Chapter 1
Floating-Point LLL: Theoretical and Practical Aspects

Damien Stehlé

Abstract The text-book LLL algorithm can be sped up considerably by replacing the underlying rational arithmetic used for the Gram-Schmidt orthogonalisation by floating-point approximations. We review how this modification has been and is currently implemented, both in theory and in practice. Using floating-point approximations seems to be natural for LLL even from the theoretical point of view: it is the key to reach a bit-complexity which is quadratic with respect to the bit-length of the input vectors entries, without fast integer multiplication. The latter bit-complexity strengthens the connection between LLL and Euclid's gcd algorithm. On the practical side, the LLL implementer may weaken the provable variants in order to further improve their efficiency: we emphasise on these techniques. We also consider the practical behaviour of the floating-point LLL algorithms, in particular their output distribution, their running-time and their numerical behaviour. After 25 years of implementation, many questions motivated by the practical side of LLL remain open.

1.1 Introduction

The LLL lattice reduction algorithm was published in 1982 [35], and was immediately heartily welcome in the algorithmic community because of the numerous barriers it broke. Thanks to promising applications, the algorithm was promptly implemented. For example, as soon as the Summer 1982, Erich Kaltofen implemented LLL in the Macsyma computer algebra software to factor polynomials. This implementation followed very closely the original description of the algorithm. Andrew Odlyzko tried to use it to attack knapsack-

CNRS/Universities of Macquarie, Sydney and Lyon/INRIA/ÉNS Lyon
Dept of Mathematics and Statistics, University of Sydney, NSW 2008, Australia.
e-mail: `damien.stehle@gmail.com` – `http://perso.ens-lyon.fr/damien.stehle`

based cryptosystems, but the costly rational arithmetic and limited power of computers available at that time limited him to working with lattices of small dimensions. This led him to replace the rationals by floating-point approximations, which he was one of the very first persons to do, at the end of 1982. This enabled him to reduce lattices of dimensions higher than 20.

The reachable dimension was then significantly increased by the use of a Cray-1, which helped solving low-density knapsacks [34] and disproving the famous Mertens conjecture [46]. The use of the floating-point arithmetic in those implementations was heuristic, as no precautions were taken (and no theoretical methods were available) to assure correctness of the computation when dealing with approximations to the numbers used within the LLL algorithm.

In the 1990's, it progressively became important to reduce lattice bases of higher dimensions. Lattice reduction gained tremendous popularity in the field of public-key cryptography thanks to the birth of lattice-based cryptosystems [5, 21, 27] and to the methods of Coppersmith to find small roots of polynomials [14, 15, 16], a very powerful tool in public-key cryptanalysis (see the survey [38] contained in this book). Lattice-based cryptography involves lattice bases of huge dimensions (502 for the first NTRU challenge and several hundreds in the case of the GGH challenges), and in Coppersmith's method lattices of dimensions between 40 and 100 are quite frequent [10, 39]. Nowadays, very large dimensional lattices are also being reduced in computational group theory or for factoring univariate polynomials.

All competitive implementations of the LLL algorithm rely on floating-point arithmetic. Sometimes, however, one wants to be sure of the quality of the output, to obtain mathematical results, e.g., to prove there is no small linear integer relation between given numbers. More often, one wants to be sure that the started execution will terminate. This motivates the study of the reliability of floating-point operations within LLL, which is the line of research we are going to survey below. In 1988, Schnorr described the first provable floating-point variant of the LLL algorithm [50]. This was followed by a series of heuristic and provable variants [53, 33, 43, 51]. Apart from these references, most of the practical issues we are to describe are derived from the codes of today's fastest floating-point LLL routines: LiDIa's [1], Magma's [11], NTL's [59], as well as in `fplll-2.0` [12].

The practical behaviour of the LLL algorithm is often considered as mysterious. Though many natural questions concerning the average behaviour of LLL remain open in growing dimensions (see the survey [62]), a few important properties can be obtained by experimentation. It appears then that there is a general shape for bases output by LLL, that the running-time behaves predictively for some families of inputs and that there exists a generic numerical behaviour. We explain these phenomena and describe how some of the observed properties may be used to improve the code: for example, guessing accurately the causes of an undesirable behaviour that occurs during the

execution of a heuristic variant helps selecting another variant which is more likely to work.

Road-map of the survey. In Section 1.2, we give some necessary background on LLL and floating-point arithmetic. We then describe the provable floating-point algorithms in Section 1.3, as well as the heuristic practice-oriented ones in Section 1.4. In Section 1.5, we report observations on the practical behaviour of LLL, and finally, in Section 1.6, we draw a list of open problems and ongoing research topics related to floating-point arithmetic within the LLL algorithm.

Model of Computation. In the paper, we consider the usual bit-complexity model. For the integer and arbitrary precision floating-point arithmetics, unless stated otherwise, we restrict ourselves to naive algorithms, i.e., we do not use any fast multiplication algorithm [20]. This choice is motivated by two main reasons. Firstly, the integer arithmetic operations dominating the overall cost of the described floating-point LLL algorithms are multiplications of large integers by small integers (most of the time linear in the lattice dimension): using fast multiplication algorithms here is meaningless in practice since the lattice dimensions remain far below the efficiency threshold between naive and fast integer multiplications. Secondly, we probably do not know yet how to fully exploit fast multiplication algorithms in LLL-type algorithms: having a quadratic cost with naive integer arithmetic suggests that a quasi-linear cost with fast integer arithmetic may be reachable (see Section 1.6 for more details).

Other LLL-reduction algorithms. Some LLL-type algorithms have lower complexity upper bounds than the ones described below, with respect to the lattice dimension [55, 60, 32, 33, 51]. However, their complexity upper bounds are worse than the ones below with respect to the bit-sizes of the input matrix entries. Improving the linear algebra cost and the arithmetic cost can be thought of as independent strategies to speed up lattice reduction algorithms. Ideally, one would like to be able to combine these improvements into one single algorithm. Improving the linear algebra cost of LLL is not the scope of the present survey, and for this topic we refer to [48].

Notation. During the survey, vectors will be denoted in bold. If $\mathbf{b}$ is an n-dimensional vector, we denote its i-th coordinate by $\mathbf{b}[i]$, for $i \leq n$. Its length $\sqrt{\sum_{i=1}^{n} \mathbf{b}[i]^2}$ is denoted by $\|\mathbf{b}\|$. If $\mathbf{b}_1$ and $\mathbf{b}_2$ are two n-dimensional vectors, their scalar product $\sum_{i=1}^{n} \mathbf{b}_1[i] \cdot \mathbf{b}_2[i]$ is denoted by $\langle \mathbf{b}_1, \mathbf{b}_2 \rangle$. If x is a real number, we define $\lfloor x \rceil$ as the closest integer to x (the even one if x is equally distant from two consecutive integers). We use bars to denote approximations: for example $\bar{\mu}_{i,j}$ is an approximation to $\mu_{i,j}$. By default, the function $\log$ will be the base-2 logarithm.

1.2 Background Definitions and Results

An introduction to the geometry of numbers can be found in [37]. The algorithmic aspects of lattices are described in the present book, in particular in the survey [48], and therefore we only give the definitions and results that are specific to the use of floating-point arithmetic within LLL. In particular, we briefly describe floating-point arithmetic. We refer to the first chapters of [26] and [40] for more details.

1.2.1 Floating-Point Arithmetic

Floating-point arithmetic is the most frequent way to simulate real numbers in a computer. Contrary to a common belief, floating-point numbers and arithmetic operations on floating-point numbers are rigorously specified, and mathematical proofs, most often in the shape of error analysis, can be built upon these specifications.

The most common floating-point numbers are the binary double precision floating-point numbers (doubles for short). They are formally defined in the IEEE-754 standard [2]. The following definition is incomplete with respect to the IEEE-754 standard, but will suffice for our needs.

Definition 1. A double consists of 53 bits of mantissa m which are interpreted as a number in $\{1, 1+2^{-52}, 1+2\cdot 2^{-52}, \ldots, 2-2^{-52}\}$; a bit of sign s; and 11 bits of exponent e which are interpreted as an integer in $[-1022, 1023]$. The real number represented that way is $(-1)^s \cdot m \cdot 2^e$.

If x is a real number, we define $\diamond(x)$ as the closest double to x, choosing the one with an even mantissa in case there are two possibilities. Other rounding modes are defined in the standard, but here we will only use the rounding to nearest. We will implicitly extend the notation $\diamond(\cdot)$ to extensions of the double precision. The IEEE-754 standard also dictates how arithmetic operations must be performed on doubles. If $op \in \{+, -, \times, \div\}$, the result of $(a\ op\ b)$ where a and b are doubles is the double corresponding to the rounding of the real number $(a\ op\ b)$, i.e., $\diamond(a\ op\ b)$. Similarly, the result of $\sqrt{a}$ is $\diamond(\sqrt{a})$.

Doubles are very convenient because they are widely available, they are normalised, they are extremely efficient since most often implemented at the processor level, and they suffice in many applications. They nevertheless have two major limitations: the exponent is limited (only 11 bits) and the precision is limited (only 53 bits).

A classical way to work around the exponent limitation is to batch an integer (most often a 32-bit integer suffices) to each double, in order to extend the exponent. For example, the pair (x, e) where x is a double and e is an integer could encode the number $x \cdot 2^e$. One must be careful that a given number may have several representations, because of the presence of two

exponents (the one of the double and the additional one), and it may thus prove useful to restrict the range of the double to $[1,2)$ or to any other binade. An implementation of such a double plus exponent arithmetic is the `dpe`[1] library written by Patrick Pélissier and Paul Zimmermann, which satisfies specifications similar to the IEEE-754 standard. We will use the term dpe to denote this extension of the standard doubles.

If for some application a larger precision is needed, one may use arbitrary precision real numbers. In this case, a number comes along with its precision, which may vary. It is usually implemented from an arbitrary precision integer package. An example is MPFR [47], which is based on GNU MP [23]. It is a smooth extension of the IEEE-754 standardised doubles. Another such implementation is the `RR`-class of Shoup's Number Theory Library [59], which can be based, at compilation time, either on GNU MP or on NTL's arbitrary precision integers. Arbitrary precision floating-point numbers are semantically very convenient, but one should try to limit their use in practice since they are significantly slower than the processor-based doubles: even if the precision is chosen to be 53 bits, the speed ratio for the basic arithmetic operations can be larger than 15.

1.2.2 Lattices

A lattice L is a discrete subgroup of some $\mathbb{R}^n$. Such an object can always be represented as the set of integer linear combinations of some vectors $\mathbf{b}_1,\ldots,\mathbf{b}_d \in \mathbb{R}^n$ with $d \leq n$. If these vectors are linearly independent, we say that they form a basis of the lattice L. A given lattice may have an infinity of bases, related to one another by unimodular transforms, i.e., by multiplying on the right the column expressions of the basis vectors by a square integral matrix of determinant ± 1. The cardinalities d of the bases of a given lattice match and are called the lattice dimension, whereas n is called the embedding dimension. Both are lattice invariants: they depend on the lattice but not on the chosen basis of the lattice. There are two other important lattice invariants: the volume $\mathrm{vol}(L) := \sqrt{\det(B^t \cdot B)}$ where B is the matrix whose columns are any basis of L, and the minimum $\lambda(L)$ which is the length of a shortest non-zero lattice vector.

Gram-Schmidt orthogonalisation. Let $\mathbf{b}_1,\ldots,\mathbf{b}_d$ be linearly independent vectors. Their *Gram-Schmidt orthogonalisation* (GSO for short) $\mathbf{b}_1^*,\ldots,\mathbf{b}_d^*$ is the orthogonal family defined recursively as follows: the vector $\mathbf{b}_i^*$ is the component of the vector $\mathbf{b}_i$ which is orthogonal to the linear span of the vectors $\mathbf{b}_1,\ldots,\mathbf{b}_{i-1}$. We have $\mathbf{b}_i^* = \mathbf{b}_i - \sum_{j=1}^{i-1} \mu_{i,j}\mathbf{b}_j^*$ where $\mu_{i,j} = \frac{\langle \mathbf{b}_i, \mathbf{b}_j^* \rangle}{\|\mathbf{b}_j^*\|^2}$. For $i \leq d$ we let $\mu_{i,i} = 1$. The quantity $\mu_{i,j}$ is the component of the vector $\mathbf{b}_i$ on the vector $\mathbf{b}_j^*$

[1] `http://www.loria.fr/~zimmerma/free/dpe-1.4.tar.gz`

when written as a linear combination of the $\mathbf{b}_k^*$'s. The Gram-Schmidt orthogonalisation is widely used in lattice reduction because a reduced basis is somehow close to being orthogonal, which can be rephrased conveniently in terms of the GSO coefficients: the $\|\mathbf{b}_i^*\|$'s must not decrease too fast, and the $\mu_{i,j}$'s must be relatively small. Another interesting property of the GSO is that the volume of the lattice L spanned by the $\mathbf{b}_i$'s satisfies $\mathrm{vol}(L) = \prod_{i \leq d} \|\mathbf{b}_i^*\|$.

