suffit qu'il n'y en ait aucun autre à distance $\gamma \cdot d$ de la cible, où γ est le facteur d'approximation de l'algorithme.

masque correctement la parité de m...). Tant que le bruit est sensiblement plus petit que x, le déchiffrement va fonctionner correctement.

Un des intérêts du système, c'est que si c_1, c_2 sont les chiffrés de m_1, m_2, alors :

$$\begin{aligned} c_1 + c_2 &= (m_1 + m_2) + 2(r_1 + r_2) + x(k_1 + k_2) \\ c_1 \times c_2 &= (m_1 + 2r_1 + xk_1)(m_2 + 2r_2 + xk_2) \\ &= m_1 m_2 + 2(m_1 r_2 + r_1 m_2 + 2r_1 r_2) + x(\dots) \end{aligned}$$

Autrement dit, $c_1 + c_2$ est le chiffré de $m_1 + m_2$ (modulo 2), tandis que $c_1 c_2$ est le chiffré de $m_1 m_2$! Mais attention, à chaque opération sur les chiffrés, le « bruit » augmente, et s'il dépasse $x/2$, alors le déchiffrement ne fonctionnera plus. L'addition ajoute un bit de bruit, mais la multiplication double le nombre de bits du bruit. On ne peut donc faire qu'un nombre limité d'opérations sur les chiffrés, mais ce n'est déjà pas si mal, vu la simplicité du mécanisme.

2.5.1 Attaque avec le problème du PGCD approché

Si nous donne des multiples $c_1 = k_1 x, c_2 = k_2 x, \dots$ d'un nombre x secret, on va probablement être capable de retrouver x en calculant le PGCD des c_i. Ceci va bien marcher même si x et les k_i sont très grands.

Mais ceci ne marcherait plus du tout si on nous donnait des multiples « bruités » de x, c.a.d. des nombres du type $c'_1 = k_1 x + r_1, c'_2 = k_2 x + r_2, \dots$, où les termes de « bruit » r_i sont petits. Reconstituer x à partir de ces multiples bruités constitue le problème Approximate GCD (PGCD approché).

En fait, casser le mécanisme de chiffrement décrit ci-dessus c'est exactement la même chose. Supposons qu'un adversaire puisse récupérer des chiffrés $c_i = m_i + 2r_i + k_i x$. Il veut récupérer la clef secrète. En fait, ce sont des multiples de x « bruités » par $r'_i = m_i + 2r_i$.

À quel point le PGCD approché est-il facile à résoudre ? Plusieurs algorithmes sont possibles, mais on en présente un relativement simple ici, dû à Lagarias en 1985 [9].

Comme le bruit est petit, on a :

$$\frac{c_i}{c_0} = \frac{k_i x + r'_i}{k_0 x + r'_0} \approx \frac{k_i}{k_0}.$$

Par conséquent, $k_0 c_i \approx k_i c_0$ et donc la différence entre ces deux nombres doit être « petite ». En fait on a :

$$k_0 c_i - k_i c_0 = k_0 r'_i - k_i r'_0$$

Et donc $|k_0 c_i - k_i c_0| \leq 2^{n(\alpha+\beta)+1}$ tandis que $|c_i| \leq 2^{n(\alpha+1)}$.

Considérons le réseau $\mathcal{L}$ engendré par les lignes de la matrice :

$$M = \begin{pmatrix} 2^{\beta n+1} & c_1 & c_2 & \dots & c_{\ell-1} \\ 0 & -c_0 & 0 & \dots & 0 \\ 0 & 0 & -c_0 & \dots & 0 \\ & & & \ddots & \\ 0 & 0 & 0 & \dots & -c_0 \end{pmatrix}$$

Le réseau contient le vecteur $\mathbf{x} = (k_0, k_1, \dots, k_{\ell-1})M = (k_0 2^{\beta n+1}, k_0 r'_1 - k_1 r'_0, \dots, k_0 r'_{\ell-1} - k_{\ell-1} r'_0)$. Chaque coordonnée de $\mathbf{x}$ est plus petite [5] (en valeur absolue) que $2^{n(\alpha+\beta)+1}$, donc la norme de $\mathbf{x}$ est majorée par $\sqrt{\ell} 2^{n(\alpha+\beta)+1}$. Retrouver ce vecteur $\mathbf{x}$ casse le schéma de chiffrement, car il révèle k_0, ce qui permet de retrouver x (en calculant la division euclidienne de c_0 par k_0).