Notice that the GSO family depends on the order of the vectors. Furthermore, if the $\mathbf{b}_i$'s are integer vectors, the $\mathbf{b}_i^*$'s and the $\mu_{i,j}$'s are rational numbers. We also define the variables $r_{i,j}$ for $i \geq j$ as follows: for any $i \in [1,d]$, we let $r_{i,i} = \|\mathbf{b}_i^*\|^2$, and for any $i \geq j$ we let $r_{i,j} = \mu_{i,j} r_{j,j} = \langle \mathbf{b}_i, \mathbf{b}_j^* \rangle$. We have the relation $r_{i,j} = \langle \mathbf{b}_i, \mathbf{b}_j \rangle - \sum_{k<j} r_{i,k}\mu_{j,k}$, for any $i \geq j$. In what follows, the *GSO family* denotes the $r_{i,j}$'s and $\mu_{i,j}$'s. Some information is redundant in rational arithmetic, but in the context of our floating-point calculations, it is useful to have all these variables.

QR and Cholesky factorisations. The GSO coefficients are closely related to the Q and R factors of the QR-factorisation of the basis matrix. Suppose that the linearly independent vectors $\mathbf{b}_1, \ldots, \mathbf{b}_d$ are given by the columns of an $n \times d$ matrix B. Then one can write $B = Q \cdot \begin{bmatrix} R \\ 0 \end{bmatrix}$ where Q is an $n \times n$ orthogonal matrix and R is a $d \times d$ upper triangular matrix with positive diagonal entries. The first d columns of Q and the matrix R are unique and one has the following relations with the GSO family:

- For $i \leq d$, the i-th column Q_i of the matrix Q is the vector $\frac{1}{\|\mathbf{b}_i^*\|}\mathbf{b}_i^*$.
- The diagonal coefficient $R_{i,i}$ is $\|\mathbf{b}_i^*\|$.
- If $i < j$, the coefficient $R_{i,j}$ is $\frac{\langle \mathbf{b}_j, \mathbf{b}_i^* \rangle}{\|\mathbf{b}_i^*\|} = \frac{r_{j,i}}{\sqrt{r_{i,i}}}$.

In the rest of the survey, in order to avoid any confusion between the matrices $(R_{i,j})_{i \geq j}$ and $(r_{i,j})_{i \leq j}$, we will only use the $r_{i,j}$'s.

The Cholesky factorisation applies to a symmetric definite positive matrix. If A is such a matrix, its Cholesky factorisation is $A = R^t \cdot R$, where R is upper triangular with positive diagonal entries. Suppose now that A is the Gram matrix $B^t \cdot B$ of a basis matrix B. Then the R-matrix of the Cholesky factorisation of A is exactly the R-factor of the QR-factorisation of B. The QR and Cholesky factorisations have been extensively studied in numerical analysis and we refer to [26] for a general overview.

Size-reduction. A basis $(\mathbf{b}_1, \ldots, \mathbf{b}_d)$ is called *size-reduced* with factor $\eta \geq 1/2$ if its GSO family satisfies $|\mu_{i,j}| \leq \eta$ for all $1 \leq j < i \leq d$. The i-th vector $\mathbf{b}_i$ is *size-reduced* if $|\mu_{i,j}| \leq \eta$ for all $j \in [1, i-1]$. Size-reduction usually refers to $\eta = 1/2$, but it is essential for the floating-point LLLs to allow at least slightly larger factors η, since the $\mu_{i,j}$'s will be known only approximately.

The Lenstra-Lenstra-Lovász reduction. A basis $(\mathbf{b}_1, \ldots, \mathbf{b}_d)$ is called *LLL-reduced* with factor (δ, η) where $\delta \in (1/4, 1]$ and $\eta \in [1/2, \sqrt{\delta})$ if the

basis is size-reduced with factor η and if its GSO satisfies the $(d-1)$ following conditions, often called Lovász conditions:

$$\delta \cdot \left\| \mathbf{b}_{\kappa-1}^* \right\|^2 \leq \left\| \mathbf{b}_\kappa^* + \mu_{\kappa,\kappa-1} \mathbf{b}_{\kappa-1}^* \right\|^2,$$

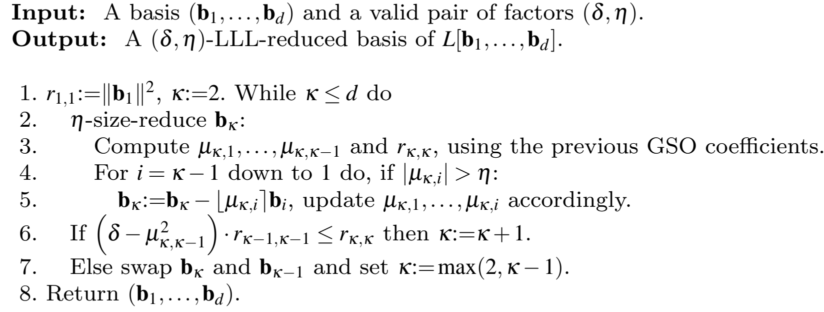
or equivalently $\left(\delta - \mu_{\kappa,\kappa-1}^2\right) \cdot r_{\kappa-1,\kappa-1} \leq r_{\kappa,\kappa}$. This implies that the norms of the GSO vectors $\mathbf{b}_1^*, \ldots, \mathbf{b}_d^*$ never drop too much: intuitively, the vectors are not far from being orthogonal. Such bases have useful properties. In particular, their first vector is relatively short. Theorem 1 is an adaptation of [35, Equations (1.8) and (1.9)].

Theorem 1. *Let $\delta \in (1/4, 1]$ and $\eta \in [1/2, \sqrt{\delta})$. Let $(\mathbf{b}_1, \ldots, \mathbf{b}_d)$ be a (δ, η)-LLL-reduced basis of a lattice L. Then:*

$$\|\mathbf{b}_1\| \leq \left(\frac{1}{\delta - \eta^2}\right)^{\frac{d-1}{4}} \cdot \mathrm{vol}(L)^{\frac{1}{d}},$$

$$\prod_{i=1}^{d} \|\mathbf{b}_i\| \leq \left(\frac{1}{\delta - \eta^2}\right)^{\frac{d(d-1)}{4}} \cdot \mathrm{vol}(L).$$

LLL-reduction classically refers to the factor pair $(3/4, 1/2)$ initially chosen in [35], in which case the quantity $\frac{1}{\delta - \eta^2}$ is conveniently equal to 2. But the closer δ and η respectively to 1 and $1/2$, the smaller the upper bounds in Theorem 1. In practice, one often selects $\delta \approx 1$ and $\eta \approx 1/2$, so that we almost have $\|\mathbf{b}_1\| \leq (4/3)^{\frac{d-1}{4}} \cdot \mathrm{vol}(L)^{\frac{1}{d}}$. It also happens that one selects weaker factors in order to speed up the execution of the algorithm (we discuss this strategy in Subsection 1.4.3).



Input: A basis $(\mathbf{b}_1, \ldots, \mathbf{b}_d)$ and a valid pair of factors (δ, η).
Output: A (δ, η)-LLL-reduced basis of $L[\mathbf{b}_1, \ldots, \mathbf{b}_d]$.

1. $r_{1,1} := \|\mathbf{b}_1\|^2$, $\kappa := 2$. While $\kappa \leq d$ do
2. η-size-reduce $\mathbf{b}_\kappa$:
3. Compute $\mu_{\kappa,1}, \ldots, \mu_{\kappa,\kappa-1}$ and $r_{\kappa,\kappa}$, using the previous GSO coefficients.
4. For $i = \kappa - 1$ down to 1 do, if $|\mu_{\kappa,i}| > \eta$:
5. $\mathbf{b}_\kappa := \mathbf{b}_\kappa - \lfloor \mu_{\kappa,i} \rceil \mathbf{b}_i$, update $\mu_{\kappa,1}, \ldots, \mu_{\kappa,i}$ accordingly.
6. If $\left(\delta - \mu_{\kappa,\kappa-1}^2\right) \cdot r_{\kappa-1,\kappa-1} \leq r_{\kappa,\kappa}$ then $\kappa := \kappa + 1$.
7. Else swap $\mathbf{b}_\kappa$ and $\mathbf{b}_{\kappa-1}$ and set $\kappa := \max(2, \kappa - 1)$.
8. Return $(\mathbf{b}_1, \ldots, \mathbf{b}_d)$.

Fig. 1.1 The LLL algorithm.

The LLL algorithm. We give in Figure 1.1 a description of LLL that we will use to explain its floating-point variants. The LLL algorithm obtains in polynomial time a (δ, η)-reduced basis, even if one chooses $\eta = 1/2$. The

factor $\delta < 1$ can be chosen arbitrarily close to 1. It is unknown whether polynomial time complexity can be achieved or not for $\delta = 1$ (partial results can be found in [6] and [36]).

The floating-point LLL algorithms do not achieve $\eta = 1/2$, because the GSO coefficients are known only approximately. Choosing $\eta = 1/2$ in these algorithms may make them loop forever. Similarly, one has to relax the LLL factor δ, but this relaxation only adds up with the already necessary relaxation of δ in the classical LLL algorithm. The LLL factor η can be chosen arbitrarily close to 1/2 in the provable floating-point L^2 algorithm of Nguyen and Stehlé [43] (to be described in Section 1.3) which terminates in quadratic time (without fast integer multiplication) with respect to the bit-size of the matrix entries. Finally, $\eta = 1/2$ can also be achieved within the same complexity: firstly run the L^2 algorithm on the given input basis with the same factor δ and a factor $\eta \in (1/2, \sqrt{\delta})$; and secondly run the LLL algorithm on the output basis. One can notice that the second reduction is simply a size-reduction and can be performed in the prescribed time.

Remarkable variables in the LLL algorithm. The LLL index $\kappa(t)$ denotes the vector under investigation at the t-th loop iteration of the algorithm. Its initial value is 2 and at the end of the execution, one has $\kappa(\tau+1) = d+1$, where τ is the number of loop iterations and $\kappa(\tau+1)$ is the value of κ at the end of the last loop iteration. We will also use the index $\alpha(t)$ (introduced in [43]), which we define below and illustrate in Figure 1.2. It is essentially the smallest swapping index since the last time the index κ was at least $\kappa(t)$ (this last time is rigorously defined below as $\phi(t)$).

Definition 2. Let t be a loop iteration. Let $\phi(t) = \max(t' < t, \kappa(t') \geq \kappa(t))$ if it exists and 1 otherwise, and let $\alpha(t) = \min(\kappa(t'), t' \in [\phi(t), t-1]) - 1$.

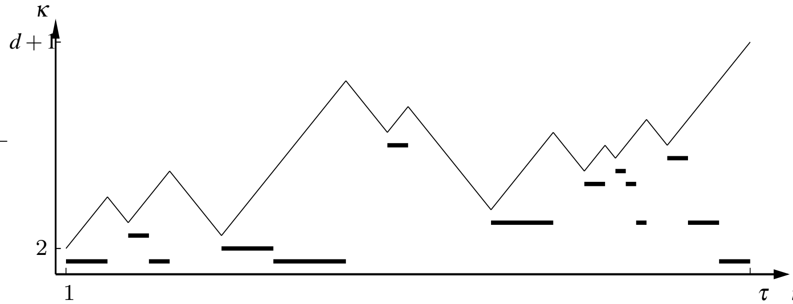


Fig. 1.2 A possible curve for $\kappa(t)$ (thin continuous line), with the corresponding curve for $\alpha(t)$ (thick line when κ increases, and same as κ otherwise)

The index $\alpha(t)$ has the remarkable property that between the loop iterations $\phi(t)$ and t, the vectors $\mathbf{b}_1, \ldots, \mathbf{b}_{\alpha(t)}$ remain unchanged: because κ remains

larger than $\alpha(t)$, these first vectors are not swapped nor size-reduced between these iterations.

1.3 The Provable Floating-Point LLL Algorithms

When floating-point calculations are mentioned in the context of the LLL algorithm, this systematically refers to the underlying Gram-Schmidt orthogonalisation. The transformations on the basis and the basis itself remain exact, because one wants to preserve the lattice while reducing it. The LLL algorithm heavily relies on the GSO. For example, the LLL output conditions involve all the quantities $\mu_{i,j}$ for $j < i \leq d$ and $\|\mathbf{b}_i^*\|^2$ for $i \leq d$. The floating-point arithmetic is used on these GSO quantities $\mu_{i,j}$ and $\|\mathbf{b}_i^*\|^2$.

In this section, we are to describe three ways of implementing this idea: the first way is the most natural solution, but fails for different reasons, that we emphasize because they give some intuition about the provable variants; the second one, due to Schnorr [50], is provable but suffers from a number of practical drawbacks; and the last one, due to Nguyen and Stehlé [43], was introduced recently and seems more tractable in practice.

The following table summarises the complexities of two rational LLL algorithms and the two provable floating-point LLLs described in this section. The second line contains the required precisions, whereas the last line consists of the best known complexity upper bounds. The variant of Kaltofen [28] differs only slightly from LLL. The main improvement of the latter is to analyse more tightly the cost of the size-reductions, providing a complexity bound of total degree 8 instead of 9. This bound also holds for the LLL algorithm. On the floating-point side, both Schnorr's algorithm and L^2 have complexities of total degree 7, but the complexity bound of L^2 is always better and is quadratic with respect to $\log B$, the bit-size of the input matrix entries.

LLL [35]	Kaltofen [28]	Schnorr [50]	L^2 [43]
$O(d \log B)$	$O(d \log B)$	$\geq 12d + 7\log_2 B$	$d \log_2 3 \approx 1.58d$
$O(d^5 n \log^3 B)$	$O(d^4 n(d + \log B)\log^2 B)$	$O(d^3 n(d + \log B)^2 \log B)$	$O(d^4 n \log B(d + \log B))$

Fig. 1.3 Complexity bounds of the original LLL and the provable floating-point LLL algorithms.

1.3.1 A First Attempt

A natural attempt to define a floating-point LLL algorithm is as follows: one keeps the general structure of LLL as described in Figure 1.1, and computes approximations to the GSO quantities, by converting into floating-point arithmetic the formulas that define them (as given in Subsection 1.2.2). The scalar product $\langle \mathbf{b}_i, \mathbf{b}_j \rangle$ is approximated by the quantity $\bar{g}_{i,j}$, computed as follows:

$$\bar{g}_{i,j}{:=}0. \texttt{ For } k \texttt{ from 1 to } n\texttt{, do } \bar{g}_{i,j}{:=}\diamond(\bar{g}_{i,j}+\diamond(\diamond(\mathbf{b}_i[k])\cdot\diamond(\mathbf{b}_j[k]))).$$

Similarly, the quantities $r_{i,j}$ and $\mu_{i,j}$ are approximated respectively by the $\bar{r}_{i,j}$'s and $\bar{\mu}_{i,j}$'s, computed as follows:

$$\bar{r}_{i,j}{:=}\bar{g}_{i,j}. \texttt{ For } k \texttt{ from 1 to } j-1\texttt{, do } \bar{r}_{i,j}{:=}\diamond\left(\bar{r}_{i,j}-\diamond\left(\bar{r}_{i,k}\cdot\bar{\mu}_{j,k}\right)\right).$$
$$\bar{\mu}_{i,j}{:=}\diamond(\bar{r}_{i,j}/\bar{r}_{j,j}).$$

As a first consequence, since the $\mu_{i,j}$'s are known only approximately, one cannot ensure ideal size-reduction anymore. One has to relax the condition $|\mu_{i,j}| \leq 1/2$ into the condition $|\mu_{i,j}| \leq \eta$ for some $\eta > 1/2$ that takes into account the inaccuracy of the $\bar{\mu}_{i,j}$'s.

This first attempt suffers from three major drawbacks. Firstly, the scalar products can be miscalculated. More precisely, the quantity $\bar{g}_{i,j}$ is a sum of floating-point numbers and the classical phenomena of cancellation and loss of precision can occur. We do not have better than the following error bound:

$$\left|\bar{g}_{i,j} - \langle \mathbf{b}_i, \mathbf{b}_j \rangle\right| \leq f(n,\ell) \cdot \sum_{k \leq n} |\mathbf{b}_i[k]| \cdot \left|\mathbf{b}_j[k]\right|,$$

where the function f depends on the precision ℓ and the number n of elements to sum. Unfortunately, such a summation prevents us from getting absolute error bounds on the $\mu_{i,j}$'s. In order to obtain an absolute error bound on $\mu_{i,j} = \frac{\langle \mathbf{b}_i, \mathbf{b}_j \rangle}{r_{i,i}}$, one would like an error on $\langle \mathbf{b}_i, \mathbf{b}_j \rangle$ which is no more than proportional to $r_{i,i} = \|\mathbf{b}_i^*\|^2$. We illustrate this with an example in dimension 2 and double precision. Consider the columns of the following matrix:

$$\begin{bmatrix} 1 & 2^{100}+2^{40} \\ -1 & 2^{100}-2^{40} \end{bmatrix}.$$

Here $r_{1,1} = 2$ and we would like the error on $\langle \mathbf{b}_2, \mathbf{b}_1 \rangle$ to be small compared to that quantity. If the scalar product of the two vectors is computed by first rounding the matrix entries to double precision, then it is estimated to 0. This implies that the computed $\bar{\mu}_{2,1}$ is 0, and the basis is deemed LLL-reduced. But it is not, since in fact $\mu_{2,1} = 2^{40}$, which contradicts the size-reduction condition. If one changes the values 2^{40} by any values below 2^{100}, one sees that the less significant bits are simply ignored though they may still be contributing significantly to $\mu_{2,1}$. In order to test the size-reduction conditions from the basis matrix, it seems necessary to use a precision which

is at least as large as the bit-length of the input matrix entries, which may be very expensive.

Secondly, the precision may not be sufficient to perform the size-reductions completely. It can easily be illustrated by an example. Consider the following lattice basis:

$$\begin{bmatrix} 1 & 2^{54}+1 \\ 0 & 1 \end{bmatrix}.$$

The algorithm will compute $\bar{\mu}_{2,1} = 2^{54}$: the bit-length of the true quantity is too large to be stored in a double precision floating-point number. Then it will try to size-reduce the second vector by performing the operation $\mathbf{b}_2 := \mathbf{b}_2 - 2^{54}\mathbf{b}_1 = (1,1)^t$. It will then check that Lovász's condition is satisfied and terminate. Unfortunately, the output basis is still not size-reduced, because $\mu_{2,1} = 1$. One can change the example to make $\mu_{2,1}$ as large as desired. The trouble here is that the mantissa size is too small to handle the size-reduction. Either more precision or a reparation routine seems necessary. Such a reparation process will be described in Subsection 1.3.3.

The third weakness of the first attempt is the degradation of precision while computing the GSO coefficients. Indeed, a given $\bar{r}_{i,j}$ is computed from previously computed and already erroneous quantities $\bar{r}_{i,k}$ and $\bar{\mu}_{j,k}$, for $k \leq j$. The floating-point errors not only add up, but also get amplified. No method to prevent this amplification is known, but it is known how to bound and work around the phenomenon: such techniques come from the field of numerical analysis. In particular, it seems essential for the good numerical behaviour of the LLL algorithm to always consider a vector $\mathbf{b}_k$ such that all previous vectors $\mathbf{b}_i$ for $i < k$ are LLL-reduced: this means that when one is computing the orthogonalisation of a vector with respect to previous vectors, the latter are always LLL-reduced and therefore fairly orthogonal, which is good for the numerical behaviour. The structure of the LLL algorithm as described in Figure 1.1 guarantees this property.

1.3.2 Schnorr's Algorithm

Schnorr [50] described the first provable variant of the LLL algorithm relying on floating-point arithmetic. Instead of using GSO coefficients represented as rational numbers of bit-lengths $O(d \log B)$, Schnorr's algorithm approximates them by arbitrary precision floating-point numbers, of mantissa size $\ell = O(d + \log B)$. This provides a gain of 2 in the total degree of the polynomial complexity of LLL: from $O(d^5 n \log^3 B)$ to $O(d^3 n (d + \log B)^2 \log B)$.

In Figure 1.4, we give a description of this algorithm. It uses exact integer operations on the basis vectors (Steps 2 and 5), and approximate operations on the inverses of the partial Gram matrices $G_k = (\langle \mathbf{b}_i, \mathbf{b}_j \rangle)_{i,j \leq k}$ and the inverse $(\nu_{i,j})_{i,j \leq d}$ of the lower triangular matrix made of the $\mu_{i,j}$'s. These op-

erations are not standard floating-point operations, since most of them are in fact exact operations on approximate values: in floating-point arithmetic, the result of a basic arithmetic operation is a floating-point number closest to the true result, whereas here the true result is kept, without any rounding. This is the case everywhere, except at Step 8, where the quantities are truncated in order to avoid a length blow-up. The truncation itself is similar to fixed-point arithmetic since it keeps a given number of bits after the point instead of keeping a given number of most significant bits. It can be checked that all quantities computed never have more than $c \cdot \ell$ bits after the point, for some small constant c depending on the chosen number of iterations at Step 7. At Step 7, a few steps of Schulz's iteration are performed. Schulz's iteration [57] is a classical way to improve the accuracy of an approximate inverse (here $\overline{G_k^{-1}}$) of a known matrix (here G_k). This is a matrix generalisation of Newton's iteration for computing the inverse of a real number.

Input: A basis $(\mathbf{b}_1,\ldots,\mathbf{b}_d)$, a precision ℓ.
Output: A $(0.95, 0.55)$-reduced basis of the lattice spanned by the $\mathbf{b}_i$'s.

1. $\kappa{:=}2, \mathbf{b}_1^*{:=}\mathbf{b}_1$.
2. $r{:=}\lfloor \langle \mathbf{b}_\kappa, \bar{\mathbf{b}}_{\kappa-1}^* \rangle / \|\bar{\mathbf{b}}_{\kappa-1}^*\|^2 \rceil$, $\mathbf{b}_\kappa{:=}\mathbf{b}_\kappa - r\mathbf{b}_{\kappa-1}$.
3. If $(\|\mathbf{b}_\kappa\|^2 - \sum_{j<\kappa-1} \langle \mathbf{b}_\kappa, \bar{\mathbf{b}}_j^* \rangle / \|\bar{\mathbf{b}}_j^*\|^2) \cdot 1.025 \geq \|\bar{\mathbf{b}}_{\kappa-1}^*\|^2$, go to Step 5. Otherwise:
4. Exchange $\mathbf{b}_\kappa$ and $\mathbf{b}_{\kappa-1}$, $\kappa{:=}\max(2, \kappa-1)$. Update $\overline{G_\kappa^{-1}}$ and go to Step 2.
5. For j from $\kappa-2$ down to 1, do $r{:=}\lfloor \langle \mathbf{b}_\kappa, \bar{\mathbf{b}}_j^* \rangle / \|\bar{\mathbf{b}}_j^*\|^2 \rceil$, $\mathbf{b}_\kappa{:=}\mathbf{b}_\kappa - r\mathbf{b}_j$.
6. Compute a first approximation of $\nu_{\kappa,1}, \ldots, \nu_{\kappa,\kappa-1}, \mathbf{b}_\kappa^*, G_\kappa^{-1}$, from the $\mathbf{b}_i$'s and the matrix $\overline{G_{\kappa-1}^{-1}}$.
7. Use a finite number of iterations of Schulz's method on $\overline{G_k^{-1}}$ using G_k. This helps improving the approximations of $\nu_{\kappa,1}, \ldots, \nu_{\kappa,\kappa-1}, \mathbf{b}_\kappa^*$ and G_κ^{-1}.
8. Truncate the $\bar{\nu}_{\kappa,i}$'s to ℓ bits after the point. Compute the corresponding vectors $\bar{\mathbf{b}}_\kappa^*$ and $\overline{G_\kappa^{-1}}$.
9. $\kappa{:=}\kappa+1$. If $\kappa \leq n$, go to Step 2.

Fig. 1.4 Schnorr's algorithm.

Schnorr proved that by taking a precision $\ell = c_1 \cdot d + c_2 \cdot \log B$ for some explicitly computable constants c_1 and c_2, the algorithm terminates and returns a $(0.95, 0.55)$-LLL-reduced basis. The constants 0.95 and 0.55 can be chosen arbitrarily close but different to respectively 1 and 0.5, by changing the constants c_1 and c_2 as well as the constant 1.025 from Step 3. Finally, it can be checked that the bit-cost of the algorithm is $O\left(d^3 n (d + \log B)^2 \log B\right)$.

This algorithm prevents the three problems of the naive floating-point LLL from occurring: the inaccuracy of the scalar products is avoided because they are always computed exactly; the incomplete size-reductions cannot occur because the precision is set large enough to guarantee that any size-reduction is performed correctly and fully at once; the accumulation of inaccuracies is restrained because most of the operations performed on approximations

are done exactly, so that few errors may add up, and the amplification of the errors (due to a bad conditioning of the problem) is compensated by the large precision.

Schnorr's algorithm gives a first answer to the question of using approximate GSO quantities within the LLL algorithm, but:

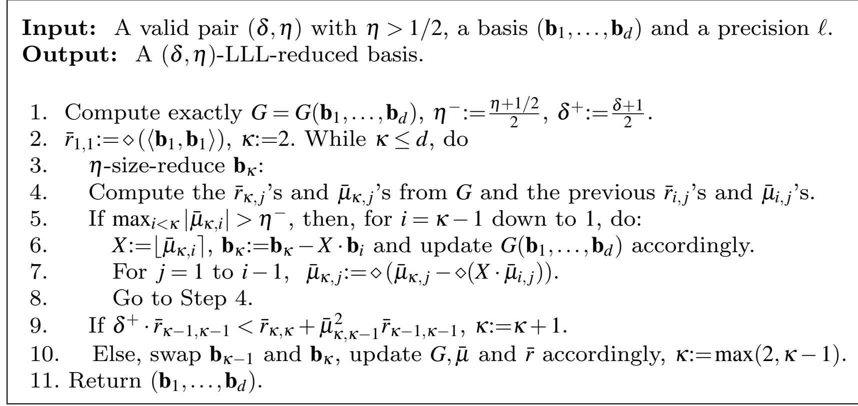
- The constants c_1 and c_2 on which the precision depends may be large. What is most annoying is that the precision actually depends on $\log B$. This means that the approximate operations on the GSO still dominate the integer operations on the basis matrix.
- As explained above, it is not using standard floating-point arithmetic, but rather a mix between exact computations on approximate values and arbitrary precision fixed-point arithmetic.