Comme la matrice est triangulaire, le volume du réseau est simplement le produit des entrées sur la diagonale :

$$\mathrm{Vol}(\mathcal{L}) = 2^{\beta n+1} {c_0}^{\ell-1} \approx 2^{1+\beta n+(\ell-1)(\alpha+1)n}$$

Par conséquent, avec l'heuristique gaussienne, on s'attend à ce que $\mathbf{x}$ soit le plus court vecteur du réseau lorsque

$$2\sqrt{\ell} \times 2^{n(\alpha+\beta)} < \sqrt{\ell} \left(2^{\beta n+(\ell-1)(\alpha+1)n}\right)^{1/\ell}$$

Ceci se simplifie en :

$$\frac{\alpha}{1-\beta} + 1 < \ell$$

Autrement dit, avec suffisamment de chiffrés, on peut retrouver la clef secrète en résolvant une instance de CVP. Le nombre des chiffrés nécessaires ne dépend pas de la taille de la clef, mais des tailles respectives du « bruit » et des multiples de x utilisés comme masques. Plus le bruit est proche de x, plus il faut de chiffrés. Plus les « multiples » de la clef secrète sont grands, plus il faut de chiffrés.

Par exemple, avec x de 128 bits, un bruit limité à 32 bits et des multiples de 2048 bits, il faudrait obtenir 22 chiffrés puis résoudre SVP en dimension 22 pour tout casser, ce qui est pratiquement facile, même avec les algorithmes exacts pour SVP.

2.5.2 (*) Attaque avec LLL

Si on se contente d'un algorithme qui résout SVP approximativement, comme LLL, avec facteur d'approximation $2^{n/2}$, que peut-on casser en temps strictement polynomial ? On a la certitude d'obtenir un vecteur au pire $2^{\ell/2}$ fois trop long.

Donc, on devrait bien récupérer $\mathbf{x}$ à condition que

$$2\sqrt{\ell} \times 2^{n(\alpha+\beta)} \times 2^{\ell/2} < \sqrt{\ell} \left(2^{\beta n+(\ell-1)(\alpha+1)n}\right)^{1/\ell}$$

Cette fois, ceci se simplifie en :

$$(\alpha + 1 - \beta) - \ell(1-\beta) + \frac{1}{2n}\ell^2 < 0$$

Si ℓ est trop grand, alors l'inéquation sera fausse. Pour trouver les valeurs admissibles de ℓ, calculons le discriminant :

$$\Delta = (1-\beta)^2 - 2\frac{\alpha+1-\beta}{n}$$

Si n est suffisamment grand ($n > 2(\alpha+1-\beta)/(1-\beta)^2$), alors le discriminant est positif, et l'inéquation est satisfaite pour ℓ entre les deux racines :

$$\ell_\pm = n\left((1-\beta) \pm \sqrt{(1-\beta)^2 - 2\frac{\alpha+1-\beta}{n}}\right)$$

5. C'est pour ça qu'on met $2^{\beta n+1}$ et pas juste 1 en haut à gauche.

Intéressons-nous à la la petite racine (c'est la plus petite valeur de ℓ qui va fonctionner) :

$$\ell_- = n(1-\beta)\left(1-\sqrt{1-2\frac{\alpha+1-\beta}{n(1-\beta)^2}}\right)$$

Utilisons le développement limité $1-\sqrt{1-2x} = x + \mathcal{O}\left(x^2\right)$ (au voisinage de zéro). On obtient :

$$\ell_- = \frac{\alpha}{1-\beta} + 1 + \mathcal{O}\left(n^{-1}\right)$$

C'est-à-dire presque la même chose qu'avant ! Si la clef est suffisamment grande, alors on ne perd quasiment rien en utilisant LLL plutôt qu'un algorithme exact pour résoudre SVP.

Mais l'attaque ne marche maintenant que si :

$$n > \frac{2}{1-\beta} + \frac{2\alpha}{(1-\beta)^2},$$

donc il est possible de choisir des paramètres (n, α, β) qui empêchent l'attaque et rendent le schéma plus sûr.

2.6 Collisions dans la fonction de Hachage d'Ajtai

Une idée relativement simple pour concevoir une famille de fonctions de hachages résistantes aux collision est la suivante. Cette idée est dûe à Miklós Ajtai (né en 1946) en 1996 [1].

Choisir un nombre premier q ainsi qu'une borne $t < (q-1)/2$. Tirer uniformément au hasard une matrice A de taille $n \times m$ (dans toute la suite on suppose que $n > m$ et que A est de rang m). La fonction prend en entrée un vecteur $\mathbf{x}$ de n entiers compris dans $\{-t, \ldots, 0, \ldots, t\}$ et renvoie :

$$f(\mathbf{x}) = \mathbf{x}A \bmod q.$$

C'est un simple produit matrice-vecteur, et par exemple, avec $q = 257, t = 8, n = 128$, et $m = 32$, on a une fonction qui compresse 512 bits en ≈ 256. De manière générale, pour que la sortie soit deux fois plus petite que l'entrée, il faut que $n \log_2 t \approx 2m \log q$. Si on fixe la taille de la sortie (à 256 bits, disons), alors ça donne $m \log_2 q \approx 256$.

Le paramètre t influence fortement la sécurité de la fonction (les petites valeurs sont plus sûres, et moins efficaces). En effet, pour inverser la fonction, on pourrait croire il suffit de résoudre $\mathbf{x}A = \mathbf{y} \bmod q$, ce qui est facile. Mais le problème, c'est qu'il y a peu de chances que les coordonnées de $\mathbf{x}$ soient toutes inférieures ou égales à t en valeur absolue.

De même, pour trouver une collision, il suffit presque de trouver une préimage de zéro : si $f(\mathbf{x}) = 0$, alors pour tout $\mathbf{y}$, on aura $f(\mathbf{y}) = f(\mathbf{x} + \mathbf{y})$. Mais il faut que $\mathbf{x} + \mathbf{y}$ respecte la contrainte imposée par t, donc que les coefficients de $\mathbf{x}$ soient plus petits que t en valeur absolue.

Trouver de « petites » solutions d'équations linéaires, c'est un boulot pour LLL ! Mais cette fois, il y a un *twist* : la réduction modulo q.

On calcule une base du noyau à gauche de A modulo q : c'est une matrice K de taille $(n-m) \times n$ telle que $K \cdot A = 0 \bmod q$, et K est de rang $n-m$. Par conséquent, on peut supposer que K est mise sous forme échelonnée réduite, donc qu'on a :

$$K = \begin{pmatrix} I & K' \end{pmatrix}$$

On considère alors le réseau de dimension n engendré par les lignes de la matrice par blocs :

$$B = \begin{pmatrix} I & K' \\ & qI \end{pmatrix},$$

Si $\mathbf{x}$ est un vecteur d'entiers tel que $f(\mathbf{x}) = 0 \bmod q$, alors $\mathbf{x}$ appartient à ce réseau, et vice-versa. Le problème consiste à trouver de petits vecteurs là-dedans. Le volume de ce réseau est q^m, donc un algorithme qui résout HSVP (comme LLL ou BKZ) avec facteur d'approximation γ^n va renvoyer un vecteur de norme inférieure à $\gamma^n q^{m/n}$.