1.3.3 The L^2 Algorithm

The L^2 algorithm was introduced by Nguyen and Stehlé [43] in 2005. It is described in Figure 1.5. L^2 is a variant of the LLL algorithm relying on arbitrary precision floating-point arithmetic for the underlying Gram-Schmidt orthogonalisation, in a provable way. Apart from giving a sound basis for floating-point calculations within LLL, it is also the sole variant of LLL that has been proven to admit a quadratic bit-complexity with respect to the bit-size of the input matrix entries. This latter property is very convenient since LLL can be seen as a multi-dimensional generalisation of Euclid's gcd algorithm, Gauss' two-dimensional lattice reduction algorithm and the three and four dimensional greedy algorithm of Semaev, Nguyen and Stehlé [58, 42], which all admit quadratic complexity bounds. This property, from which the name of the algorithm comes, arguably makes it a natural variant of LLL.

In L^2, the problem of scalar product cancellations is handled very simply, since all the scalar products are known exactly during the whole execution of the algorithm. Indeed, the Gram matrix of the initial basis matrix is computed at the beginning of the algorithm and updated for each change of the basis vectors. In fact, the algorithm operates on the Gram matrix and the computed transformations are forwarded to the basis matrix. It can be seen that this can be done with only a constant factor overhead in the overall complexity. Secondly, the size-reduction procedure is modified into a lazy size-reduction. One size-reduces as much as possible given the current knowledge of the Gram-Schmidt orthogonalisation, then recomputes the corresponding Gram-Schmidt coefficients from the exact Gram matrix and restarts the lazy size-reduction until the vector under question stays the same. When this happens, the vector is size-reduced and the corresponding Gram-Schmidt coefficients are well approximated. This lazy size-reduction was already contained inside NTL's LLL, and described in [33] in the context of a heuristic floating-point LLL algorithm based on Householder transformations. In this

context, fixing $\eta = 1/2$ can have dramatic consequences: apart from asking for something which is not reachable with floating-point computations, the lazy size-reduction (i.e., the inner loop between Steps 4 and 8 in Figure 1.5) may loop forever. Finally, an a priori error analysis provides a bound on the loss of accuracy, which provides the provably sufficient precision.



Input: A valid pair (δ, η) with $\eta > 1/2$, a basis $(\mathbf{b}_1, \ldots, \mathbf{b}_d)$ and a precision ℓ.
Output: A (δ, η)-LLL-reduced basis.

1. Compute exactly $G = G(\mathbf{b}_1, \ldots, \mathbf{b}_d)$, $\eta^- := \frac{\eta+1/2}{2}$, $\delta^+ := \frac{\delta+1}{2}$.
2. $\bar{r}_{1,1} := \diamond(\langle \mathbf{b}_1, \mathbf{b}_1 \rangle)$, $\kappa := 2$. While $\kappa \leq d$, do
3. η-size-reduce $\mathbf{b}_\kappa$:
4. Compute the $\bar{r}_{\kappa,j}$'s and $\bar{\mu}_{\kappa,j}$'s from G and the previous $\bar{r}_{i,j}$'s and $\bar{\mu}_{i,j}$'s.
5. If $\max_{i<\kappa} |\bar{\mu}_{\kappa,i}| > \eta^-$, then, for $i = \kappa - 1$ down to 1, do:
6. $X := \lfloor \bar{\mu}_{\kappa,i} \rceil$, $\mathbf{b}_\kappa := \mathbf{b}_\kappa - X \cdot \mathbf{b}_i$ and update $G(\mathbf{b}_1, \ldots, \mathbf{b}_d)$ accordingly.
7. For $j = 1$ to $i - 1$, $\bar{\mu}_{\kappa,j} := \diamond(\bar{\mu}_{\kappa,j} - \diamond(X \cdot \bar{\mu}_{i,j}))$.
8. Go to Step 4.
9. If $\delta^+ \cdot \bar{r}_{\kappa-1,\kappa-1} < \bar{r}_{\kappa,\kappa} + \bar{\mu}^2_{\kappa,\kappa-1} \bar{r}_{\kappa-1,\kappa-1}$, $\kappa := \kappa + 1$.
10. Else, swap $\mathbf{b}_{\kappa-1}$ and $\mathbf{b}_\kappa$, update $G, \bar{\mu}$ and $\bar{r}$ accordingly, $\kappa := \max(2, \kappa - 1)$.
11. Return $(\mathbf{b}_1, \ldots, \mathbf{b}_d)$.

Fig. 1.5 The L^2 algorithm.

Theorem 2 ([43, Theorem 1]). *Let (δ, η) such that $1/4 < \delta < 1$ and $1/2 < \eta < \sqrt{\delta}$. Let $c = \log \frac{(1+\eta)^2+\varepsilon}{\delta-\eta^2} + C$, for some arbitrary $\varepsilon \in (0, 1/2)$ and $C > 0$. Given as input a d-dimensional lattice basis $(\mathbf{b}_1, \ldots, \mathbf{b}_d)$ in $\mathbb{Z}^n$ with $\max_i \|\mathbf{b}_i\| \leq B$, the L^2 algorithm of Figure 1.5 with precision $\ell = cd + o(d)$ outputs a (δ, η)-LLL-reduced basis in time $O\left(d^4 n(d + \log B) \log B\right)$. More precisely, if τ denotes the number of iterations of the loop between Steps 3 and 10, then the running time is $O\left(d^2 n(\tau + d \log(dB))(d + \log B)\right)$.*

The precision $\ell = cd + o(d)$ can be made explicit from the correctness proof of [43]. It suffices that the following inequality holds, for some arbitrary $C > 0$:

$$d^2 \left(\frac{(1+\eta)^2 + \varepsilon}{\delta - \eta^2} \right)^d 2^{-\ell + 10 + Cd} \leq \min\left(\varepsilon, \eta - \frac{1}{2}, 1 - \delta \right).$$

Notice that with double precision (i.e., $\ell = 53$), the dimension up to which the above bound guarantees that L^2 will work correctly is very small. Nevertheless, the bound is likely to be loose: in the proof of [43], the asymptotically negligible components are chosen to simplify the error analysis. Obtaining tighter bounds for the particular case of the double precision would be interesting in practice for small dimensions. For larger dimensions, the non-dominating components become meaningless. Asymptotically, in the case of LLL-factors (δ, η) that are close to $(1, 1/2)$, a floating-point precision $\ell = 1.6 \cdot d$ suffices.

Here is a sketch of the complexity analysis of the L^2 algorithm. We refer to [41] for more details.

1. There are $\tau = O(d^2 \log B)$ loop iterations.
2. In a given loop iteration, there can be up to $O\left(1 + \frac{\log B}{d}\right)$ iterations within the lazy size-reduction. However, most of the time there are only $O(1)$ such loop iterations. The lengthy size-reductions cannot occur often during a given execution of L^2, and are compensated by the other ones. In the rigorous complexity analysis, this is formalised by an amortised analysis (see below for more details). In practice, one can observe that there are usually two iterations within the lazy size-reduction: the first one makes the $|\mu_{\kappa,i}|$'s smaller than η and the second one recomputes the $\mu_{\kappa,i}$'s and $r_{\kappa,i}$'s with better accuracy. This is incorrect in full generality, especially when the initial $\mu_{\kappa,i}$'s are very large.
3. In each iteration of the lazy size-reduction, there are $O(dn)$ arithmetic operations.
4. Among these arithmetic operations, the most expensive ones are those related to the coefficients of the basis and Gram matrices: these are essentially multiplications between integers of lengths $O(\log B)$ and the computed X's, which can be represented on $O(d)$ bits.

The proof of the quadratic complexity bound generalises the complexity analysis of Euclid's gcd algorithm. In Euclid's algorithm, one computes the gcd of two integers $r_0 > r_1 > 0$ by performing successive euclidean divisions: $r_{i+1} = r_{i-1} - q_i r_i$, with $|r_{i+1}| < |r_i|$, until one gets 0. Standard arguments show that the number of divisions is $O(\log r_0)$. To obtain a quadratic complexity bound for Euclid's algorithm, one has to compute q_i by using only some (essentially $\log q_i \approx \log |r_{i-1}| - \log |r_i|$) of the most significant bits of r_{i-1} and r_i, to get r_{i+1} with a bit-complexity $O(\log r_0 \cdot (1 + \log |r_{i-1}| - \log |r_i|))$. It is crucial to consider this bound instead of the weaker $O\left(\log^2 |r_{i-1}|\right)$ to be able to use an amortised cost analysis: the worst-case cost of a sequence of steps can be much lower than the sum of the worst cases of each step of the sequence. In the quadratic complexity bound of the L^2 algorithm, the euclidean division becomes the lazy size-reduction and the term $O\left(\log r_0 \cdot (1 + \log |r_{i-1}| - \log |r_i|)\right)$ becomes $O\left(\log B \cdot (d + \log \|\mathbf{b}_{\kappa(t)}\| - \log \|\mathbf{b}_{\alpha(t)}\|)\right)$ for the t-th loop iteration: intuitively, the cost of the size-reduction does not depend on the $\alpha(t) - 1$ first vectors, since the vector $\mathbf{b}_{\kappa(t)}$ is already size-reduced with respect to them. In the analysis of Euclid's algorithm, terms cancel out as soon as two consecutive steps are considered, but in the case of L^2, one may need significantly more than two steps to observe a possible cancellation. The following lemma handles this difficulty.

Lemma 1 ([43, Lemma 2]). *Let $k \in [2,d]$ and $t_1 < \ldots < t_k$ be loop iterations of the L^2 algorithm such that for any $j \leq k$, we have $\kappa(t_j) = k$. For any loop iteration t and any $i \leq d$, we define $\mathbf{b}_i^{(t)}$ as the i-th basis vector at the beginning of the t-th loop iteration. Then there exists $j < k$ such that:*

$$d(\delta-\eta^2)^{-d}\cdot\left\|\mathbf{b}_{\alpha(t_j)}^{(t_j)}\right\|\geq\left\|\mathbf{b}_k^{(t_k)}\right\|.$$

This result means that when summing all the bounds of the costs of the successive loop iterations, i.e., $O\left(\log B\cdot(d+\log\|\mathbf{b}_{\kappa(t)}\|-\log\|\mathbf{b}_{\alpha(t)}\|)\right)$, some quasi-cancellations of the following form occur: a term $\log\|\mathbf{b}_{\kappa(t)}\|$ can be cancelled out with a term $\log\|\mathbf{b}_{\alpha(t')}\|$, where the relationship between t' and t is described in the lemma. This is not exactly a cancellation, since the difference of the two terms is replaced by $O(d)$ (which does not involve the size of the entries).

The proof of correctness of the L^2 algorithm relies on a forward error analysis of the Cholesky factorisation algorithm while applied to a Gram matrix of a basis whose first vectors are already LLL-reduced. We give here a sketch of the error analysis in the context of a fully LLL-reduced basis (i.e., the whole basis is LLL-reduced). This shows the origin of the term $\frac{(1+\eta)^2}{\delta-\eta^2}$ in Theorem 2.

We define $err_j=\max_{i\in[j,d]}\frac{|\bar{r}_{i,j}-r_{i,j}|}{r_{j,j}}$, i.e., the approximation error on the $r_{i,j}$'s relatively to $r_{j,j}$, and we bound its growth as j increases. We have:

$$err_1=\max_{i\leq d}\frac{|\diamond\langle\mathbf{b}_i,\mathbf{b}_1\rangle-\langle\mathbf{b}_i,\mathbf{b}_1\rangle|}{\|\mathbf{b}_1\|^2}\leq 2^{-\ell}\cdot\max_{i\leq d}\frac{|\langle\mathbf{b}_i,\mathbf{b}_1\rangle|}{\|\mathbf{b}_1\|^2}=2^{-\ell}\cdot\max_{i\leq d}|\mu_{i,1}|\leq 2^{-\ell},$$

because of the size-reduction condition. We now choose $j\in[2,d]$. We have, for any $i\leq d$ and any $k<j$:

$$|\bar{\mu}_{i,k}-\mu_{i,k}|\lesssim\left|\frac{r_{k,k}}{\bar{r}_{k,k}}\right|err_k+|r_{i,k}|\left|\frac{1}{\bar{r}_{k,k}}-\frac{1}{r_{k,k}}\right|\lesssim(\eta+1)\cdot err_k,$$

where we neglected low-order terms and used the fact that $|r_{i,k}|\leq\eta\cdot r_{k,k}$, which comes from the size-reduction condition. This implies that:

$$\begin{aligned}|\diamond(\bar{\mu}_{j,k}\cdot\bar{r}_{i,k})-\mu_{j,k}r_{i,k}| &\lesssim |\bar{\mu}_{j,k}-\mu_{j,k}|\cdot|\bar{r}_{i,k}|+|\mu_{j,k}|\cdot|\bar{r}_{i,k}-r_{i,k}|\\ &\lesssim \eta(\eta+2)\cdot err_k\cdot\|\mathbf{b}_k^*\|^2,\end{aligned}$$

where we also neglected low-order terms and used the size-reduction condition twice. Thus,

$$err_j\lesssim\eta(\eta+2)\sum_{k<j}\frac{\|\mathbf{b}_k^*\|^2}{\|\mathbf{b}_j^*\|^2}err_k\lesssim\eta(\eta+2)\sum_{k<j}(\delta-\eta^2)^{k-j}\cdot err_k,$$

by using the fact that Lovász's conditions are satisfied. This finally gives $err_j\lesssim\left(\frac{(1+\eta)^2}{\delta-\eta^2}\right)^j\cdot err_1$.