L'entrée de la fonction de hachage est de taille $\approx n \log_2 t$ tandis que la sortie est de taille $\approx m \log_2 q$. Définissons donc le « ratio de compression » $\lambda = \frac{m \log_2 q}{n \log_2 t} > 1$. On a donc $\lambda \log_2 t = \frac{m}{n} \log_2 q$, et on peut donc dire que

l'algorithme pour HSVP va renvoyer un vecteur de norme inférieure à $\gamma^n t^\lambda$. Si ceci est inférieur à t, alors le vecteur est forcément « valide ». Par conséquent, la fonction est cassée par l'algorithme qui résout HSVP lorsque $\gamma^n \leq t^{1-\lambda}$.

Pour le cas des paramètres donnés en entrée, on a $\lambda = 1/2, n = 128$. Sur le papier, LLL garantit $\gamma = (4/3+\varepsilon)^{1/4}$. Si on imagine que $\varepsilon = 0$, alors ceci devient $(4/3)^{64} \leq t$, et la fonction n'est pas cassée.

Mais dans la réalité, LLL se comporte mieux prévu ! Il atteint $\gamma = 1.0159$ sur ce type de problème, et produit des vecteurs qui cassent $t = 16$ systématiquement. Et BKZ-20 casse $t = 4$...

Ceci dit, avec $t = 1$, la fonction doit être relativement résistante aux collisions...

Bibliographie

[1] Miklós Ajtai. Generating hard instances of lattice problems (extended abstract). In Gary L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 99–108. ACM, 1996.

[2] Miklós Ajtai. The shortest vector problem in L_2 is NP-hard for randomized reductions (extended abstract). In Jeffrey Scott Vitter, editor, *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, pages 10–19. ACM, 1998.

[3] László Babai. On lovász' lattice reduction and the nearest lattice point problem. *Comb.*, 6(1) :1–13, 1986.

[4] Irit Dinur, Guy Kindler, Ran Raz, and Shmuel Safra. Approximating CVP to within almost-polynomial factors is np-hard. *Comb.*, 23(2) :205–243, 2003.

[5] Alan M. Frieze, Johan Håstad, Ravi Kannan, J. C. Lagarias, and Adi Shamir. Reconstructing truncated integer variables satisfying linear congruences. *SIAM J. Comput.*, 17(2) :262–280, 1988.

[6] Nicolas Gama and Phong Q. Nguyen. Predicting lattice reduction. In Nigel P. Smart, editor, *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings*, volume 4965 of *Lecture Notes in Computer Science*, pages 31–51. Springer, 2008.

[7] Antoine Joux and Jacques Stern. Lattice reduction : A toolbox for the cryptanalyst. *J. Cryptol.*, 11(3) :161–185, 1998.

[8] D. Knuth. Deciphering a linear congruential encryption. *IEEE Transactions on Information Theory*, 31(1) :49–52, 1985.

[9] J. C. Lagarias. The computational complexity of simultaneous diophantine approximation problems. *SIAM J. Comput.*, 14(1) :196–209, 1985.

[10] J. C. Lagarias and Andrew M. Odlyzko. Solving low-density subset sum problems. *J. ACM*, 32(1) :229–246, 1985.

[11] L. Lovasz. *An Algorithmic Theory of Numbers, Graphs, and Convexity.* CBMS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics, 1986.

[12] Phong Q Nguyen. Public-key cryptanalysis. *Recent Trends in Cryptography, Contemp. Math*, 477 :67–120, 2009.

[13] Phong Q. Nguyen. Hermite's constant and lattice algorithms. In Phong Q. Nguyen and Brigitte Vallée, editors, *The LLL Algorithm - Survey and Applications*, Information Security and Cryptography, pages 19–69. Springer, 2010.

[14] Claus-Peter Schnorr and M. Euchner. Lattice basis reduction : Improved practical algorithms and solving subset sum problems. *Math. Program.*, 66 :181–199, 1994.

[15] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In Henri Gilbert, editor, *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Monaco / French Riviera, May 30 - June 3, 2010. Proceedings*, volume 6110 of *Lecture Notes in Computer Science*, pages 24–43. Springer, 2010.

[16] Michael J. Wiener. Cryptanalysis of short RSA secret exponents. *IEEE Trans. Inf. Theory*, 36(3) :553–558, 1990.