1.4 Heuristic Variants and Implementations of the Floating-Point LLL

Floating-point arithmetic has been used in the LLL implementations since the early 1980's, but only very few papers describe how this should be done in order to balance efficiency and correctness. The reference for LLL implementers is the article by Schnorr and Euchner on practical lattice reduction [52, 53]. Until very recently, all the fastest LLL implementations were relying on it, including the one in Victor Shoup's NTL, Allan Steel's LLL in Magma, and LiDIA's LLL (written by Werner Backes, Thorsten Lauer, Oliver van Sprang and Susanne Wetzel). Magma's LLL is now relying on the L^2 algorithm. In this section, we describe the Schnorr-Euchner heuristic floating-point LLL, and explain how to turn the L^2 algorithm into an efficient and reliable code.

1.4.1 The Schnorr-Euchner Heuristic LLL

Schnorr-Euchner's floating-point LLL follows very closely the classical description of LLL. It mimics the rational LLL while trying to work around the three pitfalls of the naive strategy (see Section 1.3). Let us consider these three difficulties separately.

It detects cancellations occurring during the computation of scalar products (at Step 3 of the algorithm of Figure 1.1), by comparing their computed approximations with the (approximate) product of the norms of the corresponding vectors. Since norms consist in summing positive values, no cancellation occurs while computing them approximately, and the computed values are therefore very reliable. If more than half the precision within the scalar product is likely to be lost (i.e., the ratio between the magnitude of the computed value and the product of the norms is smaller than $2^{-\ell/2}$ where ℓ is the precision), the scalar product is computed exactly (with the integer vectors and integer arithmetic) and then rounded to a closest double. As a consequence, not significantly more than half the precision can be lost while computing a scalar product. In NTL's LLL (which implements the Schnorr-Euchner variant), Victor Shoup replaced the 50% loss of precision test by a stronger requirement of not losing more than 15% of the precision.

Secondly, if some coefficient $\mu_{i,j}$ is detected to be large (between Steps 3 and 4 of the algorithm of Figure 1.1), i.e., more than $2^{\ell/2}$ where ℓ is the precision, then another size-reduction will be executed after the current one. This prevents incomplete size-reductions from occurring.

Finally, the algorithm does not tackle the error amplification due to the Gram-Schmidt orthogonalisation process: one selects the double precision and hopes for the best.

Let us now discuss these heuristics. If the scalar products are detected to be cancelling frequently, they will often be computed exactly with integer arithmetic. In that situation, one should rather keep the Gram matrix and update it. On the theoretical side, the Schnorr-Euchner strategy for scalar products prevents one from getting a quadratic bit complexity. In practice, it may slow down the computation significantly. In particular, this occurs when two vectors have much different lengths and are nearly orthogonal. This may happen quite frequently in some applications of LLL, one of them being Coppersmith's method [14]. In the table of Figure 1.6, we compare NTL's `LLL_XD` with Magma's `LLL` for input lattice bases that correspond to the use of Coppersmith's method for the problem of factoring with high bits known, such as described in [39], for a 1024 bit RSA modulus $p \cdot q$ and for different numbers of most significant bits of p known. The experiments were performed with NTL-5.4 and Magma-2.13, both using GNU MP for the integer arithmetic, on a Pentium double-core 3.00 GHz. In both cases, the chosen parameters were $\delta = 0.75$ and η very close to $1/2$, and the transformation matrix was not computed. The timings are given in seconds. Here Magma's `LLL` uses the Gram matrix, whereas NTL's `LLL_XD` recomputes the scalar products from the basis vectors if a large cancellation is detected. In these examples, NTL spends more than 60% of the time recomputing the scalar products from the basis vectors.

Number of unknown bits of p	220	230	240	245
Dimension of the lattice	17	22	34	50
NTL's `LLL_XD`	13.1	78.6	1180	13800
Time to compute the scalar products exactly	8.05	51.6	914	11000
Magma's `LLL`	8.47	44.5	710	10000

Fig. 1.6 Comparison between NTL's `LLL_XD` and Magma's `LLL` for lattice bases arising in Coppersmith's method applied to the problem of factoring with high bits known.

Secondly, when the $\mu_{i,j}$'s are small enough, they are never recomputed after the size-reduction. This means that they are known with a possibly worse accuracy. NTL's LLL is very close to Schnorr-Euchner's heuristic variant but differs on this point: a routine similar to the lazy size-reduction of the L^2 algorithm is used. Shoup's strategy consists in recomputing the $\mu_{k,i}$'s as long as one of them seems (we know them only approximately) larger than η, where η is extremely close to $1/2$ (the actual initial value being $1/2+2^{-26}$), and recall the size-reduction. When unexpectedly long lazy size-reductions are encountered (the precise condition being more than 10 iterations), the accuracy on the GSO coefficients is deemed very poor, and η is increased slightly to take into account larger errors. This is a good strategy on the

short term since it may accept a larger but manageable error. However, on the long term, weakening the size-reduction condition may worsen the numerical behaviour (see Theorem 2 and Section 1.5) and thus even larger errors and therefore stronger misbehaviours are likely to occur.

The fact that the error amplification is not dealt with would not be a problem if there was a way to detect misbehaviours and to handle them. This amplification may cause meaningless calculations: if the current GSO coefficients are very badly approximated, then the performed Lovász tests are meaningless with respect to the basis; it implies that the performed operations may be irrelevant, and not reducing the basis at all; nothing ensures then that the execution will terminate, since it is too different from the execution of the rational LLL. In NTL's LLL, one tries a given precision. If the execution seems too long, the user has to stop it, and restart with some higher precision or some more reliable variant, without knowing if the algorithm was misbehaving (in which case increasing the precision may help), or just long to finish (in which case increasing the precision will slow down the process even more). The lack of error detection and interpretation can be quite annoying from the user point of view: in NTL and LiDia, one may have to try several variants before succeeding.

1.4.2 Implementing the L^2 Algorithm

We now consider the task of implementing the L^2 algorithm (described in Figure 1.5). In practice, one should obviously try to use heuristic variants before falling down to the guaranteed L^2 algorithm. To do this, we allow ourselves to weaken the L^2 algorithm in two ways: we may try not to use the Gram matrix but the basis matrix only, and we may try to use a floating-point precision which is much lower than the provably sufficient one. We describe here such a possible implementation.

We consider four layers for the underlying floating-point arithmetic:

- Double precision: it is extremely fast, but has a limited exponent (11 bits) and a limited precision (53 bits). The exponent limit allows to convert integers that have less than 1022 bits (approximately half if one wants to convert the Gram matrix as well). The limited precision is less annoying, but prevents from considering high dimensions.
- Doubles with additional exponents (dpes): it is still quite fast, but the precision limit remains.
- Heuristic extended precision: if more precision seems to be needed, then one will have to use arbitrary precision floating-point numbers. According to the analysis of the L^2 algorithm, a precision $\ell \approx \log \frac{(1+\eta)^2}{\delta-\eta^2} \cdot d$ always suffices. Nevertheless, one is allowed to try a heuristic lower precision first.

- Provable extended precision: use arbitrary precision floating-point numbers with a provably sufficient mantissa size of $\ell \approx \log \frac{(1+\eta)^2}{\delta-\eta^2} \cdot d$ bits.

In [33], Koy and Schnorr suggest to extend the 53 bit long double precision to a precision of 106 bits. This is an interesting additional layer between the double precision and the arbitrary precision, since it can be implemented in an efficient way with a pair of doubles (see [31, Chapter 4.2.2, Exercise 21]).

One can also perform the computations with or without the Gram matrix. If it is decided not to consider the Gram matrix, then the scalar products are computed from floating-point approximations of the basis matrix entries. As mentioned previously, cancellations may occur and the computed scalar products may be completely incorrect. Such misbehaviours will have to be detected and handled. If it is decided to consider the Gram matrix, then there are more operations involving possibly long integers, since both the Gram and basis matrices have to be updated. One may however forget about the basis matrix during the execution of the algorithm by computing the transformation instead, and applying the overall transformation to the initial basis: it then has to be determined from the input which would be cheaper between computing with the transformation matrix and computing with the basis matrix.

So far, we have eight possibilities. A very frequent one is dpes without the Gram matrix, which corresponds to NTL's `LLL_XD` routine. This choice can be sped up by factoring the exponents of the dpes: the idea is to have one common exponent per vector, instead of n exponents. To do this, we consider $e_i = \lfloor 1 + \log \max_{j \leq n} |\mathbf{b}_i[j]| \rfloor$ together with the vector $2^{-\ell} \lfloor \mathbf{b}_i \cdot 2^{\ell - e_i} \rceil$. More summing cancellations are likely to occur than without factoring the exponents (since we may lose some information by doing so), but we obtain a variant which is essentially as fast as using the processor double precision only, while remaining usable for large matrix inputs.

1.4.3 A Thoughtful Wrapper

In a complete implementation of LLL, the choice of the variant and the transitions between variants should be oblivious to the user. When calling the LLL routine, the user expects the execution to terminate and to return a guaranteed answer. At the time of the publishing of this survey, such a routine is available only in Magma and `fplll`: using the LLL routines in the other libraries requires, to some extent, some understanding of the algorithms used. To obtain an LLL routine which is guaranteed but also makes use of heuristics, misbehaviours should be detected and interpreted in such a way that the cheapest variant that is likely to work is chosen. In the Schnorr-Euchner algorithm, two such detections already exist: scalar product cancellations and too large GSO coefficients.

When considering floating-point LLL algorithms, the main source of infinite looping is the lazy size-reduction (Steps 4 to 8 in Figure 1.5). It is detected by watching if the $\mu_{i,j}$'s appearing are indeed decreasing at each loop iteration of the size-reduction. If this stops being the case, then something incorrect is happening. The other source of infinite looping is the succession of incorrect Lovász tests. Fortunately, the proof of the LLL algorithm provides an upper bound to the number of Lovász tests performed during the execution, as a function of the input basis. One can test whether the current number of Lovász tests is higher than this upper bound. This is a crude upper bound, but this malfunction seems to be much less frequent than the incomplete size-reduction.

In Figure 1.7, we give an overview of the reduction strategy in the LLL routine of Magma. Each box corresponds to a floating-point LLL using the Gram matrix or not, and using one of the afore-mentioned floating-point arithmetics. When a variant fails, another is tried, following one of the arrows. In addition to this graph, one should re-run a provable variant at the end of the execution if it succeeded with a heuristic one, since the output might then be incorrect. Other boxes and arrows than the ones displayed may be added. For example, one may stop using the factored exponents variants if the entries of the basis matrix start being small: in this case, the doubles without the Gram matrix will be more efficient.

When a malfunction is detected (by a non-decrease of the GSO coefficients during a size-reduction or by a too large number of Lovász tests), another variant must be selected. Essentially two problems can occur: cancellations of scalar products and lack of precision for the GSO calculations. The first trouble may occur in any dimension, while the second one can only occur when the dimension increases: around $d = 30$ in the worst case and around $d = 180$ on the average case, for close to optimal LLL parameters δ and η (for a heuristic explanation of the last figure, see Section 1.5). As a consequence, if a misbehaviour is detected in a low dimension or for a small LLL index κ (the magnitude of the floating-point errors essentially depends on κ, see the first-order analysis of the end of Section 1.3), cancellations in scalar products are likely to be the cause of the problem and one should start using the Gram matrix. Otherwise, it is likely that the mantissa size is not sufficient. In Figure 1.7, this choice is represented by the arrows with the labels "Small κ" and "Large κ".

The labels "Large matrix entries" and "Small matrix entries" denote the possibility of converting the Gram matrix coefficients to double precision floating-point numbers: the top boxes do not involve the Gram matrices, but those matrices may be needed later on if misbehaviours occur.

As mentioned earlier, in order to guarantee the correctness of the output, one has to re-run the most reliable (and thus slower) variant on the output. This can dominate the overall cost, especially if we are given an already reduced basis. Villard [63] recently introduced a method to certify that a given

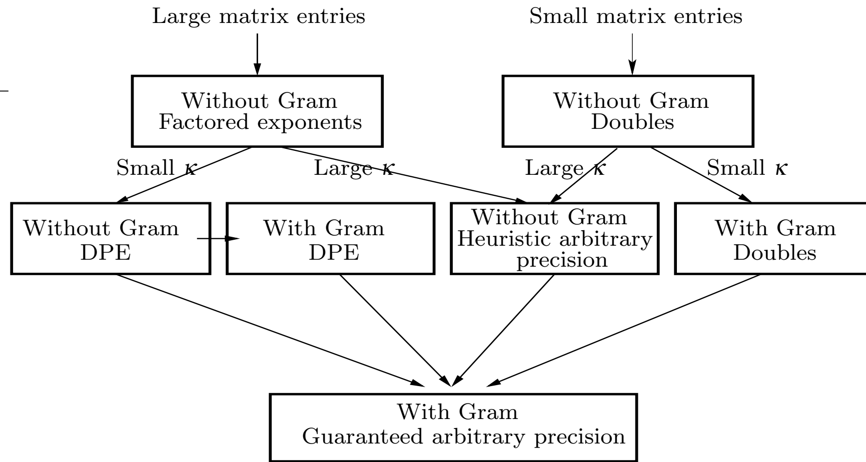


Fig. 1.7 Overview of the LLL reduction strategy in Magma.

basis is reduced. It will not always work, but if it does the result is guaranteed. It can be made very efficient (for example by using double precision floating-point numbers), and indeed much faster than using the provable precision in the L^2 algorithm. The general principle is as follows:

1. Compute an approximation $\bar{R}$ of the R-factor R of the QR-factorisation of the basis matrix.
2. Certify that the approximation $\bar{R}$ is indeed close to R, by using a result of [61] showing that it suffices to bound the spectral radius of some related matrix.
3. Check the LLL conditions in a certified way from the certified approximation $\bar{R}$ of R.

Another drawback of the general strategy is that it always goes towards a more reliable reduction. It may be that such a reliable variant is needed at some moment and becomes superfluous after some time within an execution: generally speaking, the reduction of the basis improves the accuracy of the computations and therefore some precautions may become superfluous. One would thus have to devise heuristic tests to decide if one should change for a more heuristic but faster variant. For example, suppose we did start using the Gram matrix before scalar product cancellations were detected. The most annoying scalar product cancellations occur when some vectors have very unbalanced lengths and are at the same time fairly orthogonal. One can check with the Gram matrix if it remains the case during the execution of the chosen variant. Suppose now that we did increase the floating-point precision. This was done in particular because the basis was not orthogonal enough. It may happen that it becomes significantly more orthogonal, later within the

LLL-reduction: this can be detected by looking at the decrease of the $\|\mathbf{b}_i^*\|$'s.

Finally, one may try to adapt the η and δ parameters in order to speed up the LLL reduction. If one is only interested in a reduced basis without paying attention to the LLL factors δ and η, then one should try the fastest pair, while still requiring only double precision. The first requirement usually implies a weakening of the pair (δ further away from 1 and η further away from 1/2), whereas the second one involves a strengthening, so that there is a trade-off to be determined. Furthermore, one may also try to change the LLL factors during the LLL reduction itself, for example starting with weak LLL factors to perform most of the reduction efficiently and strengthen the factors afterwards to provide a basis of a better quality.

1.4.4 Adapting the Algorithm to Particular Inputs

It is possible to adapt the algorithm to particular lattice basis inputs that occur frequently. We give here an example of a dedicated strategy called early size-reduction, which was initially introduced by Allan Steel. The computational saving of this method can easily be explained for input lattice bases of the following shape (they arise for example for detecting small integer relations between numbers):

$$\begin{pmatrix} a_1 & a_2 & a_3 & \dots & a_d \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix},$$

where the a_i's have large magnitudes. Let $A = \max_i |a_i|$. The idea of the early size-reduction is as follows: when the LLL index κ reaches a new value for the first time, instead of only size-reducing the vector $\mathbf{b}_\kappa$ with respect to the vectors $\mathbf{b}_1, \dots, \mathbf{b}_{\kappa-1}$, reduce the $\mathbf{b}_i$'s with respect to the vectors $\mathbf{b}_1, \dots, \mathbf{b}_{\kappa-1}$ for all $i \geq \kappa$. The speed-up is higher for the longest $\mathbf{b}_i$'s, so that it may be worth restricting the strategy to these ones.

One may think at first sight that this variant is going to be more expensive: in fact, the overall size-reduction of any input vector $\mathbf{b}_i$ will be much cheaper. In the first situation, if the first $i-1$ vectors behave fairly randomly, we will reduce in dimension i a vector of length $\approx A$ with respect to $i-1$ vectors of length $\approx A^{\frac{1}{i-1}}$: if the first $i-1$ vectors behave randomly, the lengths of the reduced vectors are all approximately the $(i-1)$-th root of the determinant of the lattice they span, which is itself approximately A. In the second situation, we will:

- Reduce in dimension 3 a vector of length $\approx A$ with respect to 2 vectors of length $\approx A^{\frac{1}{2}}$, when κ reaches 3 for the first time.
- Reduce in dimension 4 a vector of length $\approx A^{\frac{1}{2}}$ with respect to 3 vectors of length $\approx A^{\frac{1}{3}}$, when κ reaches 4 for the first time.
- ...
- Reduce in dimension i a vector of length $\approx A^{\frac{1}{i-2}}$ with respect to $i-1$ vectors of length $\approx A^{\frac{1}{i-1}}$, when κ reaches i for the first time.

We gain much because most of the time the number of non-zero coordinates is less than i.

We now describe a very simple dedicated strategy for lattice bases occurring in Coppersmith's method for finding the small roots of a polynomial modulo an integer [16]. We consider the univariate case for the sake of simplicity. In this application of LLL, the input basis vectors are made of the weighted coefficients of polynomials $(P_i(x))_i$: the i-th basis vector is made of the coordinates of $P_i(xX)$, where X is the weight. This implies that the $(j+1)$-th coordinates of all vectors are multiples of X^j. Rather than reducing a basis where the coordinates share large factors, one may consider the coordinates of the $P_i(x)$'s themselves and modify the scalar product by giving a weight X^j to the $(j+1)$-th coordinate. This decreases the size of the input basis with a negligible overhead on the computation of the scalar products. If X is a power of 2, then this overhead can be made extremely small.

1.5 Practical Observations on LLL

The LLL algorithm has been widely reported to perform much better in practice than in theory. In this section, we describe some experiments whose purpose is to measure this statement. These systematic observations were made more tractable thanks to the faster and more reliable floating-point LLLs based on L^2. Conversely, they also help improving the codes:

- They provide heuristics on what to expect from the bases output by LLL. For example, when LLL is needed for an application, these heuristic bounds may be used rather than the provable ones, which may decrease the overall cost. For example, in the cases of Coppersmith's method (see [38]) and the reconstruction of algebraic numbers (see [24]), the bases to be reduced will have smaller bit-lengths.
- They explain precisely which steps are expensive during the execution, so that the coder may be performing relevant code optimisations.
- They also help guessing which precision is likely to work in practice if no scalar product cancellation occurs, which helps choosing a stronger variant in case a malfunction is detected (see Subsection 1.4.3).

Overall, LLL performs quite well compared to the worst-case bounds with respect to the quality of the output: the practical approximation factor between the first basis vector and a shortest lattice vector remains exponential, but the involved constant is significantly smaller. Moreover, the floating-point LLLs also seem to outperform the worst-case bounds with respect to their running-time and the floating-point precision they require. We refer to [44] for more details about the content of this section. Further and more rigorous explanations of the observations can be found in [62].

1.5.1 The Lattice Bases Under Study

The behaviour of LLL can vary much with the type of lattice and the type of input basis considered. For instance, if the lattice minimum is extremely small compared to the other lattice minima (the k-th minimum being the smallest R such that there are $\geq k$ linearly independent lattice vectors of length $\leq R$), the LLL algorithm will find a vector whose length reaches it (which is of course not the case in general). If the basis to be reduced is or is close to being LLL-reduced, the LLL algorithm will not behave generically. For instance, if one selects vectors uniformly and independently in the d-dimensional hypersphere, they are close to be reduced with high probability (see [6, 7] and the survey [62] describing these results in a more general probabilistic setup). We must therefore define precisely what we will consider as input.

First of all, there exists a natural notion of random lattice. A full-rank lattice (up to scaling) can be seen as an element of $SL_d(\mathbb{R})/SL_d(\mathbb{Z})$. The space $SL_d(\mathbb{R})$ inherits a Haar measure from $\mathbb{R}^{d^2}$, which projects to a finite measure when taking the quotient by $SL_d(\mathbb{Z})$ (see [3]). One can therefore define a probability measure on real-valued lattices. There are ways to generate integer valued lattices so that they converge to the uniform distribution (with respect to the Haar measure) when the integer parameters grow to infinity. For example, Goldstein and Mayer [22] consider the following random family of lattices: take a large prime p, choose $d-1$ integers $x_2, \ldots, x_d$ randomly, independently and uniformly in $[0, p-1]$, and consider the lattice spanned by the columns of the following $d \times d$ matrix:

$$\begin{pmatrix} p & x_2 & x_3 & \ldots & x_d \\ 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 0 & 1 \end{pmatrix}.$$

Amazingly, these lattice bases resemble those arising from knapsack-type problems, the algebraic reconstruction problem (finding the minimal poly-

nomial of an algebraic number given a complex approximation to it) and the problem of detecting integer relations between real numbers [24]. We define knapsack-type lattice bases as follows: take a bound B, choose d integers $x_1, \ldots, x_d$ randomly, independently and uniformly in $[0, B-1]$ and consider the lattice spanned by the columns of the following $(d+1) \times d$ matrix:

$$\begin{pmatrix} x_1 & x_2 & x_3 & \ldots & x_d \\ 1 & 0 & 0 & \ldots & 0 \\ 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & 1 \end{pmatrix}.$$

In our experiments, we did not notice any difference of behaviour between these random bases and the random bases of Goldstein and Mayer. Similarly, removing the second row or adding another row of random numbers do not seem to change the observations either.

We will also describe experiments based on what we call Ajtai-type bases. Similar bases were introduced by Ajtai [4] to prove a lower-bound on the quality of Schnorr's block-type algorithms [49]. Select a parameter a. The basis is given by the columns of the $d \times d$ upper-triangular matrix B such that $B_{i,i} = \lfloor 2^{(2d-i+1)^a} \rceil$ and the $B_{j,i}$'s (for $i > j$) are randomly, independently and uniformly chosen in $\mathbb{Z} \cap [-B_{j,j}/2, \ldots, B_{j,j}/2)$. The choice of the function $2^{(2d-i+1)^a}$ is arbitrary: one may generalise this family by considering a real-valued function $f(i,d)$ and by taking $B_{i,i} = \lfloor f(i,d) \rceil$. One advantage of choosing $f(i,d) = 2^{(2d-i+1)^a}$ is that the $\|\mathbf{b}_i^*\|$'s are decreasing very quickly, so that the basis is far from being reduced.

1.5.2 The Output Quality

In low dimensions, it has been observed for quite a while that the LLL algorithm computes vectors whose lengths are close (if not equal) to the lattice minimum [45]. Hopefully for lattice-based cryptosystems, this does not remain the case when the dimension increases.

By experimenting, one can observe that the quality of the output of LLL is similar for all input lattice bases generated from the different families mentioned above. For example, in Figure 1.8, each point corresponds to the following experiment: generate a random knapsack-type basis with $B = 2^{100 \cdot d}$ and reduce it with the L^2 algorithm, with $(\delta, \eta) = (0.999, 0.501)$; a point corresponds to the value of $\frac{1}{d} \log_2 \frac{\|\mathbf{b}_1\|}{\mathrm{vol}(L)^{1/d}}$ for the corresponding returned basis. We conclude that experimentally, it seems that for a growing dimension d, the first output vector is such that:

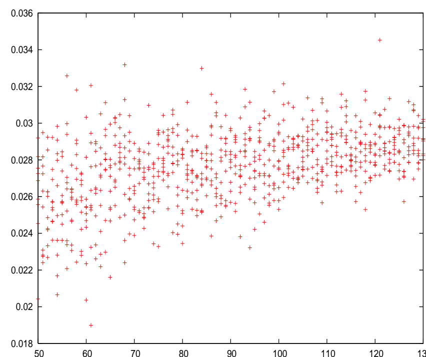


Fig. 1.8 Samples of $\frac{1}{d}\log_2 \frac{\|\mathbf{b}_1\|}{\mathrm{vol}(L)^{1/d}}$ for increasing dimensions d.

$$\|\mathbf{b}_1\| \approx c^d \cdot \mathrm{vol}(L)^{1/d},$$

where $c \approx 2^{0.03} \approx 1.02$. The exponential factor 1.02^d remains tiny even in moderate dimensions: e.g., $(1.02)^{50} \approx 2.7$ and $(1.02)^{100} \approx 7.2$.

One may explain this global phenomenon on the basis by looking at the local two-dimensional bases, i.e., the pairs $(\mathbf{b}^*_{i-1}, \mathbf{b}^*_i + \mu_{i,i-1}\mathbf{b}^*_{i-1})$. If we disregard some first and last local pairs, then all the others seem to behave quite similarly. In Figure 1.9, each point corresponds to a local pair (its coordinates being $\left(\mu_{i,i-1}, \|\mathbf{b}^*_i\|/\|\mathbf{b}^*_{i-1}\|\right)$) of a basis that was reduced with `fplll` with parameters $\delta = 0.999$ and $\eta = 0.501$, starting from a knapsack-type basis with $B = 2^{100\cdot d}$. These observations seem to stabilise between the dimensions 40 and 50: it behaves differently in low dimensions (in particular, the quantity $\frac{1}{d}\log_2 \frac{\|\mathbf{b}_1\|}{\mathrm{vol}(L)^{1/d}}$ is lower), and converges to it progressively when the dimension increases. The mean value of the $|\mu_{i,i-1}|$'s is close to 0.38, and the mean value of $\frac{\|\mathbf{b}^*_{i-1}\|}{\|\mathbf{b}^*_i\|}$ is close to 1.04, which matches the above constant 1.02. One may wonder if the geometry of "average" LLL-reduced bases is due to the fact that most LLL-reduced bases are indeed of this shape, or if the LLL algorithm biases the distribution. It is hard to decide between both possibilities: one would like to generate randomly and uniformly LLL-reduced bases of a given lattice, but it is unknown how to do it efficiently; for example, the number of LLL-reduced bases of a given lattice grows far too quickly when the dimension increases.

On the right hand-side of Figure 1.9, we did the same experiment except that we replaced LLL by the Schnorr-Euchner deep insertion algorithm [53] (see also [48]), which is a variant of the LLL algorithm where the Lovász condition is changed into the stronger requirement:

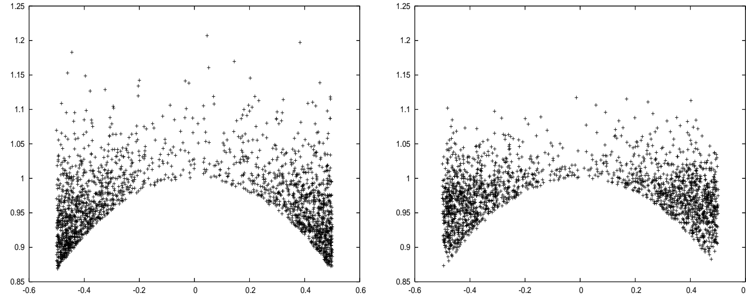


Fig. 1.9 Distribution of the local bases after LLL (left) and deep-LLL (right).

$$\forall \kappa \leq d, \forall i < \kappa, \quad \delta \cdot \|\mathbf{b}_i^*\|^2 \leq \left\| \mathbf{b}_\kappa^* + \sum_{j=i}^{\kappa-1} \mu_{\kappa,j} \mathbf{b}_j^* \right\|^2 .$$

The quality of the local bases improves by considering the deep insertion algorithm, the constant 1.04 becoming ≈ 1.025, for close to optimal parameters δ and η. These data match the observations of [9] on the output quality improvement obtained by considering the deep insertion algorithm.

1.5.3 Practical Running-Time

The floating-point LLL algorithms seem to run much faster in practice than the worst-case theoretical bounds. We argue below that these bounds should be reached asymptotically for some families of inputs. We also heuristically explain why they terminate significantly faster in practice. We will consider bases for which $n = \Theta(d) = O(\log B)$, so that the worst-case bound given in Theorem 2 is simplified to $O(d^5 \log^2 B)$.

The worst-case analysis of the L^2 algorithm given in Section 1.3 seems to be tight for Ajtai-type random bases. More precisely: if $a > 1$ is fixed and d grows to infinity, the average bit-complexity of the L^2 algorithm given as input a randomly and uniformly chosen d-dimensional Ajtai-type basis with parameter a seems to be $\Theta(d^{5+2a})$ (in this context, we have $\log B \approx d^a$).

When L^2 is run on these input bases, all the bounds of the heuristic analysis but one seem tight, the exception being the $O(d)$ bound on the size of the X's (computed at Step 6 of the L^2 algorithm, as described in Figure 1.5). Firstly, the $O(d^2 \log B)$ bound on the loop iterations seems to be tight in practice, as suggested by Figure 1.10. The left side of the figure corresponds to Ajtai-type random bases with $a = 1.2$: the points are the experimental data and the continuous line is the `gnuplot` interpolation of the form $f(d) = c_1 \cdot d^{3.2}$. The right side of the figure has been obtained similarly, for $a = 1.5$, and $g(d) =$

$c_2 \cdot d^{3.5}$. With Ajtai-type bases, size-reductions rarely contain more than two iterations. For example, for $d \leq 75$ and $a = 1.5$, less than 0.01% of the size-reductions involve more than two iterations. The third bound of the heuristic worst case analysis, i.e., the number of arithmetic operations within each loop iteration of the lazy size-reduction, is also reached.

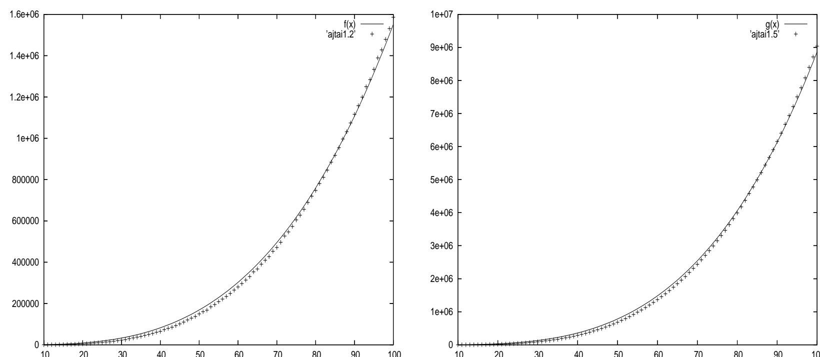


Fig. 1.10 Number of loop iterations of L^2 for Ajtai-type random bases.

These similarities between the worst and average cases do not go on for the size of the integers involved in the arithmetic operations. The X's computed during the size-reductions are most often shorter than a machine word, which makes it difficult to observe the $O(d)$ factor in the complexity bound coming from their sizes. For an Ajtai-type basis with $d \leq 75$ and $a = 1.5$, less than 0.2% of the non-zero x_i's are longer than 64 bits. In the worst case [43] and for close to optimal parameters δ and η, we have $|x_i| \lesssim (3/2)^{\kappa-i}M$, where M is the maximum of the $|\mu_{\kappa,j}|$'s before the lazy size-reduction starts, and κ is the current LLL index. In practice, M happens to be small most of the time. It is essentially the length ratio between $\mathbf{b}_\kappa$ and the smallest of the $\mathbf{b}_i$'s for $i \leq \kappa$: it is very rare that the lengths of the basis vectors differ significantly in a non-negligible number of loop iterations during an execution of the LLL algorithm. It can be argued (see [44] for more details), using the generic geometry of LLL-reduced bases described previously, that the average situation is $|X| \approx (1.04)^{\kappa-i}M$ if X is derived from $\mu_{\kappa,i}$. This bound remains exponential, but for a small M, the integer X becomes larger than a machine word only in dimensions higher than several hundreds. We thus expect the $|X|$'s to be of length $\lesssim (\log_2 1.04) \cdot d \approx 0.057 \cdot d$. For example, the quantity $(1.04)^d$ becomes larger than 2^{64} for $d \geq 1100$. Since it is not known how to reduce lattice bases which simultaneously have such huge dimensions and reach the other bounds of the heuristic worst-case complexity analysis, it is at the moment impossible to observe the asymptotic behaviour. The practical running time is rather $O(d^4 \log^2 B)$.

One can take advantage of the fact that most X's are small by optimising the operation $\mathbf{b}_\kappa := \mathbf{b}_\kappa + X\mathbf{b}_i$ for small X's. For example, one may consider the cases $X \in \{-2,-1,1,2\}$ separately. One may also use fast multiply-and-add operations, as available in Pierrick Gaudry's GNU MP patch for AMD 64 processors[2].

Furthermore, in many situations, a much better running-time can be observed. Of course, it highly depends on the input basis: if it is already reduced, it will terminate very quickly since there will be no more than d loop iterations. But this can also happen for bases that are very far from being reduced, such as knapsack-type bases. In this case, two facts improve the running-time of LLL. Firstly, the number of loop iterations is only $O(d \log B)$ instead of $O(d^2 \log B)$ in the worst case: this provably provides a $O(d^4 \log^2 B)$ worst-case complexity bound for these lattice bases (from Theorem 2). Secondly the basis matrix entries become very short much sooner than usual: if the maximum value of the index κ so far is some j, one can heuristically expect the entries of the vectors $\mathbf{b}_i$ for $i < j$ to be of length $O\left(\frac{1}{j-1}\log B\right)$ (see [44] for more details). It is not known yet how to use this second remark to decrease the complexity bound in a rigorous way, but one can heuristically expect the following worst-case bound:

$$O\left(d \log B \cdot d^2 \cdot d \cdot \frac{\log B}{d}\right) = O(d^3 \log^2 B).$$

Finally, by also considering the $\Theta(d)$ gain due to the fact that the size-reduction X's do not blow up sufficiently for dimensions that can be handled with today's LLL codes, we obtain a $O(d^2 \log^2 B)$ bound in practice (for a very large $\log B$).

1.5.4 Numerical Behaviour

We now describe the practical error amplification of the GSO coefficients in the L^2 algorithm. To build the wrapper described in Subsection 1.4.3, it is important to be able to guess up to which dimension a given precision will work, for example the double precision, which is much faster than arbitrary precision. Is it possible to predict the chance of success when using double precision? We suppose here that we work with close to optimal parameters, i.e., δ close to 1 and η close to $1/2$, and with some precision ℓ lower than the provably sufficient $\approx \log_2(3) \cdot d$ precision. We do not consider the cancellations that may arise in the scalar product computations. Two problems may occur: some lazy size-reduction or some consecutive Lovász tests may be looping forever. In both situations, the misbehaviour is due to the incorrectness of

[2] `http://www.loria.fr/~gaudry/mpn_AMD64/`

the involved approximate Gram-Schmidt coefficients. The output basis may also be incorrect, but most often if something goes wrong, the execution loops within a size-reduction.

In practice the algorithm seems to work correctly with a given precision for much larger dimensions than guaranteed by the worst-case analysis: for example, double precision seems sufficient most of the time up to dimension 170. This figure depends on the number of loop iterations performed: if there are fewer loop iterations, one can expect fewer large floating-point errors since there are fewer floating-point calculations. It can be argued that the average required precision grows linearly with the dimension, but with a constant factor significantly smaller than the worst-case one: for close to optimal LLL parameters and for most input lattice bases, L^2 behaves correctly with a precision of $\approx 0.25 \cdot d$ bits. The heuristic argument, like in the previous subsection, relies on the generic geometry of LLL-reduced bases.

1.6 Open Problems

Though studied and used extensively since 25 years, many questions remain open about how to implement the LLL algorithm as efficiently as possible and about its practical behaviour. Some open problems have been suggested along this survey. For example, Section 1.5 is essentially descriptive (though some relations between the diverse observations are conjectured), and obtaining proven precise results would help to understand more clearly what happens in practice and how to take advantage of it: the survey [62] formalises more precisely these questions and answers some of them. We suggest here a few other lines of research related to the topics that have been presented.

Decreasing the required precision in floating-point LLLs. Since processor double precision floating-point numbers are drastically faster than other floating-point arithmetics (in particular arbitrary precision), it is tempting to extend the family of inputs for which double precision will suffice. One way to do this, undertaken by Schnorr [33, 51, 48], is to use other orthogonalisation techniques like the Givens and Householder algorithms. These algorithms compute the Q (as a product of matrices) and R factors of the QR-factorisation of the basis matrix. The L^2 algorithm relies on the Cholesky factorisation (computing the R factor from the Gram matrix). Unfortunately, the condition number of the Cholesky factorisation is essentially the square of the condition number of the QR-factorisation (see [26] for more details). With fully general input matrices, this heuristically means that one needs approximately twice the precision with Cholesky's factorisation than with the QR-factorisation. Another significant advantage of relying on the QR-factorisation rather than Cholesky's is that the Gram matrix becomes superfluous: a large ratio of the integer operations can thus be avoided, which should provide better running-times, at least for input matrices having di-

mensions that are small compared to the bit-sizes of their entries. Nevertheless, the Householder and Givens algorithms have at least two drawbacks. Firstly, they require more floating-point operations: for $d \times d$ matrices, the Householder algorithm requires $\frac{4}{3}d^3 + o(d^3)$ floating-point operations whereas the Cholesky algorithm requires only $\frac{1}{3}d^3 + o(d^3)$ floating-point operations (see [26] for more details). And secondly, they suffer from potential cancellations while computing scalar products (the first of the three drawbacks of the naive floating-point LLL of Section 1.3). A reduction satisfying our definition of LLL-reduction seems unreachable with these orthogonalisation techniques. In [51], Schnorr suggests to replace the size-reduction condition $|\mu_{i,j}| \leq \eta$ by $|\mu_{i,j}| \leq \eta + \varepsilon \frac{\|\mathbf{b}_i^*\|}{\|\mathbf{b}_j^*\|}$ for some small $\varepsilon > 0$. So far, the best results in this direction remain heuristic [51, 48]: making them fully provable would be a significant achievement. It would prove that one can double the dimension up to which the double precision rigorously suffices, and provide a sounder insight on the possibilities of such orthogonalisation techniques in practice.

To decrease the precision even further, one could strengthen the orthogonality of the bases that we are reducing. To do this, deep insertions [53] (see also Section 1.5) may be used, but this may become slow when the dimension increases. Another alternative would be to perform a block-type reduction (such as described in [49, 19]), for some small size of block: one performs strong reductions such as Hermite-Korkine-Zolotarev (HKZ for short) or dual-HKZ to make these small blocks extremely reduced and thus extremely orthogonal. Indeed, a small size of block is sufficient to strengthen the overall orthogonality of the basis, and if the block-size is small enough, the actual cost of HKZ-reducing for this block-size remains dominated by the size-reduction step. Asymptotically, a block-size $k = \Theta\left(\frac{\log d}{\log\log d}\right)$ would satisfy these requirements. In practice, a block-size below 15 does not seem to create a large running-time overhead.

Using floating-point arithmetic in other lattice algorithms. Replacing the text-book rational arithmetic by an approximate floating-point arithmetic can lead to drastic theoretical and practical speed-ups. The counterpart is that the correctness proofs become more intricate. One may extend the error analysis strategy of the L^2 algorithm to derive complete (without neglected error terms) explicit error bounds for modifications of the LLL algorithm such as the algorithm of Schönhage [55], the Strong Segment-LLL of Koy and Schnorr [32]. Adapting these algorithms to floating-point arithmetic has already been considered [33, 51], but often the provably sufficient precision is quite large in the worst case (linear in the bit-size of the matrix entries), though better heuristic bounds outperform those of L^2 (see [51] and the survey [48] in this book). Developing high-level techniques to prove such bounds would be helpful. Secondly, some lattice reduction algorithms such as short lattice point enumeration, HKZ reduction and block-type reductions [29, 18, 25, 53, 19] are usually implemented with floating-point numbers, though no analysis at all has been made. This simply means that the

outputs of these codes come with no correctness guarantee. This fact is particularly annoying, since checking the solutions of these problems is often very hard. Amazingly, devising strong reduction algorithms based on floating-point arithmetic may help decreasing the precision required for the LLL-reduction, as mentioned above.

Decreasing the linear algebra cost. In all known LLL algorithms, the embedding dimension n is a factor of the overall cost. This comes from the fact that operations are performed on the basis vectors, which are made of n coordinates. This may not seem natural, since one could reduce the underlying quadratic form (i.e., LLL-reduce by using only the Gram matrix), store the transformation matrix, and finally apply it to the initial basis. Then the cost would be a smaller function of n. We describe here a possible way to reduce a lattice basis whose embedding dimension n is much larger than its rank d. It consists in applying a random projection (multiply the embedding space by a random $d \times n$ matrix), reducing the projected lattice, and applying the obtained transformation to the initial basis: one then hopes that the obtained lattice basis is somehow close to being reduced, with high probability. Results in that direction are proved in [8]. This strategy can be seen as a dual of the probabilistic technique recently introduced by Chen and Storjohann [13] to decrease the number of input vectors when they are linearly dependent: their technique decreases the number of input vectors while the one above decreases the number of coordinates of the input vectors.

Decreasing the integer arithmetic cost. Finally, when the size of the matrix entries is huge and the dimension is small, one would like to have an algorithm with a sub-quadratic bit-complexity (with respect to the size of the entries). Both Euclid's and Gauss' algorithms have quasi-linear variants (see [30, 54, 64, 56]): is it possible to devise a LLL algorithm which is quasi-linear in any fixed dimension? Eisenbrand and Rote [17] answered the question positively, but the cost of their algorithm is more than exponential with respect to d. So we may restate the question as follows: is it possible to devise a LLL algorithm whose bit-complexity grows quasi-linearly with the size of the entries and polynomially with the dimension?

Acknowledgements. The author gratefully thanks John Cannon, Claude-Pierre Jeannerod, Erich Kaltofen, Phong Nguyen, Andrew Odlyzko, Peter Pearson, Claus Schnorr, Allan Steel, Brigitte Vallée and Gilles Villard for helpful discussions and for pointing out errors on drafts of this work.

References

1. LIDIA 2.1.3. A C++ library for computational number theory. Available at http://www.informatik.tu-darmstadt.de/TI/LiDIA/.

2. IEEE Standards Committee 754. ANSI/IEEE standard 754-1985 for binary floating-point arithmetic. Reprinted in SIGPLAN Notices, 22(2):9–25, 1987.
3. M. Ajtai. Random lattices and a conjectured 0-1 law about their polynomial time computable properties. In *Proceedings of the 2002 Symposium on Foundations of Computer Science (FOCS 2002)*, pages 13–39. IEEE Computer Society Press, 2002.
4. M. Ajtai. The worst-case behavior of Schnorr's algorithm approximating the shortest nonzero vector in a lattice. In *Proceedings of the 35th Symposium on the Theory of Computing (STOC 2003)*, pages 396–406. ACM Press, 2003.
5. M. Ajtai and C. Dwork. A public-key cryptosystem with worst-case/average-case equivalence. In *Proceedings of the 29th Symposium on the Theory of Computing (STOC 1997)*, pages 284–293. ACM Press, 1997.
6. A. Akhavi. Worst-case complexity of the optimal LLL algorithm. In *Proceedings of the 2000 Latin American Theoretical Informatics conference (LATIN 2000)*, volume 1776 of *Lecture Notes in Computer Science*, pages 355–366. Springer-Verlag, 2000.
7. A. Akhavi, M.-F. Marckert, and A. Rouault. On the reduction of a random basis. In *Proceedings of the 4th Workshop on Analytic Algorithmics and Combinatorics*. SIAM Publications, 2007.
8. A. Akhavi and D. Stehlé. Speeding-up lattice reduction with random projections (extended abstract). In *Proceedings of the 2008 Latin American Theoretical Informatics conference (LATIN'08)*, volume 4957 of *Lecture Notes in Computer Science*, pages 293–305. Springer-Verlag, 2008.
9. W. Backes and S. Wetzel. Heuristics on lattice reduction in practice. *ACM Journal of Experimental Algorithms*, 7:1, 2002.
10. D. Boneh and G. Durfee. Cryptanalysis of RSA with private key d less than $N^{0.292}$. *IEEE Transactions on Information Theory*, 46(4):233–260, 2000.
11. W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system. I. The user language. *Journal of Symbolic Computation*, 24(3–4):235–265, 1997.
12. D. Cadé and D. Stehlé. fplll-2.0, a floating-point LLL implementation. Available at `http://perso.ens-lyon.fr/damien.stehle`.
13. Z. Chen and A. Storjohann. A BLAS based C library for exact linear algebra on integer matrices. In *Proceedings of the 2005 International Symposium on Symbolic and Algebraic Computation (ISSAC'02)*, pages 92–99. ACM Press, 2005.
14. D. Coppersmith. Finding a small root of a bivariate integer equation. In *Proceedings of Eurocrypt 1996*, volume 1070 of *Lecture Notes in Computer Science*, pages 178–189. Springer-Verlag, 1996.
15. D. Coppersmith. Finding a small root of a univariate modular equation. In *Proceedings of Eurocrypt 1996*, volume 1070 of *Lecture Notes in Computer Science*, pages 155–165. Springer-Verlag, 1996.
16. D. Coppersmith. Small solutions to polynomial equations, and low exponent RSA vulnerabilities. *Journal of Cryptology*, 10(4):233–260, 1997.
17. F. Eisenbrand and G. Rote. Fast reduction of ternary quadratic forms. In *Proceedings of the 2001 Cryptography and Lattices Conference (CALC'01)*, volume 2146 of *Lecture Notes in Computer Science*, pages 32–44. Springer-Verlag, 2001.
18. U. Fincke and M. Pohst. A procedure for determining algebraic integers of given norm. In *Proceedings of EUROCAL*, volume 162 of *Lecture Notes in Computer Science*, pages 194–202, 1983.
19. N. Gama and P. Q. Nguyen. Finding short lattice vectors within Mordell's inequality. In *Proceedings of the 40th Symposium on the Theory of Computing (STOC'08)*. ACM, 2008.
20. J. von zur Gathen and J. Gerhardt. *Modern Computer Algebra, 2nd edition.* Cambridge University Press, 2003.

21. O. Goldreich, S. Goldwasser, and S. Halevi. Public-key cryptosystems from lattice reduction problems. In *Proceedings of Crypto 1997*, volume 1294 of *Lecture Notes in Computer Science*, pages 112–131. Springer-Verlag, 1997.
22. D. Goldstein and A. Mayer. On the equidistribution of Hecke points. *Forum Mathematicum*, 15:165–189, 2003.
23. T. Granlund. The GNU MP Bignum Library. Available at `http://gmplib.org/`.
24. G. Hanrot. LLL: a tool for effective diophantine approximation. This book.
25. B. Helfrich. Algorithms to construct Minkowski reduced and Hermite reduced lattice bases. *Theoretical Computer Science*, 41:125–139, 1985.
26. N. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM Publications, 2002.
27. J. Hoffstein, J. Pipher, and J. H. Silverman. NTRU: a ring based public key cryptosystem. In *Proceedings of the 3rd Algorithmic Number Theory Symposium (ANTS III)*, volume 1423 of *Lecture Notes in Computer Science*, pages 267–288. Springer-Verlag, 1998.
28. 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.
29. R. Kannan. Improved algorithms for integer programming and related lattice problems. In *Proceedings of the 15th Symposium on the Theory of Computing (STOC 1983)*, pages 99–108. ACM Press, 1983.
30. D. Knuth. The analysis of algorithms. In *Actes du Congrès International des Mathématiciens de 1970*, volume 3, pages 269–274. Gauthiers-Villars, 1971.
31. D. Knuth. *The Art of Computer Programming, vol. 2, third edition*. Addison-Wesley, 1997.
32. H. Koy and C. P. Schnorr. Segment LLL-reduction of lattice bases. In *Proceedings of the 2001 Cryptography and Lattices Conference (CALC'01)*, volume 2146 of *Lecture Notes in Computer Science*, pages 67–80. Springer-Verlag, 2001.
33. H. Koy and C. P. Schnorr. Segment LLL-reduction of lattice bases with floating-point orthogonalization. In *Proceedings of the 2001 Cryptography and Lattices Conference (CALC'01)*, volume 2146 of *Lecture Notes in Computer Science*, pages 81–96. Springer-Verlag, 2001.
34. J. C. Lagarias and A. M. Odlyzko. Solving low-density subset sum problems. *Journal of the ACM*, 32:229–246, 1985.
35. A. K. Lenstra, H. W. Lenstra, Jr., and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:513–534, 1982.
36. H. W. Lenstra, Jr. Flags and lattice basis reduction. In *Proceedings of the third European congress of mathematics, volume 1*. Birkhäuser, 2001.
37. J. Martinet. *Perfect Lattices in Euclidean Spaces*. Springer-Verlag, 2002.
38. A. May. Using LLL-reduction for solving RSA and factorization problems: a survey. This book.
39. A. May. *New RSA Vulnerabilities Using Lattice Reduction Methods*. PhD thesis, University of Paderborn, 2003.
40. J.-M. Muller. *Elementary Functions, Algorithms and Implementation*. Birkhäuser, 1997.
41. P. Nguyen and D. Stehlé. An LLL algorithm with quadratic complexity. *To appear in SIAM J. Comput.*
42. 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.
43. 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.

44. P. Nguyen and D. Stehlé. LLL on the average. In *Proceedings of the 7th Algorithmic Number Theory Symposium (ANTS VII)*, volume 4076 of *Lecture Notes in Computer Science*, pages 238–256. Springer-Verlag, 2006.
45. A. M. Odlyzko. The rise and fall of knapsack cryptosystems. In *Proceedings of Cryptology and Computational Number Theory*, volume 42 of *Proceedings of Symposia in Applied Mathematics*, pages 75–88. American Mathematical Society, 1989.
46. A. M. Odlyzko and H. J. J. te Riele. Disproof of Mertens conjecture. *Journal für die reine und angewandte Mathematik*, 357:138–160, 1985.
47. The SPACES Project. MPFR, a LGPL-library for multiple-precision floating-point computations with exact rounding. Available at `http://www.mpfr.org/`.
48. C. P. Schnorr. Hot topics of LLL and lattice reduction. This book.
49. C. P. Schnorr. A hierarchy of polynomial lattice basis reduction algorithms. *Theoretical Computer Science*, 53:201–224, 1987.
50. C. P. Schnorr. A more efficient algorithm for lattice basis reduction. *Journal of Algorithms*, 9(1):47–62, 1988.
51. C. P. Schnorr. Fast LLL-type lattice reduction. *Information and Computation*, 204:1–25, 2006.
52. C. P. Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. In *Proceedings of the 1991 Symposium on the Fundamentals of Computation Theory (FCT'91)*, volume 529 of *Lecture Notes in Computer Science*, pages 68–85. Springer-Verlag, 1991.
53. C. P. Schnorr and M. Euchner. Lattice basis reduction: improved practical algorithms and solving subset sum problems. *Mathematics of Programming*, 66:181–199, 1994.
54. A. Schönhage. Schnelle Berechnung von Kettenbruchentwicklungen. *Acta Informatica*, 1:139–144, 1971.
55. A. Schönhage. Factorization of univariate integer polynomials by Diophantine approximation and improved basis reduction algorithm. In *Proceedings of the 1984 International Colloquium on Automata, Languages and Programming (ICALP 1984)*, volume 172 of *Lecture Notes in Computer Science*, pages 436–447. Springer-Verlag, 1984.
56. A. Schönhage. Fast reduction and composition of binary quadratic forms. In *Proceedings of the 1991 International Symposium on Symbolic and Algebraic Computation (ISSAC'91)*, pages 128–133. ACM Press, 1991.
57. G. Schulz. Iterative berechnung der reziproken matrix. *Zeitschrift für Angewandte Mathematik und Mechanik*, 13:57–59, 1933.
58. I. Semaev. A 3-dimensional lattice reduction algorithm. In *Proceedings of the 2001 Cryptography and Lattices Conference (CALC'01)*, volume 2146 of *Lecture Notes in Computer Science*, pages 181–193. Springer-Verlag, 2001.
59. V. Shoup. NTL, Number Theory C++ Library. Available at `http://www.shoup.net/ntl/`.
60. A. Storjohann. Faster algorithms for integer lattice basis reduction. Technical report, ETH Zürich, 1996.
61. J.-G. Sun. Componentwise perturbation bounds for some matrix decompositions. *BIT Numerical Mathematics*, 31:341–352, 1992.
62. B. Vallée and A. Vera. Probabilistic analyses of lattice reduction algorithms. This book.
63. 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)*, pages 361–368. ACM Press, 2007.
64. C. K. Yap. Fast unimodular reduction: planar integer lattices. In *Proceedings of the 1992 Symposium on the Foundations of Computer Science (FOCS 1992)*, pages 437–446. IEEE Computer Society Press, 1992.